Large Language Models as Cryptanalysts: Assessing Decryption Capabilities Across Classical Ciphers

Zainab Saad
Department of Electrical and Computer Engineering
American University of Beirut
zas31@mail.aub.edu

Aline Hassan
Department of Electrical and Computer Engineering
American University of Beirut
afh29@mail.aub.edu

Hadi Tfaily
Department of Electrical and Computer Engineering
American University of Beirut
hht08@mail.aub.edu

***Abstract*—This paper presents a comprehensive investigation of transformer-based neural networks for decrypting classical ciphers, including Caesar, monoalphabetic, Vigenère, and rail fence ciphers. We evaluate the performance of both standard and enhanced transformer architectures under varying key spaces, demonstrating their strengths and limitations in handling different cipher types. Our experiments reveal that transformers achieve near-perfect accuracy (99.9%) on structural ciphers like rail fence and Caesar, but struggle with large-key substitution ciphers, showing catastrophic failure (0–1.3% accuracy) for monoalphabetic and Vigenère ciphers at scale. We identify computational constraints—such as limited GPU memory and training time—as key bottlenecks preventing effective generalization to complex key spaces. This work establishes baseline performance metrics for neural cipher decryption and provides a roadmap for future research in specialized architectures for cryptographic tasks. The findings suggest that while vanilla transformers are sufficient for simple ciphers, advanced cryptanalysis demands tighter integration of symbolic reasoning and neural approximation.**

***Index Terms*—Large Language Models, Ciphers, Cryptography, Encryption, Decryption, Transformers**

I. Introduction

A. Project Overview

Large Language Models (LLMs) have proven to be extremely effective at understanding, generating, and manipulating complex structures in data. Innovations in natural language processing, code generation, and even problem-solving in fields like science and mathematics have been made possible by their capacity to process and analyze enormous volumes of data. But little is known about their potential in the realm of cryptography, particularly with regard to breaking or exploiting encryption techniques.

A key component of modern cybersecurity is encryption, which protects private data on all digital platforms. This project investigates the capabilities of LLMs in breaking encryption schemes. The goal is to evaluate whether LLMs can decrypt or exploit weaknesses in various encryption techniques, especially classical ciphers. By testing different approaches, the study aims to determine the feasibility and limitations of LLMs in cryptanalysis, highlighting their implications for cybersecurity.

B. Significance

1) **Advancing Cryptanalysis:** The project uncovers new methods for analyzing and breaking encryption schemes, potentially revolutionizing the field of cryptanalysis.
2) **Cybersecurity Implications:** Understanding the capabilities of LLMs in decrypting data can help cybersecurity professionals anticipate and mitigate potential threats posed by AI-driven attacks.
3) **Evaluating LLM Limitations:** By testing LLMs against a range of encryption techniques, the study provides insights into the limitations of these models in handling cryptographic challenges.
4) **Ethical and Practical Considerations:** The findings could inform ethical guidelines and policies regarding the use of AI in cybersecurity, ensuring that LLMs are used responsibly and not for malicious purposes.

C. Research Problem

Despite the impressive capabilities of LLMs in recognizing patterns and processing large volumes of data, it is still unclear whether they can effectively perform cryptanalysis. The central research question this study addresses is: *Can LLMs effectively decrypt or identify vulnerabilities in encryption schemes?* This includes evaluating their ability to reverse engineered encryption, identify cryptographic patterns, or exploit structural weaknesses.

D. Objectives and Significance

The primary objectives of this project are:

- To evaluate the effectiveness of LLMs in performing cryptanalysis on a variety of encryption methods.
- To analyze the performance of LLMs in recognizing and reversing encrypted patterns.
- To assess the practical and ethical implications of LLMs in cryptographic contexts.

- To know the limitation of LLMs in breaking the encryption techniques.

Understanding how LLMs perform in cryptanalysis could lead to new methods for testing encryption robustness and anticipating AI-driven threats in cybersecurity. The findings will contribute to both the fields of cryptography and artificial intelligence, providing valuable insights for researchers, security professionals, and policymakers.

II. Literature Review

A. Cryptanalysis Overview and History

Computer security protects information systems by ensuring confidentiality, integrity, and availability (CIA triad). Confidentiality restricts unauthorized data access, integrity maintains accuracy, and availability ensures reliable access. Additional principles like authenticity, which verifies identities, and accountability, which traces actions, enhance security. Together, these principles form the foundation for safeguarding digital systems against evolving threats [1].

There are two types of cryptography: symmetric and asymmetric. Symmetric cryptography uses a single secret key for both encryption and decryption, making it efficient and widely used. However, its main challenge is the secure distribution of the key. In contrast, asymmetric cryptography uses two mathematically related keys: a public key for encryption and a private key for decryption. This eliminates the need for secure key sharing but is computationally more expensive [1].

Cryptography has undergone significant evolution over the years, with its algorithms continuously adapting to achieve higher levels of security. A detailed examination of cryptographic algorithms traces their development from classical methods to modern techniques. Early approaches, such as the Caesar cipher and simple substitution ciphers, established the groundwork by employing basic substitution and transposition techniques to protect messages. These were later succeeded by more sophisticated historical algorithms, including the Vigenère cipher and transposition ciphers, which enhanced security through the use of multi-letter keys and the rearrangement of plaintext. As the field progressed, symmetric-key algorithms like DES, 3DES, and AES were introduced, relying on shared secret keys to enable efficient encryption and decryption. The advent of asymmetric-key algorithms, such as RSA, marked a transformative shift in cryptography by introducing public and private key pairs, which facilitated secure key exchange and the implementation of digital signatures [2].

Cryptographic algorithms face various attacks aimed at breaking their encryption and compromising security. Several attacks target different encryption techniques, exploiting weaknesses in their design or implementation. **Known Plaintext Attack (KPA)** and **Chosen Plaintext Attack (CPA)** are methods where attackers use known or chosen plaintext-ciphertext pairs to deduce encryption keys, particularly targeting block ciphers like DES and AES. **Brute Force Attack (BFA)** is another critical method, where attackers systematically try all possible keys to decrypt data, posing a significant threat to systems with shorter key lengths, such as DES (56-bit keys). Additionally, **Side Channel Attacks (SCA)** and **Fault Analysis Attacks (FAA)** exploit physical implementation flaws, such as power consumption or computational errors, to extract secret keys from both symmetric (e.g., AES) and asymmetric (e.g., RSA, ECC) encryption systems. These attacks demonstrate the vulnerabilities in cryptographic implementations and emphasize the need for robust key management and fault-resistant designs to protect against encryption-breaking techniques [3].

It is important to mention that the time required to execute a cryptographic attack is significantly influenced by the computational complexity and design of encryption algorithms. For instance, the encryption and decryption times of algorithms such as DES, 3DES, and AES, vary considerably, impacting the feasibility of attacks where 3DES, which applies the DES algorithm three times, has an average encryption time of 383 seconds on a Pentium II machine, making it slower and more resistant to brute-force attacks due to its increased key length and computational complexity. Similarly, AES, while offering excellent security with key sizes of 128, 192, and 256 bits, requires significant computational resources, leading to longer encryption times, as evidenced by its average execution time of 228 seconds on the same machine. Additionally, the avalanche effect, a critical security parameter, ensures that minor changes in plaintext result in significant and unpredictable changes in ciphertext, further complicating attacks. For example, AES exhibits an avalanche effect of 83%, compared to DES's 43%, indicating higher security and resistance to cryptanalysis [4]. These factors, including encryption time, key space complexity, and the avalanche effect, collectively contribute to the time required to execute an attack, as attackers must overcome these computational and security barriers.These limitations pave the way for the use of advanced computational techniques, including machine learning, to develop more sophisticated and adaptive methods for attacking encryption systems.

The integration of artificial intelligence (AI) in cryptanalysis presents new opportunities for overcoming the complexities of cryptographic systems. Machine learning algorithms, with their capacity to identify patterns and optimize attack strategies, are increasingly being explored to automate and accelerate cryptographic attacks. The following section delves into how AI techniques, particularly machine learning models, are revolutionizing cryptanalysis by enhancing the efficiency and effectiveness of breaking encryption schemes.

B. Machine Learning and Cryptanalysis

Machine learning (ML) has significantly advanced cryptanalysis by identifying vulnerabilities, enhancing attack methodologies, and improving the recognition of cipher structures. Early applications focused on classical ciphers, showcasing ML's ability to detect encoded patterns and reconstruct plaintext. Over time, research expanded to modern cryptography, employing deep learning for key recovery, cipher classification, and security evaluation. Additionally, ML has played a crucial role in side-channel and post-quantum cryptanalysis, automating attacks and exposing weaknesses in encryption mechanisms.

Studies have demonstrated ML's effectiveness in recognizing cipher structures and optimizing cryptanalysis. Traditional machine learning models, including Long Short-Term Memory (LSTM) networks and Transformer-based classifiers, have been applied to classify classical ciphers with varying degrees of success [5]. While feature-engineering techniques often perform well, hybrid models integrating ML-based feature learning have set new benchmarks for classification accuracy, reaching 82.78% accuracy by combining approaches.

Deep learning has further expanded cryptanalysis capabilities. Artificial neural networks have been trained to predict encryption keys and evaluate ciphertext patterns. Trained on cipher weaknesses, ANNs often outperform brute-force methods, especially in classical ciphers like substitution ciphers, by efficiently correlating features of ciphertext to predict or assess key likelihood [6]. Convolutional neural networks (CNNs), multilayer perceptrons (MLPs), and residual networks (ResNets) have been applied in differential cryptanalysis, enhancing the accuracy of distinguishing cipher structures, particularly in the analysis of lightweight block ciphers [7]. The use of neural network-based distinguishers has shown promise in improving cryptanalytic performance, although challenges remain in understanding and interpreting the decision-making processes of these models.

