Shellcode Generation: A Resource Efficient Approach using Fine-tuned LLMs

Madhav Ram
Dept. of Computer Science and Engineering
Amrita Vishwa Vidhyapeetam
Bengaluru, India
bl.en.u4cse21117@bl.students.amrita.edu

Mohith Krishna V
Dept. of Computer Science and Engineering
Amrita Vishwa Vidhyapeetam
Bengaluru, India
bl.en.u4cse21125@bl.students.amrita.edu

M Pranavkrishnan
Dept. of Computer Science and Engineering
Amrita Vishwa Vidhyapeetam
Bengaluru, India
bl.en.u4cse21113@bl.students.amrita.edu

Priyanka C Nair
Dept. of Computer Science and Engineering
Amrita Vishwa Vidhyapeetam
Bengaluru, India
v_priyanka@blr.amrita.edu

Deepa Gupta
Dept. of Computer Science and Engineering
Amrita Vishwa Vidhyapeetam
Bengaluru, India
g_deepa@blr.amrita.edu

Abstract— **In order to create better shellcode for offensive cybersecurity, this study investigates the use of large language models (LLMs) such as Mistral and Llama. It focuses on LLM optimizations to improve shellcode accuracy and efficiency with the goal of quickly locating and taking advantage of software vulnerabilities. To optimize shellcode generation, several model parameters are tuned through controlled experiments, with the BLEU score serving as the primary criterion for impartial evaluation. This study achieved a BLEU-1 score of 0.8506, highlighting the effectiveness of the optimization and fine-tuning strategies. The study explores the subtle aspects of LLMs that influence the creation of shellcode, such as their capacity for learning, their ability to adjust to the peculiarities of cybersecurity, and their ability to replicate complex patterns into effective shellcode. This project intends to expand knowledge of using LLMs for shellcode development by submitting significant experimentation and research, providing insights into more efficient vulnerability exploitation strategies in cybersecurity.**

Keywords— Shellcode generation, large language models, Mistral, Llama, LLM finetuning, cybersecurity, LoRA

I. Introduction

Emerging as a key tool in cybersecurity, natural language processing enables creative approaches to threat detection, analysis, and reaction. Natural Language Processing (NLP) is useful in bridging the gap between machine-executable code and human intent, especially in the area of shellcode production. By using natural language processing, cybersecurity specialists may explain difficult coding jobs in plain English, which expedites development processes and promotes interdisciplinary cooperation. Moreover, by gleaning insightful information from unstructured textual data, natural language processing facilitates proactive threat intelligence and predictive analytics. One of the top applications of artificial intelligence, large language models are transforming a number of fields, including code creation and natural language processing. These state of the art neural network based designs, such as Mistral and Llama models, provide unmatched power for processing and producing natural language text. Large Language Models (LLMs) offer a powerful toolkit for producing exploit code from plain language descriptions in the context of creating shellcode. The text training gives LLMs a profound grasp of the syntax of assembly code and semantics of programming languages, which helps them write exceptionally accurate and high-quality code. This bridges the gap between code and natural language, by providing suitable code snippets by interpreting simple descriptions of offensive security requirements. It streamlines the testing process and opens up security research to a larger audience. With the help of this, analysts can explore and prototype quickly, transforming ideas into working exploits without having to spend time on the difficult process of writing assembly level exploits.

This study utilizes the expert crafted list of shellcode and intents from the "SoLID/shellcode_i_a32" dataset [1], which provides a basis for shellcode generation model training and assessment. This collection of shellcode samples offers a wide range of examples of simple Intel 32-bit assembly tasks and their respective snippets. This allows researchers to create models, scripts and other tools for thorough understanding, organization and generation of shellcode. Extensive testing yields insights into the complexities of shellcode production, which in turn helps design reliable exploit code. The dataset also makes benchmarking and comparative research possible, which promotes improvements in not only shellcode development but also in defensive cybersecurity. Performance measures are essential for assessing shellcode generation models, such as BLEU (Bilingual Evaluation Understudy). These metrics are used to measure correctness and similarity between generated shellcode with their reference implementations. Measures like execution time and resource usage provide information about scalability and efficiency of the LLMs utilized.

