Tampere University

Matej Gradoš

ASSESSING CRYPTOGRAPHIC RANDOM NUMBER GENERATORS USING MACHINE LEARNING

Master of Science Thesis
Faculty of Information Technology and Communication Sciences - Tampere University
Faculty of Electrical Engineering and Communication - Brno University of Technology
Examiners: Professor Konstantinos Stefanidis
M.Sc. Sara Ricci, Ph.D.
June 2025

ABSTRACT

Matej Gradoš: Assessing Cryptographic Random Number Generators using Machine Learning
Master of Science Thesis
Tampere University
Double Master's Degree programme in Communications and Networking
June 2025

This thesis investigates the intersection of machine learning and cryptography, with a particular emphasis on the capacity of neural networks to model cryptographic hash functions and evaluation of randomness in binary sequences. The feasibility of training neural networks to replicate the behavior of cryptographic hash functions, with particular focus on a reduced variant of the SHA-3 algorithm. Empirical results indicate that while neural networks can accurately approximate steps of the Keccak-p permutation, they exhibit limited generalization capability across multiple rounds of the Keccak-f function.

In the second phase, the focus shifts to the application of machine learning techniques for the analysis of randomness in binary sequences. Utilizing transformer-based architectures, thesis demonstrates that these models can achieve high predictive accuracy on the final bit of a sequence, including those classified random by conventional statistical test suites such as the NIST SP 800-22. These findings suggest that machine learning models may serve as practical complementary tools to traditional statistical methods, offering a novel approach for uncovering subtle, exploitable patterns that dodge standard randomness assessments.

Keywords: Machine Learning, Neural Networks, Cryptographic Hash Functions, Secure Hash Algorithm 3, SHA-3 Toy Function, Randomness Assessment, Transformer architecture, High entropy

The originality of this thesis has been checked using the Turnitin OriginalityCheck service.

USE OF ARTIFICIAL INTELLIGENCE IN THIS WORK

Artificial intelligence (AI) has been used in generating this work:

☐ No
☒ Yes

I hereby declare, that the AI-based applications used in generating this work are as follows:

Application	Version
ChatGPT	4o

Purpose of the use of AI

AI was used in this thesis for suggestions regarding spelling and phrasing. AI was also used to help with formatting the `.tex` files.

Parts of this work, where AI was used

As stated above, AI tools mentioned were used for formatting of the `.tex` files.

Acknowledgement of risks

I hereby acknowledge, that as the author of this work, I am fully responsible for the contents presented in this thesis. This includes the parts that were generated by an AI, in part or in their entirety. I therefore also acknowledge my responsibility in the case, where use of AI has resulted in ethical guidelines being breached.

PREFACE

I would like to express my sincere gratitude to my supervisors, Christof, Bilal, Gregor, Sara and Kostas, who were the cornerstone of this thesis.

Thank you for your support, guidance, and the countless insightful discussions that consistently left me feeling inspired. Thanks to you, I have grown not only as a researcher but as a thinker, and I fell in love with machine learning, cryptography and security. Fields I will continue to explore with enthusiasm and respect throughout my life.

I also wish to extend my heartfelt thanks to my colleagues from the Embedded Security Group at the Max Planck Institute for Security and Privacy. From day one, you made me feel welcome and truly part of the team. I deeply appreciate our shared laughs, debates, and the many writing tips - not to mention the invaluable advice on navigating my first days during my stay in Germany. Your camaraderie made this journey both intellectually fulfilling and personally meaningful.

To my friends I made along the way across my studies around Europe, thank you!

Lastly, to my dearest family, girlfriend, and friends: thank you for believing in me and supporting my academic journey abroad. Your love and encouragement were the foundation of my perseverance, and I truly would not have come this far without you.

Thank you all.

Bochum, Germany, 6th June 2025

Matej Gradoš

CONTENTS

1. Introduction 1
2. Preliminaries 4
   2.1 Machine Learning 4
      2.1.1 Neural Networks 5
      2.1.2 Transformer model 8
   2.2 Cryptographic Hash Functions 12
      2.2.1 Security Requirements of Hash Functions 13
      2.2.2 The Secure Hash Algorithm SHA-3 14
   2.3 Randomness & Random Number Generators 21
      2.3.1 Security Requirements of Random Number Generators 22
      2.3.2 Linear Congruential Generator 22
      2.3.3 Blum Blum Shub 23
   2.4 NIST Special Publication 800-22. 24
   2.5 Implementation background 25
      2.5.1 Python 25
      2.5.2 PyTorch 25
      2.5.3 SymPy 27
3. Teaching Neural Network the SHA-3 Hash Function 28
   3.1 Implementation of SHA-3 Toy Function 28
   3.2 Dataset preparation 31
   3.3 Neural Network Architecture 32
   3.4 Training and Experimentation Process 33
      3.4.1 Impact of Optimizer Algorithm on Loss 33
      3.4.2 Impact of Input Complexity and Hidden Layer Size on Loss and Accuracy 34
      3.4.3 Model Performance in Training and Generalization Across Multiple Rounds 35
4. Machine Learning Aided Randomness Assessment 41
   4.1 Implementation of pseudorandom binary sequence generators. 41
   4.2 Sequnce-based Randomness Assessment 43
   4.3 Dataset preparation 43
   4.4 Classification Transformer Architecture 44
   4.5 Training and Experimentation Process 45
      4.5.1 Hyperparameter Search and Optimization 46

5. Machine Learning Driven Next-Bit Prediction 48
5.1 Implementation of Linear Congruential Generator 48
5.2 Dataset Preparation 49
5.3 Next-Bit Transformer Architecture 50
5.4 Training and Experimentation Process 53
5.4.1 Hyperparameter Search and Optimization 54
Conclusion 57
References 59

LIST OF FIGURES

2.1 ReLU Activation Function plot 6
2.2 Visualization of a fully connected feedforward neural network 7
2.3 Block diagram of the Transformer Architecture 8
2.4 Multi-Head attention diagram 10
2.5 Signing a long message in block-by-block fashion 12
2.6 Signing a long message using a hash function 13
2.7 Fundamental security requirements of hash functions 14
2.8 Overview of SHA-3 and SHAKE with Keccak algorithm in its core 14
2.9 Keccak sponge construction scheme 15
2.10 Visualization of parts of the state array 17
2.11 π step applied to a single slice 19

3.1 Comparison of heatmaps selected optimizer algorithms on Loss 37
3.2 Heatmap of Loss across Input Complexity and Hidden Layer Sizes 38
3.3 Heatmap of Accuracy across Input Complexity and Hidden Layer Sizes . . 38
3.4 Model Performance in Training and Generalization Across $1-6$ Rounds . 39
3.5 Model Performance in Training and Generalization Across $7-12$ Rounds . 40

4.1 Heatmap of Average Validation Accuracy by Model Architecture 46
4.2 Heatmap of Average Validation Accuracy by Training Parameters 47

5.1 Bit position extraction for binary sequence creation 49
5.2 Heatmap of Prediction accuracy across varying n_{head} and n_{embed} 55
5.3 Train and Test loss curves 56
5.4 ROC curve for next bit prediction 56

LIST OF TABLES

2.1 Parameters of SHA-3 and SHAKE 128 / SHAKE 256 16

2.2 Suffix rules for SHA-3 and SHAKE 128 / SHAKE 256 16

2.3 Valid state sizes with parameters n_r, w and l 17

2.4 Rotation offsets defined for ρ step 18

2.5 Round constants for SHA-3 and SHAKE128/256 in hexadecimal notation . 20

2.6 Relationship between true situation and conclusion in hypothesis testing. . 24

LIST OF SYMBOLS AND ABBREVIATIONS

Adam	Adaptive Moment Estimation
AI	Artificial Intelligence
AND	logical operation, yields true only when the input values are equal
ANN	Artificial Neural Network
AUC	Area Under the Curve
BBS	Blum-Blum-Shub
BCE	Binary Cross Entropy
BERT	Bidirectional Encoder Representations from Transformer
BPE	Byte-Pair Encoding
CNN	Convolutional Neural Network
CPU	Central Processing Unit
CSPRNG	Cryptographically Secure Pseudorandom Number Generator
ECB	Electronic Code Book
ELU	Exponential Linear Unit
FAIR	Facebook's AI Research lab
FNN	Feed-forward Neural Network
GPT	Generative Pre-trained Transformer
GPU	Graphics Processing Unit
KLDivLoss	Kullback-Leibler Divergence Loss
LCG	Linear Congruential Generator
LSB	Least Significant Bit
MB	Megabyte (10^6 bytes)
MCG	Multiplicative Congruent Generator
ML	Machine Learning
MLP	Multi-Layer Perceptron
MSB	Most Significant Bit
MSE	Mean Squared Error