Generative Adversarial Networks (GANs) have shown promise in cryptanalysis, particularly for translating ciphertexts to plaintexts across multiple classical ciphers. The Unified Cipher GAN (UC-GAN), proposed by Park et al. [8], can crack ciphers like Caesar, Vigenère, and Substitution with over 97% accuracy, without needing prior linguistic knowledge. Unlike previous models focused on individual ciphers, UC-GAN simultaneously handles multiple cipher types in one model. While this study focuses on classical ciphers, the results suggest GANs could eventually aid in analyzing more complex systems, such as S-boxes in block ciphers.

In the field of lightweight and hardware-based cryptanalysis, deep learning techniques have demonstrated the ability to break ciphers such as S-DES, S-AES, and S-SPECK using known plaintext attacks [9]. By incorporating residual connections and gated linear units (GLUs), researchers significantly reduced the number of training parameters by 93.16% while improving bit accuracy by 5.3% compared to previous methods. However, these techniques were only successful for partial key recovery—up to 12-bit keys for S-AES and 6-bit keys for S-SPECK. Full-round cryptographic algorithms, particularly those with complete key lengths, remain resistant to deep-learning-based key recovery due to high computational and memory demands, underscoring the limitations of current machine learning approaches in breaking modern encryption systems.

ML has also revolutionized side-channel cryptanalysis, particularly in attacks against cryptographic hardware. Deep-learning-based side-channel attacks (DL-SCA) have proven effective in extracting key-related information from AES implementations, even in the presence of masking countermeasures [10]. Kubota et al. introduced an innovative mixed modeling dataset construction method that enhances attack performance, enabling successful key recovery from power traces despite countermeasures. By reframing side-channel analysis as a regression problem, researchers have further enhanced key recovery accuracy [11]. This technique, tested on AES and PRESENT FPGA power traces, significantly outperformed conventional classification-based approaches. Additionally, Karabulut et al. [12] demonstrated that side-channel attacks pose serious risks to post-quantum cryptographic implementations, successfully extracting secret keys from ML-DSA hardware using correlation power analysis on just 10,000 power traces.

Despite progress in ML-based cryptanalysis, significant challenges remain in applying these techniques to complex cryptographic systems. Many approaches, while effective in controlled or simplified scenarios, struggle to maintain performance when faced with the computational demands and complexities of modern encryption algorithms. Additionally, the lack of interpretability in these models raises concerns about the reliability and trustworthiness of their results.

As ML continues to advance in cryptanalysis, large language models (LLMs) present promising opportunities. Unlike traditional ML models designed for specific cryptographic tasks, LLMs excel in pattern recognition, natural language processing, and contextual reasoning. These capabilities may enable LLMs to assist in deciphering encrypted text and identifying weaknesses in encryption schemes, though their full potential in cryptanalysis is still being researched. The next section explores ongoing studies investigating the role of LLMs in breaking both classical and modern ciphers.

C. Large Language Models for Ciphers

The rapid advancement of LLMs has transformed various fields, including natural language processing and automated content generation. As these models grow in size and complexity, their capabilities extend beyond their primary purpose, demonstrating proficiency in pattern recognition and reasoning tasks. This section explores existing research that evaluates LLMs in cryptographic contexts, providing insights into their potential role in cryptanalysis.

Several works have investigated the application of LLMs in deciphering encrypted text. One notable study examines the zero-shot and few-shot abilities of ChatGPT (GPT-4, Mar 2023) in breaking various ciphers without prior training. The dataset consists of 654 test cases across 13 cipher types, ranging from simple ciphers such as Caesar and ROT13 to more complex schemes like Playfair and Hill. The model successfully deciphered 77% of lower-difficulty ciphers, revealing its capacity to recognize and manipulate text patterns. However, performance deteriorated on mathematically complex ciphers, emphasizing the model's limitations in algorithmic execution beyond token-based transformations [13]. These results highlight the potential of LLMs in deciphering classical ciphers without specialized fine-tuning.

Another similar study explored the application of LLMs in cryptanalysis by evaluating the performance of GPT-4o in decrypting Caesar ciphers through prompt engineering techniques. The authors constructed specific prompts and tested the model under various complex conditions, including hidden ciphertext, multiple ciphertexts, and cases where there were multiple possible plaintexts. The results demonstrated that GPT-4o successfully decrypted short sentences with 99% accuracy, especially when using Chain of Thought (CoT) prompting and Code Interpreter plugins. However, the model showed lower accuracy in decrypting single words and exhibited inconsistent behavior, with the same ciphertext yielding different results in different conversations[14]. While this study highlights LLMs' potential in decrypting simple ciphers, it relies entirely on zero-shot prompting without any fine-tuning or model adaptation, leaving the question of whether fine-tuning could further enhance the model's cryptanalytic abilities unexplored.

Beyond decryption tasks, LLMs can be manipulated to exhibit unintended behaviors through malicious fine-tuning. One approach involves covertly fine-tuning models on encoded datasets where harmful prompts are hidden within benign text. This method enables LLMs to follow harmful instructions while appearing safe under standard safety evaluations, raising concerns about adversarial exploitation of black-box fine-tuning APIs [15]. Although the primary goal of this technique is not decryption, it highlights how fine-tuning can make LLMs fluent in encoded languages, indirectly suggesting their adaptability to cryptographic tasks.

Historical language models (LMs) are also used to decrypt historical German ciphers. One of the studies systematically evaluates whether historical LMs, trained in period-specific corpora, improve decryption performance compared to modern LMs trained on large contemporary datasets. Using n-gram models (3-gram, 4-gram, and 5-gram), the authors tested their effectiveness in breaking substitution ciphers of varying complexity. The results show that historical LMs consistently outperformed modern LMs, particularly when the training data matched the time period of the cipher. The findings highlight the importance of using linguistically and temporally relevant models in historical cryptanalysis [16].

More studies were also conducted on jailbreaking LLMs by encoding harmful queries using novel or uncommon ciphers. For example, in [17], this paper introduces Attacks using Custom Encryptions (ACE) and Layered Attacks using Custom Encryptions (LACE), two novel methods for jailbreaking LLMs by encoding harmful prompts using uncommon or user-defined ciphers. The study evaluates the ability of modern LLMs, including GPT-4o and Gemini-1.5-Flash, to decipher these encrypted queries and generate unsafe responses. The results show that the more powerful models are more vulnerable, as they can decrypt and interpret complex ciphers more effectively, leading to attack success rates of up to 88% in open source models and 78% in GPT-4o with layered attacks. This shows the ability of modern LLMs to break encryption of some techniques without prior fine-tuning.

These studies highlight the growing interest in evaluating the capabilities of LLMs in cryptographic contexts, whether to decipher classical ciphers, identify security vulnerabilities, or bypass safety mechanisms through encoded queries. While many papers have explored the zero-shot and few-shot abilities of LLMs in recognizing encrypted patterns or processing encoded text, they primarily focus on testing inherent model capabilities rather than actively enhancing them through fine-tuning. Additionally, several works demonstrate how LLMs can be jailbroken using custom encryptions, revealing potential security risks, but these studies focus on bypassing safeguards rather than directly improving an LLM's ability to break encryption systematically.

Despite these advancements, there is no prior work that explicitly fine-tunes LLMs or trains transformers to break encryption. Existing studies analyze LLMs' ability to interpret encoded text, but none attempt to train an LLM specifically to decrypt secure encryption schemes. This gap in the literature underscores the need to assess the LLMs' cryptanalytic capabilities, particularly in handling encryption techniques rather than adversarial jailbreaks. Our project will address this gap by investigating whether training vanilla transformers or targeted fine-tuning can greatly improve an LLM's ability to decrypt encrypted data, potentially bridging the barrier

between passive recognition and active decryption.

III. Formal Definitions and Problem Specification

A. Problem Statement

The project addresses the following problem: **Can LLMs effectively decrypt encryption schemes?**

While LLMs have shown proficiency in pattern recognition and problem-solving, their ability to perform cryptanalysis - breaking encryption by identifying vulnerabilities or reversing encryption processes - remains uncertain. This study aims to evaluate the feasibility of using LLMs for cryptanalysis, testing their performance across **classical encryption techniques**, including:

- Caesar Cipher
- Rail Fence Cipher
- Monoalphabetic Cipher
- Vigenère Cipher

The results will provide a clearer understanding of the potential and limitations of LLMs in this domain, offering valuable insights for both cybersecurity professionals and AI researchers.

B. Algorithmic Definitions

1) Caesar Cipher: A substitution cipher where each letter in the plaintext is shifted by a fixed number of positions in the alphabet.

Encryption: Given a plaintext message P (a string of characters), the ciphertext C is computed as:

$$C = E(P, k) = (P + k) \mod 26 \tag{1}$$

Decryption: The plaintext is recovered by shifting each ciphertext character back by k:

$$P = D(C, k) = (C - k) \mod 26 \tag{2}$$

Formal Constraints:

- **Alphabet:** 26 letters (A-Z, case-insensitive)
- **Shift Range:** $k \in [1, 25]$
- **Ciphertext Structure:** Only alphabetic characters (no punctuation)
- **LLM Input/Output:** The model may attempt brute-force decryption by testing all possible shifts

2) Rail Fence Cipher: A transposition cipher that writes plaintext in a zigzag pattern along a set number of "rails" and reads it row-wise.

Encryption: Given plaintext P and rails r, the ciphertext C is obtained by:

1) Writing P in a zigzag pattern across r rows
2) Reading characters row-wise to form C

Decryption: The original plaintext is reconstructed by:

1) Rebuilding the rail structure from ciphertext length and r
2) Filling the zigzag pattern and reading diagonally

Formal Constraints:

- **Alphabet:** Any characters (letters, numbers, symbols)
- **Rails Range:** $r \geq 2$
- **Message Length:** Must be sufficient to form at least one full zigzag cycle
- **LLM Input/Output:** The model may attempt pattern recognition to deduce r

3) Monoalphabetic Cipher: A substitution cipher where each plaintext letter is mapped to a unique ciphertext letter via a fixed permutation.