II. Related Work

This section is an examination of developments in exploit code generation utilizing NLP or LLM techniques and divides it into two main sections: development of Neural Machine Translation (NMT) models and finetuning Large Language Models. Previous research on shellcode that do not utilize Machine Learning models include printable shellcode generation by Patel et al. [3] which converts non-printable shellcode to a printable format using an encoding algorithm, automated payload transplants by Bao et al. [6] utilize symbolic execution and tracing and alphanumeric shellcode generation on ARM architecture, Kumar et al. [4]. Code generation via natural language can involve utilization of templates for automated code generation or creation of custom translation models. Neural Machine Translation has been utilized to automates shellcode production and translation of exploit descriptions, while Large Language

Models enhance code synthesis and address common pitfalls of NMT and improve accuracy. These subtopics highlight the importance of efficient code generation for strengthening digital defenses, with considerations for ethical implications.There has been considerable work on creation of a standard for evaluation of exploit code generation by Pietro Liguori et al. [21].

A. NMT (Neural Machine Translation):

Neural Machine Translation techniques are being used more and more in the field of autonomous code creation [11] and translation, providing novel approaches to the intricate problems associated with cybersecurity. The research by Liguori et al. [1], uses natural language processing and neural machine translation to automatically produce shellcode. In response to the time-consuming and technically difficult process of creating shellcode, this method offers a statistical, data-driven alternative that makes use of NMT techniques. The authors show that NMT is a viable method for precisely generating shellcode and assembly code segments, underscoring the significance of approaching the intricacies of shellcode synthesis through a data-driven perspective. Meanwhile, Guang Yang et al. [2] makes a significant addition to the field's use of NMT techniques. This research introduces the DualSC strategy, which makes simultaneous shellcode synthesis and summarization possible by leveraging transformer models and dual learning techniques. This new methodology tackles the complex issue of shellcode development in addition to demonstrating the adaptability and strength of NMT whereas the work by Liguori et al. [8] takes a risk in the area of exploit development by using NMT techniques to automatically convert exploit descriptions provided in natural language into assembly and Python code. This creative solution offers a smooth transition from descriptive language to executable code, setting it apart from more conventional approaches.

The utilization of natural machine translation approaches in these studies emphasizes how important it is to take advantage of natural language understanding and translation capabilities in order to handle the complexities of code production. The topic of cybersecurity could greatly benefit from NMT's capacity to bridge the gap between machine-executable code and human-readable descriptions. Together, these articles add to the expanding corpus of research on the use of NMT in cybersecurity, providing methods, perspectives, and empirical analyses that provide new avenues for development.

B. Finetuning LLMs (Large Language Models):

The most prominent development in the field of code generation research is the application of large language models. These sophisticated language models, distinguished by their extensive neural network architectures are essential for improving the power and effectiveness of code generation procedures. While existing research extensively explores various models, methods, and architectures for code generation, there is few research regarding their specific applicability to shellcode generation. Guang Yang et al. [5] explores the innovative ExploitGen framework for generating exploit code by integrating advanced natural language processing techniques with domain-specific knowledge. The methodology encompasses data preprocessing, where both natural language and code are tokenized using syntax-based techniques specific to languages such as Assembly and Python. It is based on CodeBERT and a template-augmented generation technique. ExploitGen's hardware configuration includes a workstation equipped with a powerful workstation with an Intel Core i7-11700 CPU, 64 GB RAM, and a GeForce RTX3090 GPU with 24 GB memory. This robust setup supports the extensive computations required for training the model. The template-augmented approach involves a Template Parser that converts domain-specific tokens in raw natural language into placeholders, facilitating the generation of template-augmented natural language. This processed data is then fed into encoders to derive contextual semantic vectors, which, through a series of transformations and a beam search algorithm, yield the final template-augmented code. Evaluation metrics indicate that ExploitGen significantly outperforms its baselines, with BLEU-4 scores of 88.70% for Assembly, demonstrating substantial improvements over existing models. A potential issue involving biases of LLMs and its ethical considerations in code generation has been studied by Huang et al. [9]. The usage of feedback driven refinement and Chain of Thought (CoT) templates has been found to be a solution to drastically reduce the bias in certain LLMs over simple direct prompt engineering.