NAdam	Nesterov-accelerated Adaptive Moment Estimation
NIST	National Institute of Standards and Technology
NLLLoss	Negative Log Likelihood Loss
NLP	Natural Language Processing
OOV	Out-Of-Vocabulary
PRNG	Pseudorandom Number Generator
ReLU	Rectified Linear Unit
RL	Reinforcement Learning
RMSprop	Root Mean Squared Propagation
RNN	Recurrent Neural Network
ROC	Receiver Operating Characteristic
RSA	Rivest–Shamir–Adleman - a public-key cryptosystem
SGD	Stochastic Gradient Descent
SHA	Secure Hash Algorithm
SP 800-22	Special Publication 800-22, A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications
Tanh	Hyperbolic Tangent Tanh
TPM	Trusted Platform Module
TRNG	True Random Number Generator
XOF	Extendable Output Function
XOR	(Exclusive OR) logical operation, yields true only when the input values differ

1. INTRODUCTION

Machine learning (ML) has demonstrated remarkable success in modeling complex functions across various domains. Deep neural networks (DNNs) have proven particularly effective at tackling challenging tasks that were once thought to require human-level intelligence. While being widely applied in areas like classification [13] and pattern recognition [49], their potential in cryptographic tasks remains an emerging and promising field of research. Modern cryptography, particularly hash functions like Secure Hash Algorithm 3 (SHA-3), rely on precise and carefully designed mathematical operations to ensure security. Hash functions are fundamental to modern digital security, designed as one-way transformations, in order to make reversing process computationally infeasible. Their outputs are seemingly structureless, that they can be indistinguishable from random noise. Numerous studies have proposed hashing functions based on neural network architectures [38], [82], [39]. These approaches build on neural networks' highly non-linear architecture and ability to model complex bit mappings, which provides an excellent foundation for creating secure, efficient, and collision-resistant cryptographic hash functions.

Beyond hashing, a recent study explored how deep neural networks can be integrated into broader cryptographic functionalities, such as encryption, authentication, and watermarking. However, a key challenge arises from the fundamental difference between traditional cryptographic systems, which rely on discrete Boolean operations, and DNNs, which operate on continuous real-valued inputs [17]. Therefore, the first part of the thesis investigates a novel research area by addressing a fundamental research question:
Can machine learning predict the output of a cryptographic hash function SHA-3?

Rather than developing a more sophisticated hashing method that integrates existing solutions with abstract enhancements, the objective was to train a neural network on fundamental principles and leverage its non-linear characteristics. By focusing on a reduced version of the SHA-3 hash function, this investigation challenges the conventional understanding of cryptographic mechanisms by examining whether neural networks can learn the hashing process. Given SHA-3's cryptographic strength, subsequent experiments with an increased number of rounds of the Keccak-f function revealed an anticipated phenomenon: the neural network was unable to differentiate the hash function's behavior from a truly random process.

This observation unveiled a research gap that motivated a shift from function approximation towards the evaluation of randomness itself. Inspired by studies on machine learning-aided cryptanalysis [19], [23], the latter part of this thesis investigates **whether machine learning models can be trained explicitly for randomness evaluation**.

Randomness plays an important role in simulations across fields such as physics [3] or statistics [24], enabling the modelling of the real world, where deterministic predictions are impractical or impossible [40]. The introduction of Random Number Generators (RNGs) allows to *some extent* produce real-world unpredictability within computers' deterministic nature. These generators generate a seemingly random sequence of numbers using various mathematical formulas and algorithms [47]. While these outputs appear random, they are deterministic and can be reproduced by obtaining a given starting value called *seed*. RNGs are widely used in cryptography, where randomness is a cornerstone of secure systems, aiming to make the system output indistinguishable from random. In literature, several statistical test suites exist to assess whether generators display a random-like behavior [36], [41].

The National Institute of Standards and Technology's (NIST) Special Publication 800-22 (SP 800-22) [27] is one of the most widely known statistical test suites used to check if the generator's output does not deviate significantly from random, neither exhibits detectable patterns. Since random numbers, by definition, are characterized by a lack of predictable patterns, we anticipated that the self-attention mechanism within the Transformer architecture would be highly effective in assessing how a random generator's output truly is. Ongoing debates about the quality and technical soundness of statistical tests assessing the randomness of binary sequences [30] encouraged the development of a machine learning model that could serve as a potential replacement, or at least a complementary tool, within the given statistical test suite.

Building upon results, the core contribution of this randomness assessment framework lies in the development of a specialized transformer model that emerged from the foundational Bidirectional Encoder Representations from Transformer (BERT) architecture [10], tailored for binary sequence randomness evaluation. By adapting the BERT architecture to work with a binary vocabulary and incorporating domain-specific modifications, such as optimized positional encodings and task-specific classification heads, this implementation bridges the gap between advanced natural language processing (NLP) [28] techniques and fundamental cryptographic analysis. The resulting model demonstrates competitive performance against well-established, widely used statistical tests and offer a better pattern recognition that could complement or enhance traditional randomness assessment methodologies.

The transition towards randomness evaluation naturally led to the concept of *next-bit prediction* [92], a foundational idea in cryptography and computation theory. In the context of PRNGs and security, *next-bit prediction* formalizes the idea of output being truly unpredictable. A sequence of bits passes the *next-bit test* if in case any part of the sequence is compromised, an attacker cannot predict the next bit with any real accuracy beyond pure chance, even with access to preceding bits and significant computing power [56].

In the last part of the thesis, we explore **whether sequences considered statistically random by the NIST SP 800-22 statistical tests also exhibit next-bit unpredictability when modeled by a machine learning model**.

Unlike traditional statistical tests examining sequences' statistical properties, next-bit prediction leverages the pattern recognition capabilities of machine learning models. Conventional approaches to binary sequence prediction rely on statistical methods and classical machine learning techniques. However, the problem with these approaches is their inability to cope with dependencies that extend across arbitrarily long sequences. Leveraging findings from the machine learning-aided randomness assessment, we developed `BinaryGPT` model, a specialized decoder-only transformer architecture designed for next-bit prediction in pseudorandom number generators. With an emphasis on analyzing the predictability of binary sequences generated by Linear Congruential Generator (LCG), this model uses causal self-attention to predict the next bit in a binary sequence.

While the NIST SP 800-22 statistical test suite classified sequences as statistically random, `BinaryGPT` model achieved beyond chance prediction accuracy on these same sequences. This demonstrates that statistical randomness alone does not guarantee computational unpredictability. Such findings can have implications for cryptographic applications where PRNGs are employed, as it challenges the assumption that passing statistical tests ensures security against sophisticated prediction attacks.

2. PRELIMINARIES

This chapter introduces machine learning, focusing on neural networks and transformers. It explores cryptographic hash functions, discusses the role of randomness in cryptography, and outlines the technical foundation of this work, including the use of Python programming language, selected libraries, and the PyTorch framework.

2.1 Machine Learning

Machine learning (ML) [45] is a subfield of artificial intelligence (AI) concerned with the question of how to construct computer programs that automatically improve with experience. *Data* is the information provided to a program in order to form its understanding of the world. Similarly, the modification of behavioral patterns based on experience is referred to as *learning* [37].

Currently, people interact with machine learning models on a day-to-day basis. A model in machine learning is like a recipe that the computer learns to follow when making decisions or predictions. Analogous to a chef perfecting a recipe, a machine learning model enhances its predictive capabilities by learning from data. Think of how a spam filter works. In spam filters, the computer is taught to spot junk mail by presenting many examples. Two kinds of emails to study, spam and legitimate emails (sometimes called "ham"), are shown to our model. The model learns from these examples. This collection of emails, both spam and ham, is called a *training set*. Each email inside this set represents a new lesson, also called a *sample* or *training instance*. Computer's task is to identify the spam emails in emails never presented to it. The computer undergoes a learning process by iterating throughout the training set. Its performance and improvement are assessed by measuring its *accuracy*.

This approach falls under supervised learning, where the model learns from labeled data to make predictions [89]. However, not all machine learning tasks deal with labeled data. In such cases, models are based on unsupervised learning, which involves finding patterns or structures in unlabeled data [90], [18]. In unsupervised learning, the model identifies patterns or clusters in data, such as categorizing emails into promotional, personal, or spam, without predefined or preexisting labels. Such capabilities are often powered by neural networks, a fundamental concept in machine learning and artificial intelligence.