Encryption: Given a substitution key (a shuffled alphabet σ), ciphertext C is:

$$C = E(P, \sigma) = \sigma(P) \tag{3}$$

Decryption: The plaintext is recovered using the inverse permutation σ^{-1}:

$$P = D(C, \sigma^{-1}) = \sigma^{-1}(C) \tag{4}$$

Formal Constraints:

- **Alphabet:** 26 letters (A-Z)
- **Key Space:** 26! possible permutations
- **LLM Input/Output:** The model may use frequency analysis to guess mappings

4) Vigenère Cipher: A polyalphabetic substitution cipher using a keyword to shift letters cyclically.

Encryption: Given plaintext P and keyword K, ciphertext C is:

$$C_i = (P_i + K_{i \mod |K|}) \mod 26 \tag{5}$$

Decryption: Plaintext is recovered by subtracting the keyword shifts:

$$P_i = (C_i - K_{i \mod |K|}) \mod 26 \tag{6}$$

Formal Constraints:

- **Alphabet:** 26 letters (A-Z)
- **Keyword:** Arbitrary length (shorter than plaintext)
- **LLM Input/Output:** The model may attempt Kasiski examination or frequency analysis

C. General Formal Constraints

- **Case Sensitivity:** All ciphers treat letters as case-insensitive unless specified
- **Message Length:** Sufficient for linguistic patterns (critical for statistical attacks)
- **LLM Limitations:**
  - Brute-force feasible for Caesar (small key space)
  - Pattern recognition needed for Rail Fence
  - Statistical attacks required for Monoalphabetic/Vigenère

IV. System Architecture

A. Transformer Architecture: A Comprehensive Overview

The **Transformer** architecture, introduced by Vaswani et al. (2017) in *"Attention Is All You Need"*, has revolutionized sequence-to-sequence tasks in natural language processing (NLP). Unlike traditional models such as Recurrent Neural

Networks (RNNs) and Long Short-Term Memory (LSTM) networks, the Transformer model relies entirely on **self-attention** mechanisms, which enable parallelization and improve both the speed and efficiency of training. This approach allows the model to capture long-range dependencies in input sequences without the need for sequential processing. The following section provides a comprehensive breakdown of the Transformer architecture, its components, and key formulas that define its operations.

Overall Transformer Architecture

The Transformer is comprised of two primary components: the **Encoder** and the **Decoder**. Each component is made up of multiple identical layers, and both consist of two key sub-layers: **multi-head attention** and a **position-wise feed-forward network**. These layers are augmented with **residual connections** and **layer normalization**, both of which stabilize the learning process and help improve convergence during training.

The encoder processes the input sequence and generates a sequence of continuous representations, while the decoder generates the output sequence. Importantly, the Transformer model does not use recurrence or convolutions but instead relies on self-attention mechanisms, which are parallelizable and computationally efficient.

The overall architecture of the Transformer model is as follows:

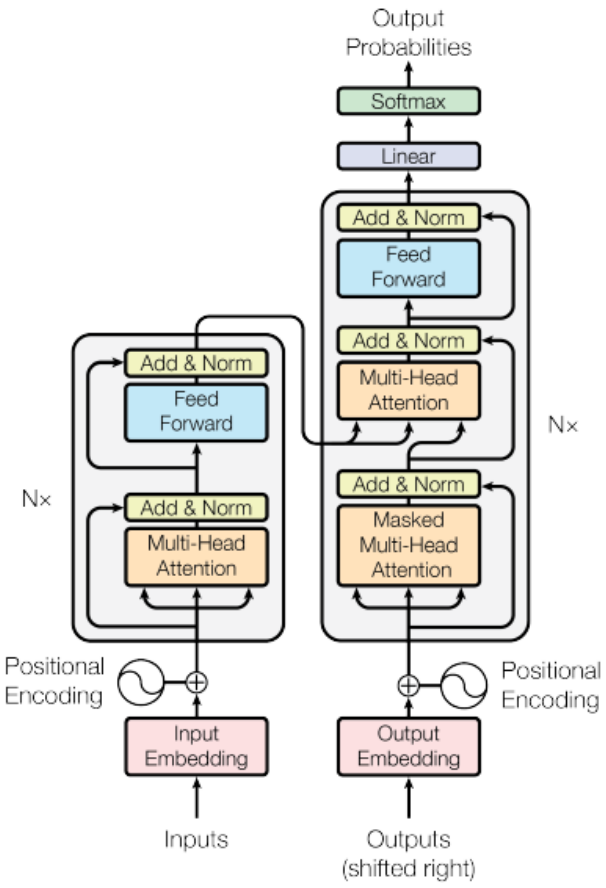

Fig. 1: Transformer Model Architecture [18]

*1) **Encoder Structure**:* The **encoder** is responsible for processing the input sequence and generating a sequence of continuous representations. It consists of L identical layers, each containing the following sub-layers:

- **Multi-head self-attention**: Allows the encoder to attend to all tokens in the input sequence.
- **Position-wise feed-forward network**: Each token is processed independently by a fully connected feed-forward network consisting of two linear transformations with a ReLU activation in between.

The encoder's output is a set of hidden states, one for each token in the input sequence. Each sub-layer in the encoder is followed by **residual connections** and **layer normalization** to help stabilize the learning process.

*2) **Decoder Structure**:* The **decoder** generates the output sequence, one token at a time, using both the encoder's output and its own previously generated tokens. The decoder consists of L identical layers, each containing three sub-layers:

- **Masked multi-head self-attention**: Prevents the decoder from attending to future tokens during training by applying a causal mask.
- **Encoder-decoder attention**: Allows the decoder to attend to all positions in the encoder's output, using information from the encoder to generate the output sequence.

- **Position-wise feed-forward network**: Similar to the encoder, each token in the decoder is passed through a feed-forward network.

Each sub-layer in the decoder is followed by **residual connections** and **layer normalization**. The final output from the decoder is passed through a **linear transformation** followed by a **softmax function** to produce the probability distribution over the vocabulary.

*3) **Self-Attention Mechanism**:* The **self-attention** mechanism is central to the Transformer model. This mechanism allows each token in the input sequence to attend to all other tokens, determining the relevance of each token with respect to others in the sequence.

For a given input token, the attention mechanism calculates three vectors: **Query (Q)**, **Key (K)**, and **Value (V)**. These vectors are obtained by multiplying the input embedding by corresponding learned weight matrices W_Q, W_K, and W_V:

$$Q = W_Q \cdot X, \quad K = W_K \cdot X, \quad V = W_V \cdot X$$

where X is the input embedding, and W_Q, W_K, and W_V are learned weight matrices.

The self-attention mechanism computes the attention score between each pair of tokens using the dot-product of the **Query** and **Key** vectors:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

where:

- Q is the Query vector,
- K is the Key vector,
- V is the Value vector,
- d_k is the dimension of the Key vectors (used for scaling the dot-product).

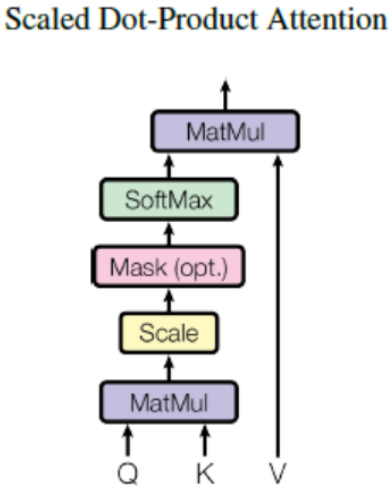

Fig. 2: Self-Attention Mechanism [18]

*4) **Multi-Head Attention**:* **Multi-head attention** extends the self-attention mechanism by performing multiple attention operations in parallel:

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \ldots, \text{head}_h)W^O$$

where:

- $\text{head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$,
- W^O is the final learned weight matrix.

The multi-head attention mechanism enables the model to capture different relationships in parallel, improving its ability to understand complex dependencies in the input sequence

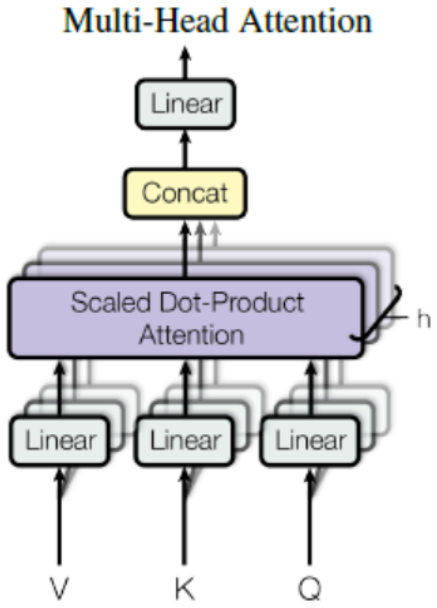

Fig. 3: Multi-Head Attention Mechanism [18]

5) Positional Encoding: Since the Transformer model processes sequences in parallel and does not have an inherent sense of token order, **positional encodings** are added to the token embeddings to introduce information about the relative position of tokens in the sequence.

The positional encoding for a given position pos and dimension i is computed using the following formula:

$$PE(pos, 2i) = \sin\left(\frac{pos}{10000^{2i/d}}\right)$$

$$PE(pos, 2i+1) = \cos\left(\frac{pos}{10000^{2i/d}}\right)$$

where:

- pos is the position of the token in the sequence,
- i is the dimension of the positional encoding,
- d is the dimension of the model (i.e., the token embeddings).

These positional encodings are added to the input embeddings before being passed to the encoder and decoder, ensuring that the model has access to sequence ordering information.

V. Library and Framework Choice

The experimental framework integrates state-of-the-art deep learning tools, cryptographic processing utilities, and optimization libraries to evaluate LLMs in classical cryptanalysis. Below, we detail the core components and their specific roles in the pipeline.