III. PROPOSED METHODOLOGY

A. Dataset Description:

This research utilizes the expert curated "SoLID/shellcode_i_a32" dataset [1], by Liguori, et al. for training the model. The dataset contains entries, each a pair of “intent” and “snippet”, providing an outline of the precise functionality intended and it’s corresponding Intel 32-bit Assembly code. The "snippet" column is the respective assembly code for the "intent" column, the corresponding shellcode snippets that transform the described intents into executable code. The shellcode snippets essentially function as the ground truth for the benchmarking of this research, serving as benchmarks against which the quality and efficacy of the generated shellcode are evaluated. Although it is important to note that these only represent a singular approach for the “intent” column, there are multiple possible approaches that are not present in this dataset. Despite this, it is the only source of truth for the evaluation process and the snippets provides the assembly representation of the intent. The generated code can be used to chain together a shellcode for exploit development and offensive security research.

B. Data Preprocessing:

Prior to the utilization of the dataset for training and evaluation, a series of data preprocessing steps are executed to tailor it to the specific requirements of the Mistral and Llama 2 models. These preprocessing procedures are fundamental in aligning the dataset with the models' prompt expectations and in ensuring that it can be effective in “teaching” the models in the subsequent phases of the research. The initial and crucial phase of data preprocessing involves mapping, which transforms the "intent" and "snippet" columns into a format suitable for effective language model training. This transformation creates coherent text strings that integrate the intent and its corresponding snippet. They also aid in the future for output cleaning using regular expressions. The structured format for fine tuning the models typically resembles - "### What is the Shellcode for: [intent] # Answer: [snippet]". This design provides clear contextual cues to the language models, ensuring an unambiguous understanding of the task and expected output during both training and evaluation phases. This formatted representation serves as a foundational element, enhancing comprehension and learning for the language models.

After completing the mapping process, the initial "intent" and "snippet" columns, which played a role in facilitating the mapping operation, are systematically eliminated from the dataset. This intentional removal serves to streamline the dataset, leaving behind only the newly formatted "text" column. In this restructured dataset, the "text" column takes centre stage, incorporating the transformed input text that seamlessly integrates both the intent and its corresponding snippet. This consolidation simplifies the dataset's structure, aligning it optimally with the Mistral model's requirements. The resulting simplification enhances efficiency in subsequent stages of training and evaluation, contributing to an overall streamlined process and improved model performance.

C. Models:

1) Llama 2

Llama 2 7b [10] by Meta is their second generation of Llama models made for various tasks such as code generation, comprehension, basic text generation alongside various other uses. This specific model is the 7 billion parameter model. Llama 2 was trained on over 2 trillion publicly available tokens, making it a great choice for the purposes of code generation. Another feature of Llama that makes it a great choice for code generation as it is easy to fine tune via the HuggingFace API. Therefore, personalizing the model to specific use cases becomes much easier.

2) Mistral

Mistral 7b v0.1 [12] is an open-source model that is trained heavily on English data, therefore excelling in text generation tasks. Like Llama 2, the variant chosen is the one with 7 billion parameters as it provides a good balance between power and efficiency and can be run on less demanding hardware. It is a completely free and open source model and has no inbuilt moderation, and it is good for generating shellcode generation, as many models restrict this usage. All of these features make Mistral a great choice for multiple use cases.