Neural networks are particularly effective for problems that traditional algorithms find difficult to solve. Reinforcement learning (RL) [31] represents the best of both worlds. It is a unique approach in machine learning, where an *agent* learns how to make decisions by interacting with an environment. Unlike supervised and unsupervised learning, which rely on learning from labeled or unlabeled data, reinforcement learning learns through trial and error.

2.1.1 Neural Networks

Neural networks [84] are computational models inspired by the human brain's architecture. They consist of interconnected groups of nodes, called *neurons* or artificial neurons, which are fundamental units of the brain and nervous system, the cells responsible for receiving sensory input from the external world [86]. It learns and stores information by adjusting the connections, called *weights*, between the neurons. Design of a neural network, including the number of neurons, layers, and their interconnections, defines its potential and usability.

A *neuron* [20] is the fundamental unit of a neural network. It functions as a computational node, accepting one or more numerical inputs, processing them through a mathematical operation, and producing a single numerical output. Functionality of a single neuron in a neural network can be expressed by the following equation

$$y = f\left(\sum x_i w_i + b\right) \tag{2.1}$$

In equation 2.1, f represents the activation function that determines the neuron's output. Each neuron takes multiple inputs x_i, which are individually scaled by corresponding weights w_i to adjust their significance. The bias term b, serves as a threshold that offsets the weighted sum, enabling the neuron to adjust its response threshold and better adapt to the underlying data distribution and network architecture [20].

Weights are essential numerical values that determine the strength and impact of connections between neurons across layers. These weights hold an important role in the learning process of Artificial Neural Networks (ANNs). Each weight is assigned to a specific connection between two neurons, quantifying the influence of one neuron's output on another's input. As input signals propagate through the network, they are multiplied by the associated weights. This weighted summation of inputs at each neuron determines the final output of the network [20].

Bias, referred to as the *threshold*, is a constant value added to the weighted summation of the inputs to adjust the activation function's output. This adjustment enables the neuron to generate outputs for inputs that may not otherwise trigger the activation function.

In practice, the bias term is usually treated as an additional weight associated with a constant input of "1", allowing the model to dynamically modify the threshold throughout the training process [84], [20].

Activation function

Activation function is a crucial mathematical tool that transforms the neuron's input into its output. By introducing non-linear transformations, these functions allow neural networks to model complex relationships that go beyond simple linear connections. Function's differentiability is particularly important, as it enables the network to learn through a process called *backpropagation* [88], a key training mechanism that adjusts the network's internal parameters. Activation functions can be categorized into two main categories.

Linear activation functions output the weighted sum directly without any additional transformation. Although being simple, they do not introduce nonlinearity, which limits the network's ability to solve complex problems. Such limitation results in the model remaining linear regardless of the network's depth. Contrary to linear activation functions, **nonlinear** activation functions are more commonly used thanks to their ability to introduce complexity into the model. Some of the popular nonlinear activation functions include Rectified Linear Unit (ReLU), Leaky ReLU, Sigmoid, Hyperbolic Tangent Tanh (Tanh), Softmax, Exponential Linear Unit (ELU), or Softplus [20].

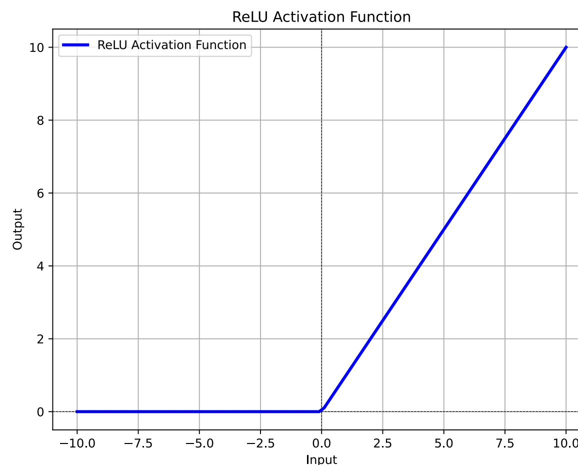

Figure 2.1. *ReLU Activation Function plot*

ReLU function is a simple, yet powerful activation function denoted as follows

$$\text{ReLU}(x) = x^+ = \max(0, x) = \frac{x + |x|}{2} = \begin{cases} x & \text{if } x > 0, \\ 0 & \text{if } x \leq 0 \end{cases}$$

In mathematical terms, the ReLU function computes the maximum between 0 and x, where x is any given input value. ReLU function is represented graphically by a ramp function, as visualized in Figure 2.1. It maintains a constant output of zero for negative inputs and exhibits a linear increase for positive inputs [12]. This activation function is recommended for use with most feedforward neural network architectures [20].

Layers

Individual layers form the structural backbone of neural networks, organizing neurons into groups that transform input data. Each layer within the neural network represents a specific stage of computational processing, in which neurons employ weights, biases, and activation functions to transform input signals [20]. A neural network generally consists of three fundamental layer types:

- **Input layer**, which receives raw data,
- **Hidden layers**, where the primary computational transformations occur,
- **Output layer**, which produces the final network prediction.

Organization of these layers and connections between them determine the type of neural network and its functionality. For example, fully connected layers, where each neuron is connected to every neuron in the subsequent layer, characterize a Feedforward Neural Network (FNN). In contrast, networks with specialized layer structures, such as *convolutional layers* in a Convolutional Neural Network (CNN). Similarly, recurrent layers with feedback connections and cycles define Recurrent Neural Networks (RNNs). Thus, the choice of layer types and their connectivity enables neural networks to be tailored to specific tasks and data processing [84].

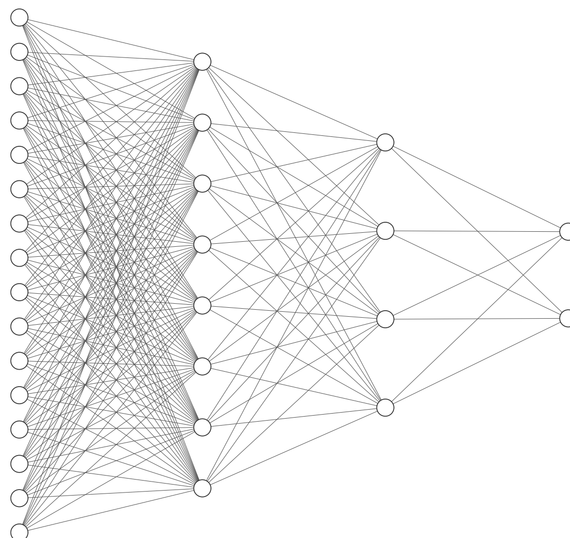

Figure 2.2. *Visualization of a fully connected feedforward neural network*

In feedforward neural networks information flows from input layers, through one or more hidden layers, directly to the output layer. Each layer in FNN processes the output of the previous layer, without cycles or feedback loops, as visualized in Figure 2.2. Each neuron in hidden layers computes a weighted sum of its inputs, applies a bias, and passes the result through an activation function to introduce nonlinearity [84]. FNNs are also less demanding in terms of computational resources compared to other architecture types.

2.1.2 Transformer model

First proposed in the 2017 paper titled: *Attention Is All You Need* [85], by researchers from Google DeepMind [21], former Google Brain. Transformer is a deep learning architecture based on the multi-head attention mechanism. Ever since, this architecture has revolutionized the field of machine learning and Natural Language Processing (NLP) [28]. Thanks to the self-attention mechanism and the absence of recurrent and convolutional components, Transformers have redefined what is possible across number of applications, including machine translation [85] or text summarization [87].