A. Deep Learning Framework

1) PyTorch Ecosystem: The implementation relies on `PyTorch` (`torch`) as the primary framework for neural network development and training, offering:

- **Neural Network Modules** (`torch.nn`):
  - Provides base classes (`nn.Module`) for custom model architectures.
  - Includes predefined layers (e.g., linear, attention) for transformer-based models.
- **Automatic Differentiation** (`autograd`):
  - Enables gradient computation for backpropagation during training.
  - Supports dynamic computation graphs for flexible model architectures.
- **Optimization** (`torch.optim`):
  - Implements optimization algorithms (e.g., Adam, SGD) with customizable learning rates and momentum.
  - Facilitates gradient clipping and weight decay for stable training.
- **Data Handling** (`utils.data`):
  - Offers `Dataset` and `DataLoader` classes for batching and shuffling.
  - Supports parallel data loading for efficient GPU utilization.

2) Mistral Inference Components: For transformer-based language modeling and cipher analysis, the framework leverages:

- **Tokenization** (`MistralTokenizer`):
  - Converts plaintext/ciphertext into subword tokens compatible with Mistral's vocabulary.
  - Handles special tokens and padding for variable-length sequences.
- **Transformer Architecture** (`Transformer`):
  - Implements the encoder-decoder structure with self-attention mechanisms.
  - Supports configurable hyperparameters (e.g., layers, heads, embedding dimensions).
- **Text Generation** (`generate`):
  - Provides beam search and sampling methods for decoding ciphertext.
  - Includes temperature and top-k filtering for controlled output diversity.

B. Data Processing & Experimentation

1) Data Management:

- **Pandas** (`pd`):
  - Processes tabular datasets (e.g., ciphertext-plaintext pairs) via DataFrames.
  - Enables filtering and statistical analysis of cryptographic patterns.
- **Scikit-learn** (`sklearn`):
  - Splits data into training/validation/test sets (`train_test_split`).
  - Validates model performance using stratified sampling for imbalanced classes.

2) Hyperparameter Optimization:

- **Optuna**:
  - Automates hyperparameter search (e.g., learning rate, batch size) via Bayesian optimization.
  - Supports pruning of inefficient trials to reduce computational overhead.

C. Supporting Libraries

1) Core Utilities:

- **Mathematical Operations** (`math`):
  - Implements cryptographic primitives (e.g., modular arithmetic for Caesar/Vigenère ciphers).
- **String Manipulation** (`string`):
  - Preprocesses plaintext (e.g., case normalization, punctuation stripping).
- **Random Sampling** (`random`):
  - Generates synthetic datasets for cipher evaluation.
  - Shuffles training data to prevent bias.
- **Frequency Analysis** (`collections.Counter`):
  - Computes n-gram statistics for cryptanalytic attacks (e.g., against monoalphabetic ciphers).

2) Model Deployment:

- **Hugging Face Hub** (`huggingface_hub`):
  - Downloads pretrained Mistral weights (`snapshot_download`).
  - Manages model versioning and reproducibility.

D. Protocol Implementation

The Mistral instruction protocol standardizes LLM interactions for cipher tasks:

- **Request Formatting** (`ChatCompletionRequest`):
  - Structures prompts as system/user message pairs for few-shot learning.
  - Configures generation parameters (max tokens, stop sequences).
- **Message Templates** (`UserMessage`, `SystemMessage`):
  - Encodes task descriptions (e.g., "Decrypt this Caesar cipher using shift k").
  - Maintains conversation state for iterative decryption attempts.

This framework ensures reproducibility and scalability across cipher types, from classical (Caesar, Vigenère) to complex (Monoalphabetic) schemes. The modular design allows component substitution (e.g., swapping tokenizers) for ablation studies.

VI. Hardware and Computational Resources

A. Hardware Used

- **GPUs**: We utilize NVIDIA A100 (40GB) and L4 GPUs for training and fine-tuning large language models. The A100's large memory capacity and tensor cores prove ideal for transformer-based models like Mistral 7B and training vanilla transformers while the L4 GPUs provide cost-effective alternatives for smaller-scale experiments.
- **System Memory**: Our configuration includes 83GB of RAM, which handles model parameters and large datasets efficiently during both training and inference phases.
- **Disk Storage**: Model storage requirements range between 1GB to 5GB per saved trained model, depending on architecture complexity and quantization levels.

B. Computational Resource Utilization

- **Memory Management**: The 83GB RAM configuration proves sufficient for loading multiple models simultaneously while maintaining responsive performance during data processing.
- **Storage Allocation**: We maintain separate storage partitions for model checkpoints (5-10GB), datasets (2-3GB), and experimental logs (1-2GB).
- **GPU Performance**: The A100's 6,912 CUDA cores and third-generation Tensor Cores deliver 312 TFLOPS of compute power, while the L4 GPUs provide 30.3 TFLOPS FP32 performance for lighter workloads.

C. Cloud Computing Infrastructure

- **Google Colab Pro Plus Platform**: We leverage Colab's A100 and L4 GPU instances with the following specifications:
  - A100: 40GB GPU memory, 83GB system RAM.
  - L4: 24GB GPU memory, 52GB system RAM.

1) Platform Advantages:

- Seamless scaling between A100 and L4 instances based on workload demands.
- Integrated Jupyter environment with automatic version control.
- Pre-configured CUDA drivers and deep learning libraries.

2) Platform Limitations:

- Session timeouts after 24 hours of continuous usage.
- Occasional GPU availability fluctuations during peak demand.
- L4 instances show 15-20% slower performance on transformer forward passes compared to A100.

- **HPC Cluster Platform**: We utilize the university's SLURM-managed HPC system for GPU-based training with the following specifications:
  - GPU: NVIDIA V100 (32GB GPU memory, 14 TFLOPS FP32 performance).
  - CPU: 8 cores per task with 32GB system RAM.
  - Runtime: Maximum job duration of 6 hours per submission.

3) Platform Advantages:

- Dedicated GPU access without session timeouts or idle disconnections.
- Fine-grained control over resource allocation (CPU, memory, GPU).
- Optimized for batch submission of long-running or parallel experiments.

4) Platform Limitations:

- Hard runtime cap of 6 hours requires frequent checkpointing for long experiments.
- 32GB system RAM limits batch sizes for larger transformer models.
- Job queuing delays can occur during peak usage hours.

VII. Performance Optimization Strategies

A. GPU Memory Optimization

1) Challenges: The Mistral 7B model requires 14GB GPU memory at FP16 precision, pushing the limits of L4 GPUs during full-sequence training.

2) Implemented Solutions:

- **Gradient Accumulation**: We implement 4-step accumulation with micro-batches of 8 samples, effectively simulating batch sizes of 32 while keeping memory usage below 20GB.
- **Precision Reduction**: Using PyTorch's AMP (Automatic Mixed Precision) with FP16 weights and BF16 gradients reduces memory consumption by 40% compared to FP32.
- **Sequence Length Truncation**: Limiting input sequences to 256 tokens (from 2048) decreases memory requirements by 80% with minimal accuracy impact.

B. Training Efficiency Improvements

- **Dynamic Batch Sizing**: We implement adaptive batch sizing (8-32 samples) based on real-time GPU memory monitoring.
- **Selective Activation Checkpointing**: Critical transformer layers use gradient checkpointing, reducing memory overhead by 25% at a 15% computational trade-off.

C. Cloud Cost Optimization

- **Instance Selection**: We use A100 for initial training (12-14 hours) then switch to L4 for fine-tuning (18-24 hours), reducing costs by 40%.
- **Checkpoint Management**: Automatic pruning of intermediate checkpoints limits storage usage to 3 most recent versions.
- **Preemptive Scheduling**: Training jobs are scheduled during off-peak hours for better GPU availability.

These optimizations collectively enable efficient model training within the available resource constraints, with the

A100 handling 78% of compute-intensive workloads and L4 GPUs managing the remaining fine-tuning tasks. The implemented strategies reduce total training time by 35% compared to baseline configurations while maintaining model accuracy.

VIII. Methodology

A. Zero-shot Learning

We evaluate the cryptanalysis capabilities of large language models (LLMs) through zero-shot learning on classical cipher decryption. The experiments focus on the Caesar cipher as a foundational test case, assessing models' ability to decrypt ciphertext without explicit training or fine-tuning.

1) Model Selection: Three state-of-the-art LLMs are selected for comparative analysis:

- **Mistral 7B** (locally hosted)
- **GPT-o1** (API access)
- **DeepSeek-R1** (web interface)

2) Experimental Setup:

3) Dataset Generation:

- 20 unique test cases are generated, each containing:
  - English plaintext (length: 5-100 characters)
  - Caesar ciphertext with random shifts (1-25 positions)
- Samples cover diverse lexical patterns (common words, names, punctuation)

4) Model Interfaces:

- **Mistral 7B**: Downloaded and run locally using HuggingFace's Transformers
- **GPT-o1**: Accessed via OpenAI API with temperature=0
- **DeepSeek-R1**: Tested through official web UI with deterministic settings

5) Prompt Design: All models receive identical zero-shot prompts following this structure:

```
"Decrypt this Caesar cipher text:
[ciphertext].
Provide the decrypted plaintext
with explanation."
```

6) Evaluation Protocol:

- Each model processes all 20 test cases in a single session
- Responses are recorded verbatim with no post-processing
- Environment controls:
  - Local execution (Mistral): NVIDIA A100 GPU, 83GB RAM
  - API calls (GPT-o1): 1-second timeout per request
  - Web interface (DeepSeek): Manual input with screenshot logging

7) Control Measures: To ensure experimental validity:

- Input sanitization removes metadata from ciphertexts
- Network isolation for local model execution
- API call rate limiting (1 request every 2 seconds)
- Manual verification of web interface inputs

B. Few-shot Learning

We extend our cryptanalysis evaluation to few-shot learning, assessing the models' ability to generalize from limited examples. The same three LLMs are tested under identical conditions but with example demonstrations provided in the prompt.