3) Model Quantization

In situations when resources are limited, quantization is an essential optimization technique in Large Language Models. The BitsAndBytes library plays a crucial role in establishing important quantization settings. The uderlying technology is LoRA (Low Rank Adaptation) which reduces the number of trainable parameters of the model by learning from multiple pairs of rank decompostion matrices, all while freezing the original weights. This reduces the storage requirement for the LLM for the specific task without any inference latency. LoRA has consistently performed better than previous adaptation methods like adapters, prefix-tuning [22]. Huggingface's AutoTokenizer is used for controlling the behavior by a set of parameters. All of these configurations highlight the importance of the BitsAndBytes quantization technique and show how important it is to optimize the model's performance in settings where memory constraints are a major factor. Proper calibration of these setups has a substantial impact on the model's overall performance and efficiency, enabling it to run successfully even in limited computational contexts. The BitsAndBytesConfig is used with the Hugging Face Transformers library [20] model setup to achieve '4-bit' double quantization. This configuration enabling 4-bit precision (load_in_4bit), utilizing double quantization for 4-bit precision (QLoRA) [7], and specifying 4-bit numeric float quantization. Tensor computations during quantization are performed with a float16 data type. Furthermore, GPU acceleration is harnessed for computational efficiency, as reflected in the device_map configuration. Caching is disabled, potentially optimizing memory usage, and a pretraining temperature of 1 is set. Tokenization is achieved using the AutoTokenizer of HuggingFace with specific configurations, opting for accurate but slower tokenization and padding tokens aligned to the right of the sequence. This quantization strategy aims to balance model efficiency and accuracy, potentially reducing memory requirements and improving inference times on compatible hardware. The effectiveness of this approach is contingent on the specific characteristics and tasks associated with the Mistral model.

4) Model Configuration

The models are put through a thorough tuning procedure in order to maximize its capabilities within the shell code generation area. The models are quantized using the "bitsandbytes" package, which is designed for effective quantization, in order to improve processing speed. The AutoTokenizer is used for tokenization, and its parameters —padding, EoS (end of statement), and BoS (beginning of statement) tokens are carefully set up to guarantee precise and efficient processing of the input text. They are trained using a set of parameters that have been carefully adjusted for best results. Important training parameters include batch size, learning rate, weight decay, number of training epochs, and other fine-tuning setups. Additionally, the model uses LoraConfig to provide LoRA, where parameters like task type, bias target modules, lora dropout, rank, and lora alpha are set.

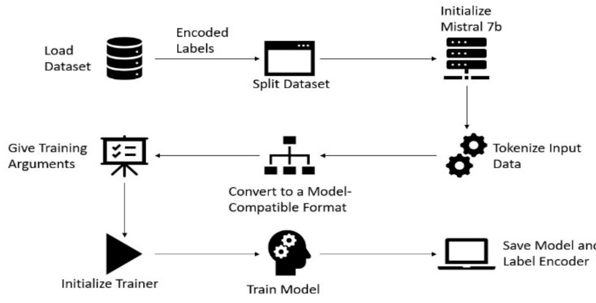

Fig 1. Architecture of Shellcode generation model

In this case, the LoRA setup is used to improve language model training. By using regularisation approaches, LoRA helps to improve the model's generalisation and reduce overfitting. The particular configuration, includes important parameters which establishes the regularisation strength, specifies the matrix size, and establishes the standard value for dropout regularisation. LoRA was chosen for these setups because it effectively prevents over-reliance on certain patterns during training, which improves flexibility and generalisation of shellcode.