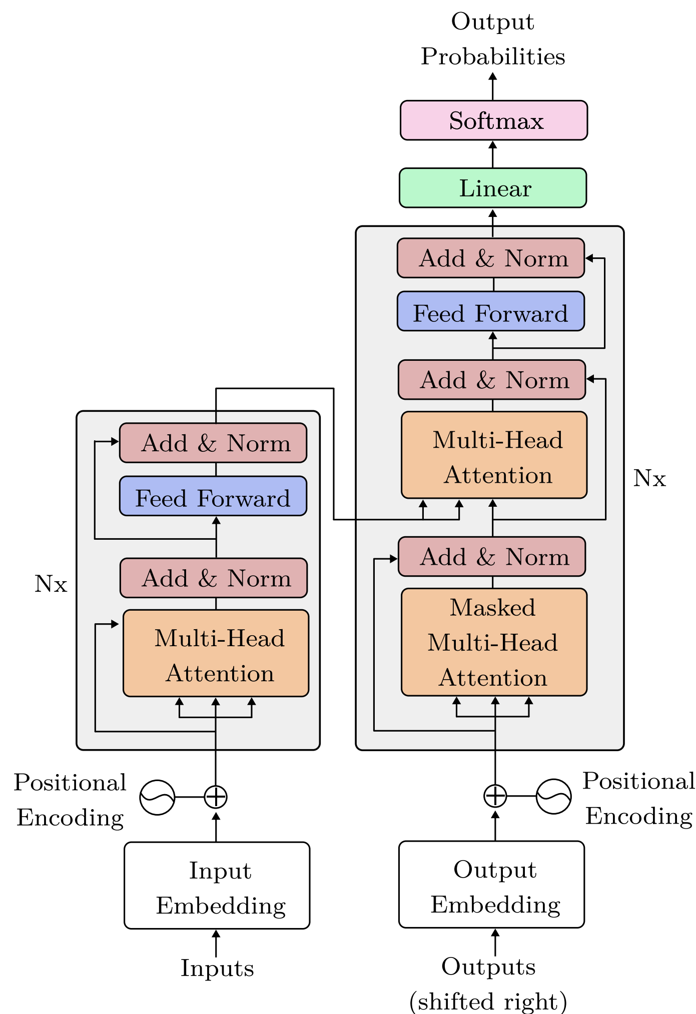

Figure 2.3. *Block diagram of the Transformer Architecture*

Novel architecture, pictured in Figure 2.3, presents notable advantages over prior traditional machine learning architectures, including recurrent and convolutional network designs. Firstly, it supports parallel processing of input data. Unlike RNNs, which process data sequentially, this approach allows for parallel processing, which significantly speeds up the training process. Secondly, by incorporating self-attention mechanisms, this architecture can capture relationships between distant elements within sequences significantly better than RNNs employed in classical scenarios [85]. Furthermore, the underlying structure demonstrates effective scaling capabilities, which enable the development of parameter-rich models such as Bidirectional Encoder Representations from Transformer (BERT) [10] and Generative Pre-trained Transformer (GPT) [93] families.

Transformer models employ an encoder-decoder structure where the encoder processes input sequences to create continuous representations that the decoder transforms into outputs. The self-attention mechanism makes the architecture stand out, which enables the model to evaluate word relevance regardless of positional constraints effectively. It takes a query and a set of key-value pairs as input, ultimately producing an output vector. The result is calculated as a weighted sum of vectors, where a compatibility function of the query with the corresponding key computes the weight assigned to each value. Calculation is performed on a set of queries, packed together into a matrix Q. Formally, this process can be noted as

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

where Q corresponds to queries, K to keys, and V to values.

Multi-head attention is another key structural component of the Transformer architecture, with its structure shown in Figure 2.4. It allows the model to process different aspects of the input at the same time, which helps it understand various connections within the input sequences. Instead of using one attention function with full d_{model}-dimensional keys, this architecture takes queries, keys, and values and transforms them multiple times using different learned projections. This maps them into smaller, more manageable d_k and d_v dimensional spaces. Each set of values is then processed through an attention function in parallel, producing outputs concatenated and projected once more to obtain the final output. This allows the model to attend to information from different representation subspaces across various positions while effectively capturing diverse data features. Thanks to the multi-head attention mechanism, the model can effectively model and find complex patterns and long-range dependencies in data compared to single-head attention mechanisms.

In mathematical terms, this can be expressed as:

$$MultiHead(Q, K, V) = Concat(head_1, head_2, ...head_n)W^O,$$

where $head_n = QW_n^Q, KW_n^K, VW_n^V$.

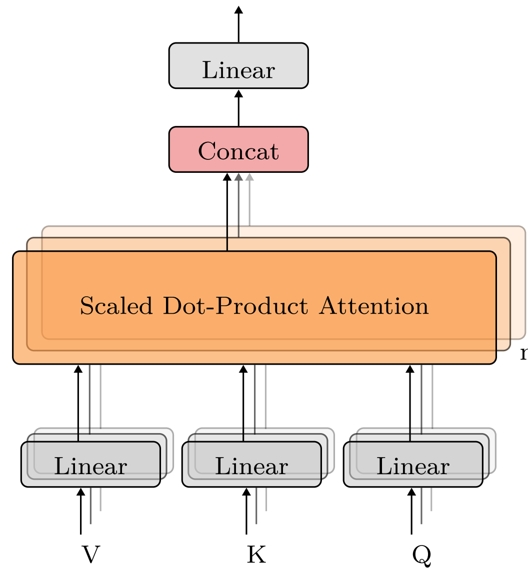

Figure 2.4. *Multi-Head attention diagram*

As a cover-up for architecture's lack of recurrent or convolutional elements, position information must be explicitly incorporated to preserve the sequence's original ordering. This is achieved by adding positional encodings directly to the input embeddings at the base of both encoder and decoder components. Positional embeddings are designed with the same dimensionality as the embeddings, allowing a seamless combination through addition and ensuring the model can distinguish token positions within sequences [85].

Tokenization

To process raw input data effectively, the architecture requires the input to be converted into an understandable format. This is accomplished through a process known as *tokenization*, which transforms sequences such as text or binary streams into discrete, model-readable units called *tokens*. Unlike traditional NLP approaches that often rely on word-level tokenization [34], [58], modern transformer architectures typically employ subword tokenization strategies [58] to balance vocabulary size with representation flexibility. The primary challenges that tokenization addresses include handling Out-Of-Vocabulary (OOV) words, maintaining semantic meaning across different languages, and efficiently representing text with a finite vocabulary [44].

Word-based tokenization [58] splits text at word boundaries, typically using spaces and punctuation as separators. This approach creates a *one-to-one* mapping between words and tokens, which is both intuitive and preserves word meaning. However, vocabulary size can become unmanageably large as it must include every possible word, leading to increased model size and computational memory requirements.

On the other hand, **character-based** tokenization represents text as individual characters, creating a much smaller vocabulary than word-based approaches. While character tokenization creates extremely long sequences since each character becomes a separate token, this approach increases computational demands and makes it more challenging to capture long-range dependencies.

Subword tokenization [2] is the best of both worlds between word and character approaches by breaking words into meaningful subunits. Commonly used algorithms for subword-based tokenization include Byte-Pair Encoding (BPE) [6], WordPiece [91], Unigram [83], or BPE-Dropout [52], [51]. BPE begins by treating each character as a separate unit and then repeatedly combines the most common neighboring pairs to create a vocabulary of frequently occurring word parts. WordPiece improves upon BPE by using the likelihood of combinations in the training data to guide its merging process, thus favoring subwords that best explain the data. Unigram begins with a large vocabulary and then strategically removes tokens to fit the training data best, meaning a single text can be broken down into tokens in more than one correct way. Compared to standalone BPE, BPE-Dropout introduces randomness during tokenization to improve robustness.

2.2 Cryptographic Hash Functions

A hash function is a fundamental component in today's computer science and cryptography. Hash functions are components for many important information security applications, including the generation and verification of digital signatures, key derivation, and pseudorandom bit generation [16]. Data provided to a hash function as an input is referred to as the *message*, and the resulting output is referred to as the *(message) digest* or *hash value*. These computed hash values can act as a unique fingerprint for the message. Unlike other cryptographic algorithms, hash functions do not use, nor rely on a key.
The drive to develop efficient hash algorithms came from the need to compute digital signatures, like RSA [14], more efficiently. For example, when computing digital signatures of large messages, an intuitive way would be to divide the message into smaller blocks and sign each block individually, similar to Electronic Code Book (ECB) [9] mode used in block ciphers, shown in Figure 2.5.