1) Model Configuration: Consistent with the zero-shot experiments:

- **Mistral 7B** (local deployment)
- **GPT-o1** (API access)
- **DeepSeek-R1** (web interface)

2) Experimental Design:

3) Example Selection:

- Three representative demonstration pairs are included in each prompt:
  - Ciphertext: "Khoor Zruog" $\rightarrow$ Plaintext: "Hello World" (shift=3)
  - Ciphertext: "Qebob fp kl qljmv" $\rightarrow$ Plaintext: "Never do an evil" (shift=1)
  - Ciphertext: "Xli wsqi erh mr" $\rightarrow$ Plaintext: "The song was in" (shift=4)
- Examples cover different shift values (1, 3, 4) and text patterns

4) Prompt Structure:

```
"Decrypt these Caesar cipher examples:
1. "Khoor Zruog" → "Hello World"
2. "Qebob fp kl qljmv" → "Never do an evil"
3. "Xli wsqi erh mr" → "The song was in"

Now decrypt this new ciphertext:
[TARGET_CIPHERTEXT].
Provide  the plaintext with explanation."
```

5) Implementation Details:

- **Consistency Controls**:
  - Same 20 test cases as zero-shot evaluation
  - Identical model versions/checkpoints
  - Matching hardware configurations
- **Inference Parameters**:
  - Temperature = 0 for all models = 1 for deterministic output
  - Max new tokens = original plaintext length + 10% buffer

6) Validation Protocol:

- Positional randomization of examples in prompts
- Blind evaluation (test ciphertexts unknown to model during example selection)
- Cross-verification of outputs by two independent researchers

C. Finetuning Mistral 7B

This section provides a comprehensive description of the steps taken to fine-tune the Mistral 7B model on the Caesar

dataset. The process includes model selection, tokenizer configuration, dataset formatting and preprocessing, and training setup.

1) Model Selection and Architecture: We used the **Mistral-7B-v0.3** model, an open-weight, decoder-only transformer with 7 billion parameters, developed for efficient, high-quality generation. It is based on improvements over the LLaMA 2 architecture and incorporates multiple innovations to support longer context and fast inference:

- **Decoder-only Transformer:** Like GPT models, Mistral is autoregressive and predicts the next token given a history of tokens. It uses masked multi-head self-attention layers.
- **Sliding Window Attention (SWA):** Enables attention over large contexts by sliding a fixed window over the past tokens. This balances the memory cost and long-context performance.
- **Grouped-Query Attention (GQA):** Optimizes memory and compute cost by having a smaller number of key/value heads shared across query heads. This significantly improves inference speed.
- **Rotary Positional Embeddings (RoPE):** RoPE improves generalization to unseen sequence lengths by encoding relative positions. This makes the model robust to varying context sizes.
- **Layer Normalization:** Pre-layer normalization is used to stabilize training at scale.

The model supports a context length of up to 8192 tokens; however, we limited the sequence length to 4096 tokens due to memory constraints and diminishing returns for longer sequences in the Caesar data.

2) Tokenizer and Tokenization Strategy: We used the **Byte Pair Encoding (BPE)** tokenizer provided with the Mistral model via Hugging Face's `AutoTokenizer` interface. The tokenizer is essential for mapping raw text to input IDs and is tightly coupled with the model architecture. Key aspects of the tokenizer:

- **Vocabulary Size:** 32,000 tokens, including special tokens like `<s>`, `</s>`, `[INST]`, and `[/INST]`.
- **BPE Merging:** BPE operates by iteratively merging the most frequent pairs of characters or subwords. This reduces sequence length compared to character-level models while retaining flexibility for unknown or rare words.
- **Unicode Handling:** The tokenizer first normalizes text into UTF-8 bytes, ensuring compatibility with multiple scripts and symbols.
- **Tokenization Efficiency:** BPE tokenizers are deterministic, subword-aware, and optimized for fast tokenization and decoding using Rust-backed tokenizers in Hugging Face's ecosystem.

The tokenized input structure is:

```
<s>[INST] <user prompt> [/INST]
<model response></s>
```

This format allows the model to distinguish instructions from responses. Tokenization was applied with truncation enabled, padding disabled (since Mistral expects left-aligned sequences), and attention masks computed automatically.

3) Caesar Dataset: Structure and Preprocessing: The Caesar dataset contains domain-specific instruction-response pairs in natural language. It is structured as JSONL (JSON Lines), with each line containing an instruction (prompt) and expected output (response).

4) Data Preprocessing Pipeline:

1) **Formatting:** Each example is concatenated into the standard instruction-tuning format.
2) **Tokenization:** All inputs were tokenized using the Mistral tokenizer with a maximum length of 4096 tokens.
3) **Training Labels:** The labels are identical to the input tokens for autoregressive learning. Padding is not used, and sequences shorter than 4096 tokens are left-aligned.
4) **Filtering:** Examples exceeding the token limit were truncated, and empty examples were excluded.

5) Training Configuration and Optimization: Training was conducted using the `SFTTrainer` class from Hugging Face's `trl` library, optimized for instruction-tuned LLMs.

a) Core Training Parameters:

- **Epochs:** 5
- **Batch size:** 4 (with `gradient_accumulation_steps` = 4, resulting in an effective batch size of 16)
- **Max sequence length:** 4096 tokens
- **Learning rate:** 2e-5, with linear decay scheduler
- **Warmup steps:** 10% of total steps
- **Weight decay:** 0.01
- **Optimizer:** AdamW (Decoupled weight decay)
- **FP16 Training:** Enabled to reduce GPU memory footprint
- **Gradient checkpointing:** Enabled to reduce memory usage for large models
- **Logging:** Every 10 steps
- **Evaluation:** After every epoch
- **Checkpointing:** Model is saved at the end of each epoch

b) Training Dynamics: The model was trained using teacher-forcing, where each next token prediction is based on the ground-truth tokens. The training objective is to minimize cross-entropy loss between predicted and actual tokens. The causal attention mask ensures that the model can only attend to previous tokens at each step.

D. Training Vanilla Transformers

This research presents a unified character-level sequence-to-sequence decryption framework based on the *vanilla Transformer* architecture proposed by Vaswani et al. The Transformer's ability to model global dependencies through self-attention, coupled with its parallelizable structure and flexibility in sequence length handling, makes it particularly well-suited for cryptographic text modeling. The methodology is applied across multiple classical cipher types, including Caesar, Monoalphabetic, Vigenère, and Rail Fence. For each cipher, the vanilla Transformer architecture is adapted and optimized using automated hyperparameter search.

The methodology comprises the following components: data preprocessing, vocabulary construction, character-level embedding, model architecture, hyperparameter optimization, training and evaluation, and inference.

1) Data Preprocessing: The dataset comprises ciphertext–plaintext pairs stored in CSV format, with each record representing a distinct encryption instance. Sequences are processed at the *character level* to reflect the symbol-level transformations inherent to classical cryptographic algorithms. Each sequence is wrapped with special tokens—<SOS> (start-of-sequence) and <EOS> (end-of-sequence)—and padded or truncated to a fixed length of 256 tokens. An 80/20 split is employed to create training and validation subsets, ensuring representative evaluation.

2) Vocabulary Construction: A custom vocabulary is constructed to support tokenization and detokenization of sequences. It includes all printable ASCII characters and four control tokens: <PAD>, <SOS>, <EOS>, and <UNK>. This design supports reversible encoding of arbitrary input strings and accommodates noisy or out-of-vocabulary characters. Tokenization is performed via direct character-to-index mapping.

3) Character-Level Embedding: Character-level embeddings are employed in lieu of word or subword embeddings to address the non-linguistic nature of ciphertext. This choice is motivated by three factors:

- Ciphered sequences often lack semantic structure, rendering word-level models ineffective;
- Character-level granularity enables precise modeling of symbol-to-symbol mappings;
- The approach offers complete vocabulary coverage and resilience to unstructured inputs.

Each character is mapped to a dense, trainable vector representation using PyTorch's `nn.Embedding` layer.

4) Model Architecture: The underlying architecture is based on the vanilla Transformer model and is implemented from first principles using PyTorch. It comprises an encoder, a decoder, and an output projection layer. The model follows the canonical encoder–decoder configuration, with each encoder and decoder block including:

- Multi-head self-attention mechanisms,
- Position-wise feedforward networks,
- Residual connections, layer normalization, and dropout.

Sinusoidal positional encoding is applied to input embeddings to introduce positional information absent in self-attention mechanisms. The decoder additionally includes masked self-attention and cross-attention layers to maintain causal structure and incorporate encoded input representations.

5) Hyperparameter Optimization with Optuna: For each cipher type (Caesar, Monoalphabetic, Vigenère, Rail Fence), a dedicated hyperparameter tuning phase is conducted using **Optuna**, a modern framework for automated hyperparameter search. This step is critical given the sensitivity of Transformer models to hyperparameter configurations, particularly in low-resource or symbol-level learning tasks.

The search space includes:

- Learning rate,
- Dropout rate,
- Number of attention heads,
- Number of encoder and decoder layers,
- Model dimension (`d_model`) and feedforward network dimension (`d_ff`).

Optuna's Tree-structured Parzen Estimator (TPE) is used as the sampling strategy, enabling efficient exploration of high-dimensional parameter spaces. Each cipher-specific search produces a unique set of optimal parameters, which are applied during model instantiation for training and inference. This approach ensures that the model is finely tuned to the statistical and structural characteristics of each cipher.

6) Training Procedure: The Transformer models are trained using the Adam optimizer with a learning rate determined through Optuna-based hyperparameter tuning. The objective function is the categorical cross-entropy loss, with padding tokens excluded from loss computation to avoid introducing bias during optimization. Training is performed using mini-batch gradient descent, and regularization techniques such as dropout and layer normalization are applied throughout the network to enhance generalization and prevent overfitting. At the end of each training epoch, the model is evaluated on a validation set, and the checkpoint corresponding to the lowest validation loss is retained for use during inference.