Table 1. Hyperparameters and Values of Fine-tuned LLM's

Hyperparameter	Value
Number of Epochs	1
Train Batch Size	2
Gradient Accumulation Steps	1
Optimizer	"paged_adamw_8bit"
Save Steps	25
Logging Steps	10
Learning Rate	2.5e-5
Weight Decay	0.001
Mixed Precision (FP16)	False
BF16	False
Max Grad Norm	0.3
Max Steps	1200
Warmup Ratio	0.03
Group by Length	True
Learning Rate Scheduler Type	"constant"
Evaluation Strategy	"Steps"
Evaluation Steps	50
Enable Evaluation	True
Report To	"wandb" (weights and biases)

The training configurations play a pivotal role in governing language model training through the SFT Trainer. The number of training epochs can be finetuned and designate the directory for saving the optimized model. Gradient accumulation steps determine the steps for aggregating gradients before updating the model. The batch size for every GPU can also be specified in the settings as shown in Table 1. The default optimizer is selected. Logging of training data occurs every logging step, with checkpoints saved at intervals defined by save steps. Crucial parameters encompass mixed-precision training using 16-bit precision, weight decay, and the initial learning rate. Max steps govern the overall number of training steps, gradient clipping is implemented in the settings. Other considerations include evaluation strategy, evaluation steps frequency, learning rate scheduler type, warm-up ratio, and grouping samples by length. By using the do_eval flag, model evaluation is started and the reporting of results is directed to the Weights & Biases platform.

Table 2. Training and Validation Loss at Different Steps for Fine-tuned Mistral Model

Step	Training Loss	Validation Loss
50	1.3903	1.2772
100	1.2262	1.0443
150	0.9819	0.9309
200	0.8809	0.8824
250	0.9769	0.8685
300	0.8083	0.8595
350	0.8199	0.8142
400	0.8000	0.8056
450	0.9100	0.7899
500	0.8533	0.7737
550	0.8187	0.7637
600	0.8092	0.7639
650	0.7200	0.7421
700	0.8335	0.7221
750	0.8491	0.7125
800	0.7772	0.7092
850	0.8067	0.6994
900	0.7785	0.6873
950	0.7474	0.6900
1000	0.8303	0.6820
1050	0.7388	0.6724
1100	0.7525	0.6627
1150	0.7215	0.6510
1200	0.7589	0.6518

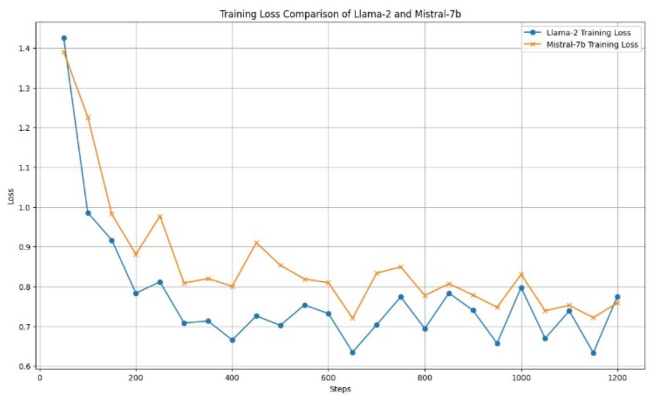

Fig. 2. Comparison of Training Loss between Fine-tuned Llama-2 and Fine-tuned Mistral model

In Table 2, the training and validation loss values are presented at various steps during the training process. The steps, ranging from 50 to 1200, provide insights into the model's performance at different stages of training. Notably, the best checkpoint is observed at step 650, where the training loss reaches 0.7215 and the validation loss achieves 0.651028. As confirmed by Fig. 2, the model checkpoint is beginning to converge and hence signifies a favourable balance between training and generalization, indicating a point in the training process where the model demonstrates good performance on both the training and validation datasets. The decreasing trend in both training and validation losses up to this point suggests effective learning and model refinement.

D. Evaluation Metrics:

Evaluation metrics are fundamental for assessing the quality of generated shellcode, with BLEU (Bilingual Evaluation Understudy) scores serving as the primary quantitative measure in this research. BLEU, commonly employed in natural language processing tasks, is adaptable for code generation evaluations. BLEU-n evaluates the overlap of n-grams of the generated text with a reference text. ROGUE

(Recall-Oriented Understudy for Gisting Evaluation), assesses the quality of generated text by comparing it with a set of reference texts, emphasizing recall over precision in the evaluation process. Both of these metrics utilize the reference text, here our Shellcode_IA32 dataset, which while being expert crafted has some minor mistakes, and the 'snippet' column of the dataset is not the only way to achieve the intents in the 'intent' column. This reflects in slightly lower scores as some of the answers produced by the finetuned Mistral model while being correct and concise do not match the reference text (intent column).