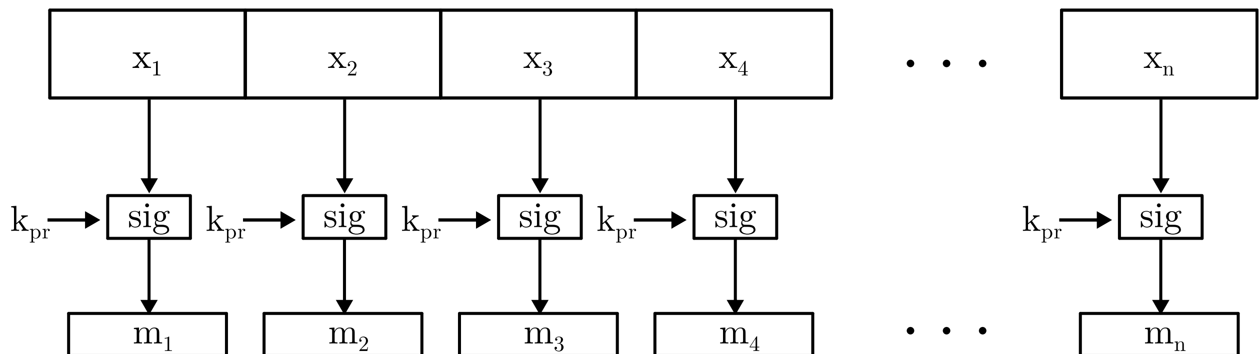

Figure 2.5. *Signing a long message in block-by-block fashion*

Although this approach seems like a reasonable way to handle the operation, it introduces several challenges. Firstly, digital signatures rely on computationally intensive operations, making signing and verification of large messages time-consuming and energy-intensive. Additionally, the time and resources spent signing large messages are mirrored in the verification process. Secondly, the intuitive approach of dividing the message into smaller blocks and signing each block individually dramatically increases the amount of data that needs to be transmitted. Not only must the actual message be sent, but also many signatures, each roughly the same size as the message itself. Signing long messages block-by-block is problematic and introduces additional vulnerabilities, such as removal or reordering individual blocks and their corresponding signatures, or even creating new messages by combining fragments of original messages and signatures. While manipulations within a single block might be prevented, the overall message integrity is compromised. Hash functions provide a suitable solution to the mentioned challenges. By computing a unique fingerprint for a message of any length, we can sign this hash instead of all the individual blocks of the message, as depicted in Figure 2.6.

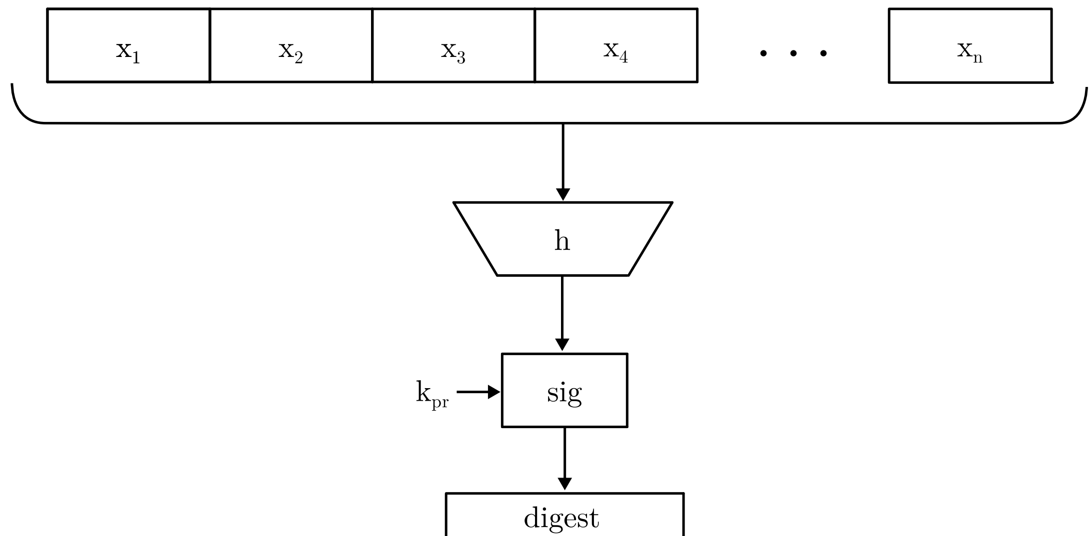

Figure 2.6. *Signing a long message using a hash function*

Mathematically, a hash function h can be defined as a mapping from an input of arbitrary bit length to a fixed-size output of n bits. The output length n, of most modern hash functions range from 256 to 512 bits. Each output bit can be either 0 or 1. To represent the arbitrary input length, asterisk "$*$" is used:

$$h : \{0,1\}^* \to \{0,1\}^n$$

2.2.1 Security Requirements of Hash Functions

Hash functions are highly sensitive to all input bits. That means every little tempering of the input message, yields a completely new message digest. This characteristic serves as a stepping stone for several important security properties of hash functions, including *preimage resistance*, *second preimage resistance*, and *collision resistance*. Understanding how these properties interrelate is vital for assessing the overall security of cryptographic hash functions.

Preimage resistance or one-wayness, is defined as the difficulty of finding any input that corresponds to a specific output of the given hash function. Formally, given a hash value z, it must be computationally infeasible to find an input message m, such that $z = h(m)$. If the hash function lacks one-wayness, an attacker could derive the message m by computing the inverse of the hash function $h^{-1}(z) = m$.

Second preimage resistance or weak collision resistance, refers to fact that an attacker should not be able to find two different messages $m_1 \neq m_2$, that produce the same hash value $z_1 = h(m_1) = h(m_2) = z_2$. This property prevents an attacker from easily substituting one valid message for another without altering the hash.

Figure 2.7. *Fundamental security requirements of hash functions*

Lastly, a hash function is considered **collision-resistant**, if it is difficult to find two different inputs $m_1 \neq m_2$, that produce the same hash output $h(m_1) = h(m_2)$. This is a more challenging property to achieve than weak collision resistance, as an attacker can manipulate both inputs to create a collision. An attacker modifies the messages m_1 and m_2, without changing their overall meaning. He keeps doing this until he finds two different versions of the message that have the same hash value.

2.2.2 The Secure Hash Algorithm SHA-3

Secure Hash Algorithm 3 (SHA-3) is a cryptographic hash function standardized by the National Institute of Standards and Technology (NIST) in 2015, as **FIPS 202** [16]. It is aimed to serve as an alternative to SHA-2 [15], but not as a replacement - rather a parallel option in case a vulnerability is ever discovered in SHA-2. SHA-3, together with SHAKE128, and SHAKE256 are based on the Keccak algorithm. Similarly to traditional hash functions, SHA-3 produces a fixed-size output.

However, SHAKE128 and SHAKE256 differ by offering the ability to compute an output bit string of arbitrary length, making them an *eXtendable-Output Functions* (XOFs). The Keccak algorithm can be customized using adjustable parameters, allowing it to be used for both SHA-3 and SHAKE, as shown in Figure 2.8.

Figure 2.8. *Overview of SHA-3 and SHAKE with Keccak algorithm in its core*

Unlike the Merkle-Damgård structure [25], used in the earlier SHA-1 and SHA-2 hash functions, SHA-3 relies on the unique sponge construction of Keccak. This new design provides greater adaptability, robustness, and protection against cryptographic attacks. After preprocessing the message, which divides the message into blocks and provides padding, comes a sponge construction that includes two phases:

1. **Absorbing Phase** processes the message blocks x_i.
2. **Squeezing Phase** calculates and produces an output of adjustable length.

Initially, the input undergoes a preprocessing step, involving the addition of a suffix and padding. Subsequently, in the absorbing phase, input blocks x_i are processed sequentially. The squeezing phase then generates output blocks y_j as needed.

At the core of SHA-3 and SHAKE128/256 lies the *Keccak-f* function. This function can be configured with following parameters:

- **State width (b)** determines the size of the internal state
- **Bit rate (r)** specifies length of input and output blocks
- **Capacity (c)** parameter influences the security level of the cryptographic operation