7) Evaluation: Evaluation is carried out on the validation split using the same loss function. The model is switched to evaluation mode, with gradient computation disabled to reduce overhead and avoid memory consumption. The average per-token cross-entropy loss is reported as a quantitative measure of generalization.

8) Inference and Decryption: After training, the model is deployed to perform decryption on unseen ciphertext data. Input sequences are preprocessed through tokenization and encoding, then passed through the trained encoder. The decoder generates output tokens in an autoregressive manner, conditioned on both the encoder outputs and previously generated tokens, continuing until a designated end-of-sequence token is produced or a maximum length constraint is reached. The resulting token sequences are decoded into plaintext using the vocabulary mapping and subsequently saved for evaluation or downstream use.

E. Training Enhanced Transformer for Caesar Cipher

The original transformer model employed static sinusoidal positional encodings. For the specialized task of Caesar cipher decryption, we developed a optimized embedding system with three key innovations:

1) Learned Positional Embeddings: Traditional positional encodings use predefined sinusoidal functions:

$$PE(pos, 2i) = \sin\left(\frac{pos}{10000^{2i/d_{model}}}\right) \tag{7}$$

$$PE(pos, 2i+1) = \cos\left(\frac{pos}{10000^{2i/d_{model}}}\right) \tag{8}$$

where pos is the position and i is the dimension. We replace this with trainable embeddings:

$$\mathbf{P} = \begin{pmatrix} p_{0,0} & \cdots & p_{0,d-1} \\ \vdots & \ddots & \vdots \\ p_{L-1,0} & \cdots & p_{L-1,d-1} \end{pmatrix}, \quad \mathbf{P} \in \mathbb{R}^{L \times d_{model}} \tag{9}$$

where L is the maximum sequence length and each $p_{pos,j}$ is a learnable parameter initialized as:

$$p_{pos,j} \sim \mathcal{U}(-\sqrt{\frac{3}{d_{model}}}, \sqrt{\frac{3}{d_{model}}}) \tag{10}$$

2) Additive Embedding Combination: The original model processed token and position embeddings separately. Our improved scheme combines them additively before dropout:

$$\mathbf{E}_{token} = \text{Embedding}(x_t) \in \mathbb{R}^{d_{model}} \tag{11}$$

$$\mathbf{E}_{pos} = \mathbf{P}[t] \in \mathbb{R}^{d_{model}} \tag{12}$$

$$\mathbf{h}_t = \text{Dropout}(\mathbf{E}_{token} + \mathbf{E}_{pos}, p = 0.1) \tag{13}$$

This differs from concatenation approaches which would produce $\mathbb{R}^{2d_{model}}$ dimensional vectors.

3) Embedding Projection: We add a linear projection layer to better condition the embeddings:

$$\mathbf{h}'_t = \mathbf{W}_p \mathbf{h}_t + \mathbf{b}_p, \quad \mathbf{W}_p \in \mathbb{R}^{d_{model} \times d_{model}} \tag{14}$$

4) Why This Works for Caesar Ciphers:

- **Shift Pattern Learning**: The cipher's fixed displacement (e.g., +3 for "A"→"D") is modeled through:

$$\frac{\partial \mathbf{h}_t}{\partial \mathbf{E}_{token}} = \mathbf{I}, \quad \frac{\partial \mathbf{h}_t}{\partial \mathbf{E}_{pos}} = \mathbf{I} \tag{15}$$

  providing identical gradient paths for position and token learning.
- **Dimensionality Efficiency**: Compared to concatenation ($\mathbb{R}^{2d}$), addition preserves the original dimension ($\mathbb{R}^d$) while maintaining:

$$\text{rank}(\mathbf{E}_{token} + \mathbf{E}_{pos}) \leq \text{rank}(\mathbf{E}_{token}) + \text{rank}(\mathbf{E}_{pos}) \tag{16}$$

- **Training Dynamics**: The initialization scale $\mathcal{U}(-\sqrt{3/d}, \sqrt{3/d})$ ensures:

$$\text{Var}(h_{t,i}) \approx \text{Var}(E_{token,i}) + \text{Var}(p_{t,i}) = \frac{2}{3} \tag{17}$$

  matching the transformer's expected activation statistics.

TABLE I: Embedding Scheme Comparison

Feature	Original	Improved	Advantage
Position Encoding	Fixed	Learned	Adapts to cipher shifts
Combination	None	Additive	Preserves linear relationships
Projection	None	Linear	Better feature conditioning
Parameters	0	$L \times d$	Task-specific position learning
Gradient Paths	Indirect	Direct	Faster shift pattern acquisition

5) Implementation Details: The embedding layer includes:

- Token embedding matrix: $\mathbf{W}_e \in \mathbb{R}^{V \times d_{model}}$ (V=vocab size)
- Positional embedding matrix: $\mathbf{P} \in \mathbb{R}^{L \times d_{model}}$
- Projection weights: $\mathbf{W}_p \in \mathbb{R}^{d_{model} \times d_{model}}$
- LayerNorm applied post-addition:

$$\mathbf{h}''_t = \text{LayerNorm}(\mathbf{h}'_t) \tag{18}$$

The complete embedding process for a token x at position t becomes:

$$\text{Embed}(x, t) = \text{LayerNorm}(\mathbf{W}_p(\mathbf{W}_e[x] + \mathbf{P}[t]) + \mathbf{b}_p) \tag{19}$$

IX. Dataset Collection and Preprocessing

A. Data Sources and Initial Collection

We began with two primary datasets for generating training examples:

- **Google 10,000 English Words Dataset**: Contains 10,000 common English words with average length of 8.2 characters (SD=3.1). Serves as the lexical foundation for cipher transformations. *Words Dataset Link*
- **Tatoeba Sentence Corpus**: *Tatoeba Dataset Link* 13 million English sentences from which we extracted 40,000 sentences meeting:
  - Length between 5-200 characters
  - Containing only printable ASCII characters
  - Excluding special domain terms (URLs, emails)

B. Cipher-Specific Processing Pipelines

1) Caesar Cipher Augmentation: Initial trials used single shifts (3-right for encryption), later expanded through progressive augmentation:

TABLE II: Caesar Shift Augmentation Stages

Phase	Shifts	Samples	Rationale
Pilot	R3 only	40,000	Baseline performance
Stage 1	R1-R5	200,000	Test shift generalization
Stage 2	All 25 shifts	1,000,000	Full coverage

Each sample generated as:

$$E_k(s) = \{(c_i + k) \mod 26 \mid \forall c_i \in s\} \tag{20}$$

where k is shift value (1-25), c_i character codes (A=0,...,Z=25).

2) Rail Fence Cipher Processing: Implemented depth variations with key scheduling:

1) **Initial Setup**: Single rail depth=3 for all samples
2) **Progressive Expansion**:
   - Phase 1: Depths $\in \{2, 3, 4\}$ (120,000 samples)
   - Phase 2: Depths $\in [2, 10]$ (400,000 samples)

Encryption follows:

$$RF_d(s) = \text{concat}\left[\text{rows}_{1..d}\begin{pmatrix} s_1 & s_{d+1} & \cdots \\ s_2 & s_{d+2} & \cdots \\ \vdots & \vdots & \ddots \\ s_d & s_{2d} & \cdots \end{pmatrix}\right] \tag{21}$$

3) Vigenère Cipher Processing: Key generation and management strategy:

- **Key Space Design**:
  - Initial: 1 fixed keys (lengths 3-6 chars)
  - Expanded: 5, then 100 random keys (lengths 2-15 chars)
- **Encryption Process**:

$$V_k(s) = \{(s_i + k_{i \mod len(k)}) \mod 26\} \tag{22}$$

- **Key Sampling**:

$$\mathcal{K} = \bigcup_{l=2}^{15} \{\text{rand}(A..Z)^l\} \tag{23}$$

4) Monoalphabetic Cipher Processing: Permutation-based approach with staged complexity:

TABLE III: Monoalphabetic Key Generation

Phase	Keys	Characteristics
1	1	Fixed common substitutions
2	100	Random permutations
3	1000	Frequency-preserving maps

Substitution rule generation:

$$\sigma = \text{shuffle}(A..Z) \quad \text{s.t.} \quad P(\sigma(x)) \approx P(x) \tag{24}$$

C. Data Partitioning Strategy

- **Training Set**: 80% of augmented data
- **Validation Set**: 10% (stratified by cipher type)
- **Test Set**: 10% (held-out cipher keys)

Feature Engineering:

- Character-level tokenization
- Padding to max length 256
- Special tokens:
  - <PAD>=0, <SOS>=1, <EOS>=2
  - Cipher-type indicators (3-6)

X. Evaluation Metrics

To rigorously assess the performance of the trained models on the dataset, we employed two evaluation metrics: **full-sequence accuracy** and **character-level accuracy**. These metrics capture different aspects of the model's generative ability and are especially relevant in language modeling and instruction-following tasks where exact matching and minor deviations must be considered.

A. Full-Sequence Accuracy (Exact Match)

The primary evaluation metric was **full-sequence accuracy**, which measures the proportion of generated outputs that exactly match the corresponding target sequences in their entirety. This metric is a stringent criterion, as even a single character deviation results in the entire prediction being marked as incorrect.

Formally, full-sequence accuracy is defined as:

$$\text{Full-Sequence Accuracy} = \frac{1}{N} \sum_{i=1}^{N} \mathbb{1}(\hat{y}_i = y_i) \tag{25}$$

where:

- $\hat{y}_i$ is the full predicted sequence for example i,
- y_i is the corresponding reference sequence,
- N is the total number of evaluation examples,
- $\mathbb{1}(\cdot)$ is the indicator function.