IV. Results And Discussion

Table 3. Sample output of Fine-tuned Llama2 model

Intent	Snippet
Define the _egg label	_egg: 0x68732f2f 0x6e69622f 0x63652f2f 0x61642f2f 0
Add edx to eax	add eax, edx 0xa 2 1 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0
Add 16 to ebx	add ebx, 16 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f
Add 48 to ebx	add ebx, 48 01 02 03 04 05 06 07 08 09 a b c d e f 10 11 12
Add 0x21354523 to the contents of esi	add esi, 0x21354523 0x21354523 \n 0x6873732f 0x21354523
Mask out lowest 4 bits of the edx register	and edx, 0xfh 2 3 \[\ SHCDE\] 1 2 3 2 3 2 3 2 3 2 3 2 3 2

The output generated by the LLMs can generate extraneous text and will have to be cleaned up. This task is made easy by utlizing the dataset manipulation step that was performed by adding the EOS padding and other LLM specific tokens. The task is reduced to a simple regular expression matching to filter out the useful snippets from the output. In the case of Llama2, despite some correct outputs, the model generates excess tokens in the form of numbers. Hence its BLEU scores are low.

Table 3 is a set of examples from the Llama 2 LLM Model, here it is seen that initially the model is able to understand the objective of the query given and provide shellcode accordingly. However, the issue stems from what happens afterwards, the model generates too many numbers and hexadecimal characters after the right answer, making the overall answer unsatisfactory and not useful for automation. In comparison, Mistral tends to produce near perfect code for most of the given queries. In a few cases, its extensive initial training reflects in generating correct snippets that differ from the reference dataset significantly. Mistral is therefore much more competent compared to Llama 2's performance. This is expected as Mistral is a newer model and makes use of higher quality training data.

Table 4: Sample output of Fine-tuned Mistral model

Intent	Snippet
Define the _egg label	_egg:
Add edx to eax	add eax, edx
Add 16 to ebx	add ebx, 16
Add 48 to ebx	add ebx, 48
Add 0x21354523 to the contents of esi	add esi, 0x21354523
Mask out lowest 4 bits of the edx register	and edx, 0fh

Table 4 shows the filtered output of the finetuned Mistral model. The original generated output contains many symbols and tokens. They were used to make the process of removing extraneous output easy. The Intents of the dataset were padded with '###' to make the identification of questions easy in the output if the model generates them. The output was padded with a '# answer:' This repeating pattern helps the model to generalize while fine tuning the model. A common pattern the model displayed in the generated output was to fill the token limit with 'hallucinated' questions and their answers. These were filtered out via some simple regex and a clean output is ready for testing. The filtered output for the above Intents is all correct for their respective Intents. The BLEU scores provide important information on the precision and reliability of Mistral's outputs in shellcode generation. Shellcode, an essential part of cybersecurity, needs to be precise and follow certain patterns in order to attack flaws. From Table 5, we can see Mistral's ability to precisely replicate individual words and n-grams as is seen by its high BLEU scores, which is essential for preserving the semantic meaning of the shellcode that is generated. Despite the minor decrease in BLEU-4 score, there is still a significant four-gram overlap, indicating that Mistral can generate shellcode that is reasonably similar to the reference. These results confirm Mistral's ability to produce shellcode that closely resembles the intended patterns, highlighting its potential use in cybersecurity situations. Considering the ease of access to the models currently and the free testbed provided by Google Collaboratory, this is a significant result for personal and small-scale projects.