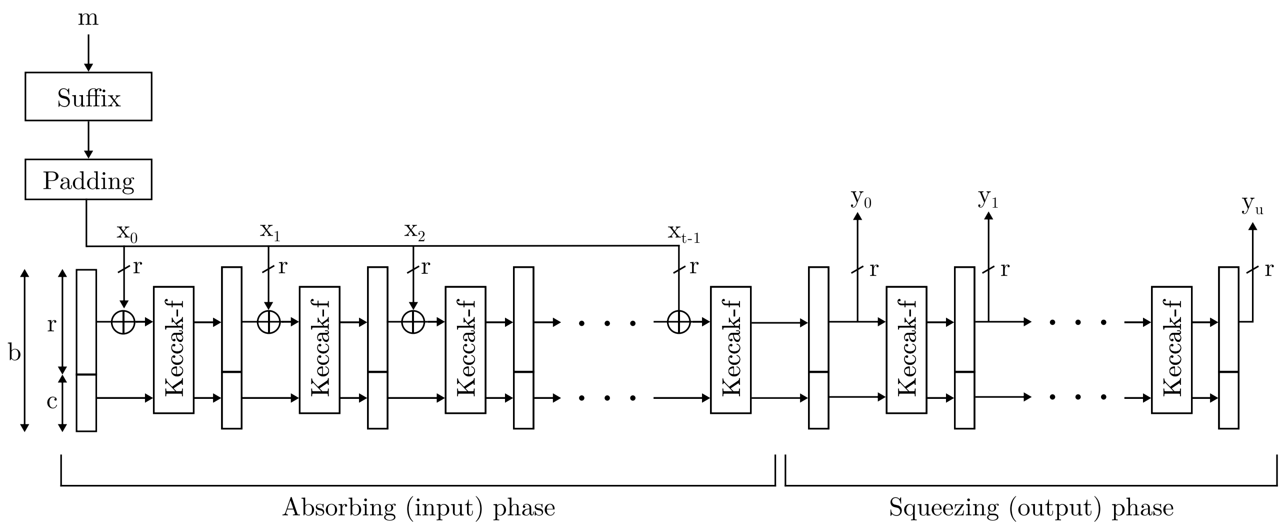

Figure 2.9. *Keccak sponge construction scheme*

For SHA-3 and SHAKE128/256 the state is fixed at $b = 1600$. Different combinations of r and c lead to different security levels, as presented in Table 2.1. Security level denotes the number of computations an attacker has to perform in order to find a collision. Before processing the input message, SHA-3 and SHAKE128/256 add suffixes and padding to the message. Steps regarding the preprocessing are not part of the *Keccak* algorithm [4]. Suffix bits are added to the message before the padding itself. Rules for suffixes used in SHA-3 and SHAKE 128/256 are outlined in Table 2.2.

	Function type	b (state) [bits]	r (rate) [bits]	c (capacity) [bits]	Security level [bits]	Hash output length [bits]
SHA3-224	hash	1600	1152	448	112	224
SHA3-256	hash	1600	1088	512	128	256
SHA3-384	hash	1600	832	768	192	384
SHA3-512	hash	1600	576	1024	256	512
SHAKE128	XOF	1600	1344	256	128	arbitrary
SHAKE256	XOF	1600	1088	512	256	arbitrary

Table 2.1. *Parameters of SHA-3 and SHAKE 128 / SHAKE 256*

These suffixes ensure that even if the same message is used for calculating a SHA-3 hash or generating a pseudorandom sequence with SHAKE128/256, the resulting bits are unique. This process is also referred to as domain separation [16], [47].

	suffix
SHA-3	suf = 01
SHAKE 128 / SHAKE 256	suf = 1111

Table 2.2. *Suffix rules for SHA-3 and SHAKE 128 / SHAKE 256*

The message and suffix are extended with padding bits to reach length that is divisible by r. This is required, since Keccak algorithm processes the input message in blocks of r bits. The padding, referred to as *multi-rate* padding [16], is a sequence of bits that starts with a "1", followed by a certain number of "0"s, and ends with "1", as represented mathematically by the equation below

$$\text{pad}(m, \text{suf}) = m||\text{suf}||10^*1$$

Keccak-f in literature is often referred to as the *Keccak-f permutation* due to its bijective nature, mapping each b-bit input to a unique b-bit output. A *Keccak-p* permutation round consists of five consecutive steps **Theta** (θ), **Rho** (ρ), **Pi** (π), **Chi** (χ), **Iota** (ι), which manipulate a state array initialized with the input data. The *Keccak-p* permutation operates on a $b-$bit state array, where $b = 5 \times 5 \times w$ bits, see Figure 2.10. In NIST's **FIPS 202** standard, two additional values related to b are defined

$$w = \frac{b}{25} \qquad\qquad l = \log_2\left(\frac{b}{25}\right)$$

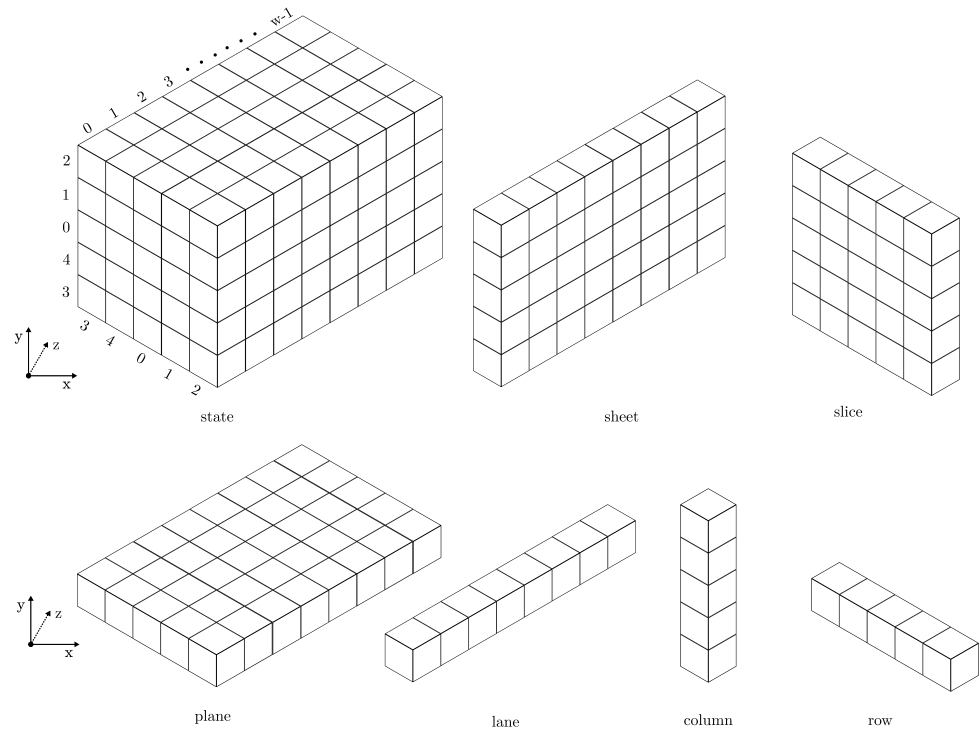

Figure 2.10. *Visualization of parts of the state array*

Table 2.3 lists the seven valid state sizes and their corresponding rounds n_r.

State size b [bits]	25	50	100	200	400	800	**1600**
Number of rounds n_r	12	14	16	18	20	22	**24**
Lane w [bits]	1	2	4	8	16	32	**64**
l	0	1	2	3	4	5	6

Table 2.3. *Valid state sizes with parameters n_r, w and l*

Theta Step

θ step is a bitwise operation that modifies each bit in the state. Each bit in the state is replaced by the XOR sum of itself and ten neighboring bits. This can be visualized as adding bits from left and right columns, wrapping around the 5×5 grid, formally outlined in Algorithm 2.1.

Algorithm 2.1 Algorithm of Theta step θ calculation

Input: 3-D array $A[x, y, z]$ of size $5 \times 5 \times w$
Output: Modified array A'

Step 1: For all pairs (x, z) such that $0 \leq x < 5$ and $0 \leq z < w$, let

$$C[x, z] = A[x, 0, z] \oplus A[x, 1, z] \oplus A[x, 2, z] \oplus A[x, 3, z] \oplus A[x, 4, z]$$

Step 2: For all pairs (x, z) such that $0 \leq x < 5$ and $0 \leq z < w$, let

$$D[x, z] = C[(x - 1) \bmod 5, z] \oplus C[(x + 1) \bmod 5, (z - 1) \bmod w]$$

Step 3: For all triples (x, y, z) such that $0 \leq x < 5$, $0 \leq y < 5$, and $0 \leq z < w$, let

$$A'[x, y, z] = A[x, y, z] \oplus D[x, z]$$

Rho Step

ρ step shifts the bits within each lane. The amount of shift, or offset, is determined by the lane's fixed x and y coordinates. This shift can be thought of as modifying the z-coordinate of each bit by adding the offset, while keeping the x and y coordinates unchanged. This modification is done modulo the lane size w. Formally presented in Algorithm 2.2.

	x = 3	**x = 4**	**x = 0**	**x = 1**	**x = 2**
y = 2	153	231	3	10	171
y = 1	55	276	36	300	6
y = 0	28	91	0	1	190
y = 4	120	778	210	66	253
y = 3	21	136	105	45	15

Table 2.4. *Rotation offsets defined for ρ step*

Algorithm 2.2 Algorithm of Rho step ρ calculation

Input: 3D array $A[x, y, z]$ with dimensions $5 \times 5 \times w$
Output: Modified 3D array $A'[x, y, z]$

Step 1: For all z such that $0 \leq z < w$, let

$$A'[0, 0, z] = A[0, 0, z]$$

Step 2: Let $(x, y) = (1, 0)$.