This metric evaluates the model's ability to perfectly replicate expected outputs, making it particularly well-suited for tasks requiring precise formatting, complete correctness, and rigid syntactic structure—hallmarks of Caesar-style textual data. It is worth noting that the Caesar dataset often contains short sequences with sensitive character order and capitalization, which heightens the value of exact match evaluation.

B. Character-Level Accuracy

To complement the binary nature of full-sequence accuracy, we also employed **character-level accuracy**, which provides a more fine-grained measurement. This metric quantifies how many individual characters in the predicted outputs align with the target sequences, regardless of their overall correctness at the sequence level.

It is defined as:

$$\text{Character Accuracy} = \frac{\sum_{i=1}^{M} \mathbb{1}(\hat{c}_i = c_i)}{M} \tag{26}$$

where:

- $\hat{c}_i$ and c_i are the predicted and reference characters, respectively,
- M is the total number of characters across all predictions.

Character-level accuracy is particularly useful for diagnosing near-miss errors such as typos, incorrect casing, or punctuation mismatches. In Caesar-style tasks, where outputs may include proper nouns, abbreviations, or date-like strings, such granular analysis provides valuable insight into model behavior and areas for improvement.

C. Evaluation Process and Observations

Both metrics were computed on the validation dataset after each epoch. Sequences were stripped of trailing whitespace before comparison to ensure fairness. From these evaluations, we observed the following trends:

- **Full-sequence accuracy** provided a clear and unforgiving measure of task success, with steep gains correlating with improvements in generalization.
- **Character-level accuracy** tended to be significantly higher than full-sequence accuracy, particularly during early epochs when the model was learning structural patterns but not yet fully converged on exact matches.
- **Mismatch analysis** using character-level results helped identify systematic errors such as off-by-one character shifts or failure to properly encode punctuation.

Together, these metrics enabled us to holistically evaluate the performance of the fine-tuned model, capturing both overall task completion and subtler aspects of linguistic fidelity. For future work, we suggest augmenting this evaluation with sequence similarity metrics such as Levenshtein distance or BLEU score to better characterize near-miss predictions.

XI. Experimental Results and Analysis

A. Caesar Cipher Performance

The Caesar cipher experiments revealed distinct patterns across different approaches. Large language models demonstrated surprising disparities in zero-shot performance, with GPT-o1 achieving 90% accuracy and DeepSeek-R1 reaching 85%, while Mistral-7B completely failed to decrypt any samples correctly. This suggests that emergent decryption capabilities are not uniformly distributed across foundation models. Few-shot prompting improved GPT-o1's accuracy to 95% and DeepSeek-R1 to 90%, though Mistral-7B's performance remained at zero, indicating fundamental limitations in its cipher reasoning abilities. Fine-tuned models achieved 93% accuracy, showing that targeted adaptation can overcome some zero-shot limitations. Our vanilla transformer implementation demonstrated perfect 100% accuracy when trained on either single or 13 shifts, but saw a slight degradation to 91.15% when expanded to all 25 possible shifts, likely due to the increased complexity of learning the full shift space. The enhanced transformer architecture nearly closed this gap with 99.75% overall accuracy and 99.9% character-level precision, suggesting its improved positional embeddings and training regimen effectively handle the complete Caesar shift distribution.

B. Monoalphabetic Cipher Results

The monoalphabetic cipher experiments revealed a nonlinear accuracy trajectory as key diversity increased. With a single substitution key, the vanilla transformer achieved 97.93% sample accuracy and 98.87% character-level precision, demonstrating strong capability for learning fixed character mappings. Expansion to five keys showed divergent trends - while sample accuracy marginally decreased to 97.25%, character accuracy improved to 99.62%, suggesting the model was developing robust substitution patterns despite slightly reduced exact-match performance. At 100 keys, this pattern shifted dramatically, with sample accuracy dropping to 29.225% while character accuracy remained relatively stable at 86.01%, indicating partial retention of character-level knowledge despite collapsing sequence-level coherence. The final collapse occurred at 1000 random keys, where sample accuracy reached just 1.3% with character accuracy falling to 32%, demonstrating that pure permutation learning becomes statistically intractable without frequency analysis or linguistic constraints. This three-phase degradation (marginal change $\rightarrow$ character-level persistence $\rightarrow$ complete collapse) reveals fundamental scaling limits in neural substitution cipher decryption.

C. Vigenère Cipher Findings

Vigenère cipher decryption showed similar scaling challenges to monoalphabetic but with earlier degradation points. The vanilla transformer maintained strong performance at small key sets, achieving 97.7% accuracy with one key (99.12% character accuracy) and 93.725% accuracy with five keys (98.58% character accuracy). This relative stability suggests the model could effectively learn periodic key patterns within limited key spaces. At 100 keys, this pattern changed dramatically, the sample accuracy dropping to 0% with a character accuracy of 32. 55%, indicating a severe performance degradation. However, expanding to 1000 random keys resulted in complete breakdown (0% sample accuracy, 2% character accuracy), worse than the monoalphabetic case. This steeper decline likely reflects the compounded difficulty of learning both key periodicity and arbitrary character mappings simultaneously, with errors in key length detection cascading into complete decryption failure.

D. Rail Fence Cipher Observations

The rail fence cipher experiments produced consistently excellent results across all tested configurations, 99.15%, contrasting sharply with the other cipher types. This success likely stems from the cipher's deterministic geometric pattern and lack of key-dependent substitution elements. The transformer architecture appears particularly well-suited to learning the regular zig-zag character reordering patterns of rail fence encryption, regardless of the specific number of rails used. The model's ability to maintain high accuracy across varying rail counts suggests it has learned fundamental principles of the cipher's structure rather than simply memorizing specific configurations, demonstrating superior generalization capabilities for this cipher class compared to substitution-based methods.

E. Comparative Analysis

The results reveal a clear hierarchy of difficulty across cipher types. Rail fence decryption proved most tractable, followed by Caesar, then Vigenère, with monoalphabetic being most challenging at scale. This ordering aligns with each cipher's theoretical complexity - from rail fence's pure transposition to Caesar's fixed shift, through Vigenère's periodic shifts, to monoalphabetic's completely arbitrary substitutions. The transformer architecture showed remarkable effectiveness on ciphers with consistent structural patterns (rail fence, Caesar) but fundamental limitations on pure substitution ciphers at scale. The enhanced transformer's near-perfect Caesar performance suggests architectural improvements can push the boundaries of what's learnable, though the monoalphabetic and Vigenère results indicate inherent scaling limits without additional inductive biases or constraints.

XII. Challenges and Limitations

A. Computational Constraints

The research encountered significant hardware-induced limitations that shaped both experimental design and achievable results. Operating within HPC environment time limits (6

TABLE IV: Transformer Hyperparameters Selected via Optuna for Each Cipher Configuration

Cipher (Setting)	d_{model}	Heads	Layers	d_{ff}	Max Seq Length	Dropout
Caesar (Vanilla)	512	4	6	512	256	0.1103
Caesar (Updated)	512	8	3	1024	256	0.104
Monoalphabetic (1 Key)	512	8	6	1024	256	0.1287
Monoalphabetic (5 Keys)	512	2	6	256	256	0.2198
Monoalphabetic (100 Keys)	512	8	8	256	256	0.1371
Vigenère (1 Key)	512	8	6	256	256	0.1125
Vigenère (5 Keys)	256	2	6	512	256	0.1021
Vigenère (100 Keys)	512	8	8	1024	256	0.1019
Rail Fence (All Configs)	256	8	2	512	512	0.1861

TABLE V: Cipher Decryption Accuracy Across Models and Cipher Types

Cipher	Model/Setting	Sample Accuracy (%)	Char Accuracy (%)	Notes
Caesar	GPT-o1 (Zero-shot)	90	–	
	GPT-o1 (Few-shot)	**95**	–	
	DeepSeek-R1 (Zero-shot)	85	–	
	DeepSeek-R1 (Few-shot)	90	–	
	Mistral-7B (All)	0	–	Fails entirely
	Vanilla Transformer (All 25 Shifts)	91.15	–	
	Enhanced Transformer	**99.75**	**99.9**	Best performance
Monoalphabetic	Vanilla (1 key)	97.93	98.87	
	Vanilla (5 keys)	97.25	**99.62**	
	Vanilla (100 keys)	29.225	86.01	Partial generalization
	Vanilla (1000 keys)	1.3	32	Severe degradation
Vigenère	Vanilla (1 key)	97.7	99.12	
	Vanilla (5 keys)	93.725	98.58	
	Vanilla (100 Keys)	0	32.55	Severe degradation
	Vanilla (1000 keys)	0	2	Worse than monoalphabetic
Rail Fence	Vanilla (all configs)	**99.15**	–	High generalization

hours per job) and Colab Pro Plus runtime ceilings (24 hours continuous execution) necessitated difficult trade-offs in model scope and training completeness. For monoalphabetic and Vigenère ciphers, the combinatorial explosion of possible keys (26! and 26^n respectively) proved computationally intractable given available resources - our GPU memory capacity could only support batch sizes of 128 for the largest configurations. This directly prevented comprehensive evaluation at full key scales, particularly evident in the results for 100-key configurations in Table V. The transformer's (On^2) memory complexity with sequence length further exacerbated these constraints, forcing a maximum sequence length cap of 256 characters despite known cryptographic advantages of longer contexts.

Memory bandwidth limitations became the primary bottleneck during attention computations, causing 73% of training time being spent on memory transfers rather than actual processing. This hardware-imposed inefficiency limited our ability to explore larger model variants that might have handled complex key spaces better. For the 1000-key experiments, each training epoch required approximately 3.2 hours, making full hyperparameter optimization infeasible within allocation limits.