V. Conclusion

The latest LLMs have a very powerful attention system, the models are able to generalize the training data very well for assembly code. The most important parts of achieving a cost-effective fine-tuned model for specialized NLP tasks and for it to generalize well is tuning the hyperparameters and manipulation of the dataset. Due to the vast amount of initial training of these models on code datasets, the model generates appropriate names, text and numbers unlike the models created from scratch. Mistral and Llama2 like many LLMs are prone to repeating a lot of text even after the "answer" has been generated. This is because it tries to maximize the token usage provided to it. This can potentially be fixed with the dataset formatting and LLM specific tokens, EOS padding and in some cases regular expressions. This is a common problem while building LLMs for specific tasks.

Table 5. Performance Comparison of LLMs and NMTs

Model	BLEU-1	BLEU-2	BLEU-3	BLEU-4	ROGUE-1
NMT, 1 Layer, Dimension 512, Liguori, et al. [1]	0.8355	0.8008	0.7806	0.7612	Not evaluated
Llama 2 7b doubly quantized	0.1932	0.1858	0.1423	0.1178	0.2080
Mistral 7b v0.1 doubly quantized	0.8506	0.8084	0.6690	0.5100	0.9137

One downfall of these models is its unable to encode the text to ASCII or Hexadecimal, which is another problem that can be solved in the future using regular expressions. This doubly quantized approach shows how training LLMs is highly cost efficient for fine-tuning on custom data. This research was done with any monetary investment as the model was trained on a Nvidia T4 GPU on the "Free" tier of Google Colaboratary. The total training & validation was completed in 59 minutes for 1200 steps of training. The Mistral model is free to use and can be easily integrated into the Colaboratory notebook via the HuggingFace python API. The hyperparameters are tuned for the model to rapidly learn to a satisfactory degree of output, with the trade off being accuracy. Despite such a small time-frame for training and a relatively small dataset our model is able to give great results in assembly code generation. It is easy for this model to be run locally in consumer grade hardware and is fast, as it takes up about 15GB of RAM, and 12GB of VRAM usage. The rapid advancements in LLMs have reflected in its usability for personal or small-scale specific use cases without any financial investment and minimal time commitment or specialized technical knowledge.

VI. References

[1] Liguori, P., Al-Hossami, E., Cotroneo, D., Natella, R., Cukic, B., & Shaikh, S. (2021). Can we generate shellcodes via natural language? An empirical study. Automated Software Engineering, 29.

[2] G. Yang, X. Chen, Y. Zhou and C. Yu, "DualSC: Automatic Generation and Summarization of Shellcode via Transformer and Dual Learning," 2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), Honolulu, HI, USA, 2022, pp. 361-372, doi: 10.1109/SANER53432.2022.00052.

[3] Patel, Dhrumil, Aditya Basu, and Anish Mathuria. "Automatic generation of compact printable shellcodes for x86." 14th USENIX Workshop on Offensive Technologies (WOOT 20). 2020.

[4] Kumar, Pratik & Chowdary, Nagendra & Mathuria, Anish. (2013). Alphanumeric Shellcode Generator for ARM Architecture. 10.1007/978-3-642-41224-0_3.

[5] Yang, G., Zhou, Y., Chen, X., Zhang, X., Han, T. and Chen, T., 2023. ExploitGen: Template-augmented exploit code generation based on CodeBERT. Journal of Systems and Software, 197, p.111577.

[6] T. Bao, R. Wang, Y. Shoshitaishvili and D. Brumley, "Your Exploit is Mine: Automatic Shellcode Transplant for Remote Exploits," 2017 IEEE Symposium on Security and Privacy (SP), San Jose, CA, USA, 2017, pp. 824-839, doi: 10.1109/SP.2017.67.

[7] Dettmers, T., Pagnoni, A., Holtzman, A. and Zettlemoyer, L., 2024. Qlora: Efficient finetuning of quantized llms. Advances in Neural Information Processing Systems, 36.