Step 3: For t from 0 to 23:
(a) For all z such that $0 \leq z < w$, let

$$A'[x, y, z] = A[x, y, (z - \tfrac{(t+1)(t+2)}{2}) \bmod w]$$

(b) Let $(x, y) = (y, (2x + 3y) \bmod 5)$.
Step 4: Return A'.

Pi Step

The π step rearranges the lanes within a slice. This rearrangement follows a specific pattern, as shown in Figure 2.11, formally described in Algorithm 2.3.

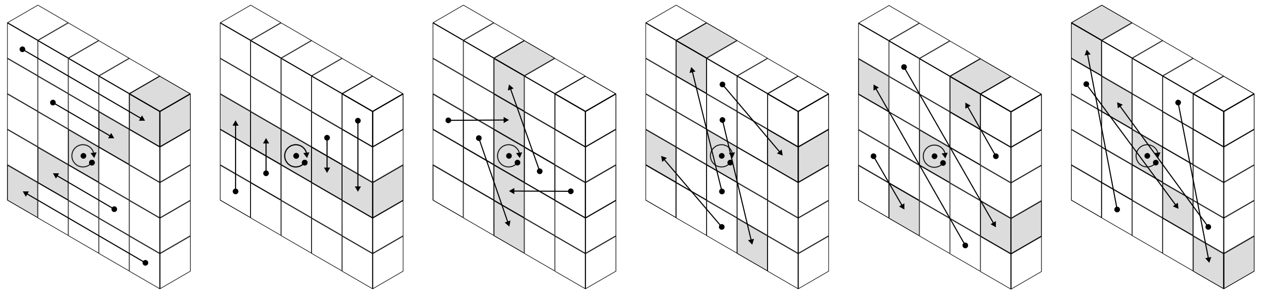

Figure 2.11. *π step applied to a single slice*

Algorithm 2.3 Algorithm of Pi step π calculation

Input: 3D array $A[x, y, z]$ with dimensions $5 \times 5 \times w$
Output: Modified 3D array $A'[x, y, z]$

Step 1: For all triples (x, y, z) such that $0 \leq x < 5$, $0 \leq y < 5$, and $0 \leq z < w$, let

$$A'[x, y, z] = A[(x + 3y) \bmod 5, x, z]$$

Step 2: Return A'.

Chi Step

The χ step is the only a nonlinear step in the *Keccak*-p. Modifies each lane of the state by XORing it with the logical AND of the complement of the next lane, and the lane two steps ahead, with all indices taken modulo 5, as shown in Algorithm 2.4.

Algorithm 2.4 Algorithm of Chi step χ calculation

Input: 3D array $A[x, y, z]$ with dimensions $5 \times 5 \times w$
Output: Modified 3D array $A'[x, y, z]$

Step 1: For all triples (x, y, z) such that $0 \leq x < 5$, $0 \leq y < 5$, and $0 \leq z < w$, let

$$A'[x, y, z] = A[x, y, z] \oplus ((A[(x+1) \bmod 5, y, z] \oplus 1) \cdot A[(x+2) \bmod 5, y, z])$$

Step 2: Return A'.

Iota Step

The ι step involves adding w-bit constant to the lane at index $[0, 0]$ within the state array. The round constant RC[i] is dynamically determined by the current round index i, outlined in the Algorithm 2.5. The corresponding round constants RC[0] through RC[23] can be found in Table 2.5.

RC[0] = 0x0000000000000001	RC[12] = 0x000000008000808B
RC[1] = 0x0000000000008082	RC[13] = 0x800000000000008B
RC[2] = 0x800000000000808A	RC[14] = 0x8000000000008089
RC[3] = 0x8000000080008000	RC[15] = 0x8000000000008003
RC[4] = 0x000000000000808B	RC[16] = 0x8000000000008002
RC[5] = 0x0000000080000001	RC[17] = 0x8000000000000080
RC[6] = 0x8000000080008081	RC[18] = 0x000000000000800A
RC[7] = 0x8000000000008009	RC[19] = 0x800000008000000A
RC[8] = 0x000000000000008A	RC[20] = 0x8000000080008081
RC[9] = 0x0000000000000088	RC[21] = 0x8000000000008080
RC[10] = 0x0000000080008009	RC[22] = 0x0000000080000001
RC[11] = 0x000000008000000A	RC[23] = 0x8000000080008008

Table 2.5. *Round constants for SHA-3 and SHAKE128/256 in hexadecimal notation*

Algorithm 2.5 Algorithm of Iota step ι calculation

Input: 3D array $A[x, y, z]$ with dimensions $5 \times 5 \times w$; round index i_r
Output: Modified 3D array $A'[x, y, z]$

Step 1: For all triples (x, y, z) such that $0 \leq x < 5$, $0 \leq y < 5$, and $0 \leq z < w$, let
$$A'[x, y, z] = A[x, y, z]$$

Step 2: Let $RC = 0^w$.

Step 3: For j from 0 to l, let
$$RC[2j - 1] = rc(j + 7i_r)$$

Step 4: For all z such that $0 \leq z < w$, let

$$A'[0, 0, z] = A'[0, 0, z] \oplus RC[z]$$

Step 5: Return A'.

2.3 Randomness & Random Number Generators

"Randomness in cryptography is like the air we breathe. You can't do anything without it." [11] It serves as a cornerstone of cryptographic systems, providing the unpredictability necessary for the generation of cryptographic keys, which are then to be distributed among participants, or generation of *nonces* (numbers only used once) in protocols, to ensure the freshness of communication, further underscoring the critical role of randomness in maintaining security throughout cryptographic operations [47]. Randomness refers to the quality or state of lacking any defined pattern, order, or lack of predictability in events or information [7], [55]. In simple terms, something is considered random if it happens purely by chance, without specific reason or pre-determined plan. For real world simulations, Random Number Generators (RNGs) were implemented to produce sequences of numbers that approximate the properties of truly random sequences. These generators fall into three main categories:

True Random Number Generators (TRNGs) [43] produce output that is impossible to predict or reproduce. Similar to flipping a coin 100 times. No one could ever recreate the exact 100-bit sequence. TRNGs rely on unpredictable physical processes, like human coin flips or dice rolls. In today's computers, modern Central Processing Units (CPUs) often have hardware-based TRNGs built-in, or a Trusted Platform Module (TPM) on the motherboard might contain one.

Pseudorandom Number Generators (PRNGs) [43], [47] generate sequence by calculating an initial *seed* value. They're often generated step-by-step, with each new element depending on the previous one. Their structure can be expressed with a following mathematical expression:

$$\begin{aligned} s_0 &= \ seed \\ s_{i+1} &= f(s_i), \quad i = 0, 1, 2, 3, \ldots \end{aligned}$$

Lastly, **Cryptographically Secure Pseudorandom Number Generators** (CSPRNGs) [43], [47], are a special kind of PRNGs that have an additional quality, that distinguishes them from other RNGs. Their output is *unpredictable*. By definition, for any given sequence of n consecutive bits from the key stream, no polynomial-time algorithm can predict the value of the succeeding bit, s_{n+1}, with a probability of success significantly greater than 0.5. Another property is the computational infeasibility of predicting any preceding bits given a subsequent output sequence.

2.3.1 Security Requirements of Random Number Generators

A toss of a fair and balanced coin illustrates true randomness. Each flip represents an independent event with exactly two equally probable outcomes, with a 50% probability. The outcome of any single toss cannot be predicted with certainty, regardless of the results of previous tosses. This independence means that even after observing ten consecutive heads, the probability of the next toss being heads remains 50% [22]. The coin toss example also demonstrates core characteristics of randomness required in random number generators used in cryptography: each flip is **unpredictable**, results are **uniformly distributed**, and every toss is **independent of the others**. Though one toss offers only a single bit of randomness, many can be combined for longer random sequences. Furthermore, the internal state of the RNG must maintain its **secrecy**. If an attacker learns the state, they could predict future outputs and potentially break the system's security.

2.3.2 Linear Congruential Generator

Linear Congruential Generator (LCG) [35] is one of the oldest, simplest, and most used types of pseudorandom number generators. The following discontinuous piecewise linear equation calculates generation of a pseudorandom sequence:

$$x_{n+1} = (a \cdot x_n + c) \mod p \tag{2.2}$$

In equation 2.2, x_{n+1} refers to the output value (or the next state), parameter a is the multiplier, x_n marks to the current state (or seed), parameter c in literature is referred to as an increment, and modulus p. Variations in these parameter choices lead to distinct LCG configurations. Notably, when the increment c is set to zero ($c = 0$), the generator simplifies to a Multiplicative Congruential Generator (MCG), sometimes also referred to as a Lehmer generator [50]. Conversely, if the increment c is non-zero ($c \neq 0$), the LCG is specifically termed a mixed congruential generator.