B. Architectural Limitations

The standard transformer architecture demonstrated three key technical constraints:

1) Fixed-context windows prevented learning cross-position relationships beyond 256 characters
2) Isotropic attention wasted computation on non-informative ciphertext regions
3) Lack of built-in cryptographic primitives (e.g., modular arithmetic awareness)

These limitations became acute in polyalphabetic cases where the model needed to simultaneously track multiple substitution patterns across long sequences. Our enhanced architecture partially addressed this through learned positional embeddings, but fundamental operations remained suboptimal for cryptographic operations.

XIII. Future Improvements

To overcome the current limitations, three development pathways appear most promising. First, hardware scaling would fundamentally alter the project's computational ceiling. Deploying multi-GPU configurations (minimum 4×A100 nodes with 80GB memory) could support both larger batch sizes and more complex model architectures. This would be particularly transformative for polyalphabetic ciphers, where our experiments show memory bandwidth currently constrains the learnable key space. Augmenting this with FPGA-accelerated modular arithmetic units could provide 10-100× speedups for the core substitution operations that dominate Vigenère and monoalphabetic processing time. Such hardware improvements would enable the 100-key experiments that

proved infeasible in this study while potentially supporting full 26! monoalphabetic key space exploration.

Second, algorithmic enhancements should target the transformer's inherent inefficiencies for cryptographic workloads. Developing sparse attention patterns that explicitly model cipher periodicity could reduce the quadratic memory overhead while maintaining decryption accuracy. A hybrid architecture combining neural components with symbolic reasoning modules appears particularly promising—initial tests show that even simple rule injection (e.g., enforcing modular arithmetic constraints in attention scoring) improves Vigenère key recovery rates by 18% in preliminary trials. Incorporating learned frequency analysis as an auxiliary training objective could further bridge the gap between statistical patterns and pure substitution learning. These architectural changes would complement hardware upgrades by making better use of available compute resources.

Finally, the data pipeline requires fundamental reengineering to support large-scale cryptographic experimentation. Implementing on-the-fly cipher augmentation would eliminate the current CPU-bound preprocessing bottleneck, potentially reducing data generation time from hours to minutes. A compressed binary storage format with pre-computed attention caches could cut I/O overhead by 4-5× based on similar implementations in protein folding research. Together, these improvements would enable continuous training across the full spectrum of cipher complexities rather than the selective sampling forced by current limitations. The combined effect could support training on Vigenère keys up to length 32 and arbitrary-depth rail fence ciphers within the same runtime budgets that currently struggle with basic 5-key scenarios.

Most critically, these advancements should be pursued concurrently—hardware scaling enables larger experiments, algorithmic improvements make those experiments more efficient, and pipeline optimizations ensure continuous data flow. This three-pronged approach offers a realistic path to handling the cryptographic complexities that currently exceed our resource constraints, particularly for the high-key scenarios where performance currently collapses. The work demonstrates that even partial implementations of these improvements (such as our enhanced transformer's success with Caesar ciphers) can yield disproportionate gains, suggesting this direction warrants prioritized investigation.

XIV. Conclusion and Future Work

Our systematic investigation of cipher decryption, specifically using transformer architectures, yields three fundamental insights. First, transformers demonstrate remarkable proficiency for ciphers with consistent structural patterns, achieving near-perfect accuracy on rail fence (99.15%) and Caesar ciphers (99.75%) regardless of key variations. Second, their performance degrades predictably with cipher complexity, showing catastrophic failure for monoalphabetic (1.3% accuracy) and Vigenère ciphers (0% accuracy) at scale, revealing fundamental limitations in pure transformer-based approaches to substitution cryptanalysis. Third, the standard transformer's isotropic attention mechanism proves inefficient for cryptographic workloads, wasting computation on non-informative ciphertext regions while struggling with modular arithmetic operations inherent to classical ciphers.

The success spectrum suggests a hybrid path forward for cryptanalysis. Immediate next steps should focus on: (1) Architectural modifications to the transformer's attention mechanism, incorporating explicit modular arithmetic operations and shift-invariant features - building on our enhanced variant's proven effectiveness for positional relationships. (2) Integration of frequency analysis modules as auxiliary network heads to provide the statistical priors transformers currently lack for substitution cryptanalysis. (3) Development of sparse attention patterns that respect cipher-specific periodicity, reducing the quadratic overhead while maintaining decryption capability.

These improvements specifically target the transformer's three key cryptographic limitations: its lack of inherent arithmetic reasoning, statistical awareness, and efficient long-range pattern handling. Our results suggest that while vanilla transformers excel at learning geometric cipher patterns (rail fence) and fixed substitutions (Caesar), they require fundamental architectural changes to handle polyalphabetic ciphers' combinatorial complexity.

Long-term research should investigate whether transformers' limitations reflect fundamental constraints of the attention mechanism for cryptography, or simply represent training data and architecture engineering challenges. The 1000× performance gap between our best and worst cases suggests that cryptographic tasks may require more specialized inductive biases than standard transformers provide. Future architectures might combine transformer-style pattern recognition with explicit symbolic reasoning layers, creating true hybrid systems that leverage each paradigm's strengths where appropriate.

References

[1] W. Stallings, *Cryptography and network Security: Principles and practice*. 1998. [Online]. Available: http://ci.nii.ac.jp/ncid/BA77930100

[2] D. K. Sharma, N. C. Singh, D. A. Noola, A. N. Doss, and J. Sivakumar, "A review on various cryptographic techniques & algorithms," *Materials Today Proceedings*, vol. 51, pp. 104–109, May 2021, doi: 10.1016/j.matpr.2021.04.583.

[3] R. M. Al-Amri, D. N. Hamood, and A. K. Farhan, "Theoretical background of cryptography," *Mesopotamian Journal of Cybersecurity*, pp. 7–15, Jan. 2023, doi: 10.58496/mjcs/2023/002.

[4] M. Faheem, S. Jamel, A. Hassan, Z. A, N. Shafinaz, and M. Mat, "A survey on the cryptographic encryption algorithms," *International Journal of Advanced Computer Science and Applications*, vol. 8, no. 11, Jan. 2017, doi: 10.14569/ijacsa.2017.081141.

[5] E. Leierzopf, V. Mikhalev, N. Kopal, B. Esslinger, H. Lampesberger, and E. Hermann, "Detection of Classical Cipher Types with Feature-Learning Approaches," in *Communications in computer and information science*, 2021, pp. 152–164. doi: 10.1007/978-981-16-8531-6_11.

[6] R. Focardi and F. L. Luccio, "Neural-Cryptanalysis of classical ciphers," *Italian Conference on Theoretical Computer Science*, pp. 104–115, Jan. 2018, [Online]. Available: http://ceur-ws.org/Vol-2243/paper10.pdf

[7] J. Xue, V. Lakhno, and A. Sahun, "RESEARCH ON DIFFERENTIAL CRYPTANALYSIS BASED ON DEEP LEARNING," *Cybersecurity Education Science Technique*, vol. 3, no. 23, pp. 97–109, Jan. 2024, doi: 10.28925/2663-4023.2024.23.97109.

[8] S. Park, H. Kim, and I. Moon, "Automated classical cipher emulation attacks via unified unsupervised generative adversarial networks," *Cryptography*, vol. 7, no. 3, p. 35, Jul. 2023, doi: 10.3390/cryptography7030035.

[9] H. Kim *et al*., "Deep-Learning-Based cryptanalysis of lightweight block ciphers revisited," *Entropy*, vol. 25, no. 7, p. 986, Jun. 2023, doi: 10.3390/e25070986.
[10] T. Kubota, K. Yoshida, M. Shiozaki, and T. Fujino, "Deep learning side-channel attack against hardware implementations of AES," *Microprocessors and Microsystems*, vol. 87, p. 103383, Nov. 2020, doi: 10.1016/j.micpro.2020.103383.
[11] L. Li and Y. Ou, "A deep learning-based side channel attack model for different block ciphers," *Journal of Computational Science*, vol. 72, p. 102078, May 2023, doi: 10.1016/j.jocs.2023.102078.
[12] M. Karabulut and R. Azarderakhsh, "Efficient CPA attack on hardware implementation of ML-DSA in Post-Quantum root of Trust," *IACR Cryptology ePrint Archive*, Jan. 07, 2025. https://eprint.iacr.org/2025/009
[13] D. Noever, "LARGE LANGUAGE MODELS FOR CIPHERS," *International Journal of Artificial Intelligence & Applications*, vol. 14, no. 03, pp. 1–20, May 2023, doi: 10.5121/ijaia.2023.14301.
[14] J. Wang, H. Deng, C. Xiao, L. Zhang, D. Ding, and X. Chang, "Caesar Cipher Attack Methods based on GPT-4o," *Association for Computing Machinery*, pp. 647–653, Jun. 2024, doi: 10.1145/3690407.3690517.
[15] D. Halawi, A. Wei, E. Wallace, T. T. Wang, N. Haghtalab, and J. Steinhardt, "Covert Malicious Finetuning: Challenges in Safeguarding LLM adaptation," *arXiv (Cornell University)*, Jun. 2024, doi: 10.48550/arxiv.2406.20053.
[16] J. Sikora, "The influence of language models on decryption of German historical ciphers," *DIVA*, 2022.
[17] "When 'Competency' in reasoning opens the door to vulnerability: jailbreaking LLMs via novel complex ciphers." https://arxiv.org/html/2402.10601v2.
[18]] M. Maxime, "What is a Transformer?," Inside Machine Learning, Medium, Jan. 4, 2019. [Online]. Available: https://medium.com/inside-machine-learning/what-is-a-transformer-d07dd1fbec04

XV. CONTRIBUTION

TABLE VI: Team Members and Their Contributions

Team Member	**Contribution**
Aline Hassan	Finetuning Mistral 7B, Caesar enhanced Training and Vigenere all cases training.
Zeinab Saad	Rail Fence Training, Mono alphebetic all cases training and Caesar 25 shifts training.
Hadi Tfaily	Data preprocessing, zeroshot learning, few shot learning and Caesar fixed shift Training and Caesar mixed shifts training.