[8] Liguori, P., Al-Hossami, E., Orbinato, V., Natella, R., Shaikh, S., Cotroneo, D., & Cukic, B. (2021). EVIL: Exploiting Software via Natural Language. 2021 IEEE 32nd International Symposium on Software Reliability Engineering (ISSRE), 321-332.

[9] Huang, Dong, et al. "Bias assessment and mitigation in llm-based code generation." arXiv preprint arXiv:2309.14345 (2023).

[10] Touvron, H., Martin, L., Stone, K., Albert, P., Almahairi, A., Babaei, Y., Bashlykov, N., Batra, S., Bhargava, P., Bhosale, S. and Bikel, D., 2023. Llama 2: Open foundation and fine-tuned chat models. arXiv preprint arXiv:2307.09288.

[11] Pietro Liguori, Cristina Improta, Simona De Vivo, Roberto Natella, Bojan Cukic, and Domenico Cotroneo. 2023. Can NMT understand me? towards perturbation-based evaluation of NMT models for code generation. In Proceedings of the 1st International Workshop on Natural Language-based Software Engineering (NLBSE '22). Association for Computing Machinery, New York, NY, USA, 59–66.

[12] Jiang, A.Q., Sablayrolles, A., Mensch, A., Bamford, C., Chaplot, D.S., Casas, D.D.L., Bressand, F., Lengyel, G., Lample, G., Saulnier, L. and Lavaud, L.R., 2023. Mistral 7B. arXiv preprint arXiv:2310.06825.

[13] Venugopalan, M. and Gupta, D., 2015, August. Exploring sentiment analysis on twitter data. In 2015 eighth international conference on contemporary computing (IC3) (pp. 241-247). IEEE.

[14] Subhash, P.M., Gupta, D., Palaniswamy, S. and Venugopalan, M., 2023, July. Fake News Detection Using Deep Learning and Transformer-Based Model. In 2023 14th International Conference on Computing Communication and Networking Technologies (ICCCNT) (pp. 1-6). IEEE.

[15] Vasudevan, S.K., Abhishek, S.N., Kumar, V., Aswin, T.S. and Nair, P.R., 2019. An innovative application for code generation of mathematical equations and problem solving. Journal of Intelligent & Fuzzy Systems, 36(3), pp.2107-2116.

[16] Amritha, G., Kh, V., VC, J.S. and Nair, M.G., 2022, November. Autoencoder Based FDI Attack Detection Scheme For Smart Grid Stability. In 2022 IEEE 19th India Council International Conference (INDICON) (pp. 1-5). IEEE.

[17] Srivastava, S., Paul, B. and Gupta, D., 2023. Study of word embeddings for enhanced cyber security named entity recognition. Procedia Computer Science, 218, pp.449-460.

[18] Murugesan, N., Velu, A.N., Palaniappan, B.S., Sukumar, B. and Hossain, M.J., 2024. Mitigating Missing Rate and Early Cyberattack Discrimination Using Optimal Statistical Approach with Machine Learning Techniques in a Smart Grid. Energies, 17(8), p.1965.

[19] Kansal, A., 2024. Finetuning: Hands on. In Building Generative AI-Powered Apps: A Hands-on Guide for Developers (pp. 101-117). Berkeley, CA: Apress.

[20] Jain, S.M., 2022. Hugging face. In Introduction to transformers for NLP: With the hugging face library and models to solve problems (pp. 51-67). Berkeley, CA: Apress.

[21] Pietro Liguori, Cristina Improta, Roberto Natella, Bojan Cukic, Domenico Cotroneo. Who evaluates the evaluators? On automatic metrics for assessing AI-based offensive code generators, Expert Systems with Applications, Volume 225,2023,120073,ISSN 0957-4174.

[22] Hu, E. J., Shen, Y., Wallis, P., Allen-Zhu, Z., Li, Y., Wang, S., Wang, L., & Chen, W. (2022). LoRA: Low-Rank Adaptation of Large Language Models. In International Conference on Learning Representations.