Careful selection of parameters is crucial for achieving optimal statistical properties, such as a long *period* and uniform distribution, within the generated pseudorandom sequences. The Hull–Dobell theorem [26] guarantees the maximum period $T_m(a,\ c)\ =\ p$, in an LCG generator defined by equation 2.2, if and only if:

- c and p are coprime,
- $a - 1$ is divisible by all prime factors of p,
- $a - 1$ is also divisible by 4 if p is divisible by 4.

It is important to mention, this property alone is not a sufficient criterion for evaluating the overall quality of the generator nor ensuring the production of high-quality pseudorandom output [48].

2.3.3 Blum Blum Shub

Based on concepts from number theory, the Blum-Blum-Shub (BBS) generator [5] is a significant improvement in producing cryptographically strong pseudorandom numbers. Unlike simpler generators, its security comes from the computational difficulty of a number theory problem called *quadratic residuosity* [32], which is connected to how hard it is to factor integers. The BBS generator works through a very compact mathematical formula:

$$x_{n+1} = (x_n)^2 \mod n \tag{2.3}$$

To ensure cryptographic strength, the modulus n is the product of two large prime numbers, p and q, congruent to $3\ mod\ 4$. This specific choice ensures that every quadratic residue modulo n has a unique square root, essential for unpredictability. While calculating the next value x_{n+1} using equation 2.3 is relatively simple, the reverse operation and determining the previous value x_n based on x_{n+1} is computationally infeasible. This holds unless prime factors of n are known, making the generator's underlying mechanism also a *one-way function*.

2.4 NIST Special Publication 800-22

The widely recognized NIST Special Publication 800-22 (**SP 800-22**) [27] is a suite of statistical tests used to assess the randomness of binary sequences. These tests are primarily applied to outputs from true random number generators and pseudorandom number generators, especially when intended to be used for cryptographic applications. First published in October 2000 and updated in 2010, SP 800-22's 15 statistical tests serve as a crucial benchmark for evaluating the quality of random number generators. They help to confirm that generated binary sequences exhibit the expected statistical characteristics of randomness, ensuring their suitability for security-critical use cases. Ongoing discussions aim to further refine and clarify the publication's objectives and address technical issues [30].

The suite verifies statistical properties by detecting any deviations from random behavior, ensuring that the generator's output is indistinguishable from random noise. Each of the 15 different statistical tests evaluates the null hypothesis H_0, that a sequence is random, using various mathematical methods to look for specific pre-defined patterns. As shown in Table 2.6, statistical hypothesis testing involves making a judgment regarding the randomness of data. This decision results in one of two outcomes. Either the null hypothesis (H_0) is accepted, meaning the data is random, or the alternative hypothesis (H_a) is accepted, signifying that the data is considered non-random.

True situation	**Accept** H_0	**Accept** H_a (reject H_0)
Data is random (H_0 is true)	No error	Type I error
Data is not random (H_a is true)	Type II error	No error

Table 2.6. *Relationship between true situation and conclusion in hypothesis testing.*

Deviations examined include *imbalance*, *pattern repetition*, *compressibility*, or *spectral structure*. A key objective of these tests is to reduce the chance of a Type II error. This means minimizing the likelihood of incorrectly determining that a sequence is random when it was actually generated by a flawed (non-random) source.

Tests together offer a broad but not exhaustive measure of a sequence's statistical randomness. All tests require a binary sequence of a specific minimum length as input. Each analyzed sequence produces a p-value, which determines whether the sequence passes or fails a given test. A test is passed if the p-value exceeds the predetermined significance level α, where $\alpha = 0.01$. If the p-value is lower than this threshold, the sequence is considered non-random.

Randomness assessment is based on three foundational assumptions regarding the nature of random binary sequences. **Uniformity** assumes that at any point in a random sequence, the occurrence of 0 or 1 is equally probable, with an expected balance across long sequences. **Scalability** posits that if a full sequence is random, any subsequence extracted from it should also exhibit random characteristics and pass statistical tests. Lastly, **consistency** requires a generator's randomness to be consistent across seed values. Success obtained by generating a sequence from a single seed is insufficient. These assumptions underlie the statistical rigor of the suite but also expose its limits.

2.5 Implementation background

This section provides an overview of the technical implementation including the choice of Python as the programming language, the use of PyTorch for building the neural network, and the specific modules and tools utilized.

2.5.1 Python

Python [53] is a high-level interpreted programming language first released in 1991, as Python 0.9.0, by Guido van Rossum. It is currently maintained by the Python Software Foundation. The latest stable version of Python is Python 3.12, released on October 2, 2023. The language is designed with an emphasis on readability and simplicity, making it one of the easiest languages to learn and use. It has become a dominant choice for machine learning thanks to the extensive ecosystem of libraries and frameworks such as TensorFlow [60], PyTorch [54], scikit-learn [57], or Keras [33]. These tools provide powerful capabilities for developing and deploying machine learning models, ranging from simple algorithms to advanced neural networks.

2.5.2 PyTorch

PyTorch [54] is a versatile and open source machine learning framework, developed by Meta AI, formerly known as Facebook's AI Research lab (FAIR), which is currently also part of the Linux Foundation umbrella [8]. PyTorch has become a leading platform for building and experimenting with deep learning models. Its modular and Python-like design makes it a popular choice for constructing complex neural networks in both academic and industry settings. PyTorch leverages *tensors* (`torch.tensor`) [61], as its core data structure, allowing efficient numerical computations and seamless integration with GPU acceleration, a key factor in executing demanding machine learning algorithms. To implement our neural network and transformer models, we utilized PyTorch's `torch.nn` modules [62].

`nn.Linear` [69] and `nn.ReLU` [73] are fundamental building blocks in PyTorch used when constructing neural networks. `nn.Linear` represents a fully connected layer that performs a linear transformation on the input, mapping it to an output space using learned weights and biases. `nn.ReLU` function adds non-linear behavior to neural networks by implementing the Rectified Linear Unit activation function, described in more detail in Section 2.1.1. Together, these modules represent core components for designing flexible and powerful deep learning architectures. For transformer architecture implementation `nn.ModuleList` [70], `nn.LayerNorm` [68] and `nn.Embedding` [66] were used. `nn.ModuleList` is commonly used to combine many layers of attention and feed-forward modules. This way, the model can process information through these layers repeatedly in a forward pass, and they're still properly recognized as parts that can be trained. In Transformer models, `nn.LayerNorm` is strategically placed either before or after the attention and feedforward layers. Its main purpose is to stabilize the training process by normalizing the outputs of these layers. `nn.Embedding` converts individual input elements, such as words or binary digits, into a more abstract, numerical format referred to as dense vector representations.

In addition to these modules, PyTorch offers a variety of loss functions for different tasks. Loss functions measure the error between a model's predictions and the ground truth values. PyTorch offers a variety of loss functions suitable for different tasks. For instance, Mean Squared Error (MSE) [71] is used for numerical predictions. Cross-Entropy Loss [65] is usually used in data categorization problems, Binary Cross-Entropy (BCE) Loss [63] for yes/no decisions, Negative Log-Likelihood Loss (NLLLoss) [72] for probabilistic models, Smooth L1 Loss [74] for more robust numerical predictions, or Kullback-Leibler Divergence Loss (KLDivLoss) [67] for comparing probability distributions. For tasks with binary outputs where the model directly provides unscaled predictions, known as logits, Binary Cross-Entropy with Logits Loss (BCEWithLogitsLoss) [64] is used.

Furthermore, a robust set of optimizer algorithms is offered. These are crucial when it comes to training deep learning models. Optimizers are responsible for updating model parameters, such as weights and biases, during training. The key optimization algorithms available in PyTorch include Stochastic Gradient Descent (SGD) [81], Adaptive Moment Estimation (Adam) [77], Root Mean Squared Propagation (RMSprop) [80], Adagrad [76], Adadelta [75], Nesterov-accelerated Adaptive Moment Estimation (NAdam) [79] and Adaptive Moment Estimation W (AdamW) [78].

2.5.3 SymPy

SymPy [59] is a Python library for symbolic mathematics. It allows you to perform algebraic operations symbolically rather than numerically. Meaning it works with variables and equations directly, rather than just numbers. Unlike libraries such as NumPy [46] or the math module [42], which perform numerical calculations, SymPy is capable of manipulating the algebraic expression directly.

3. TEACHING NEURAL NETWORK THE SHA-3 HASH FUNCTION

This chapter presents the individual steps of the implementation process, highlighting the development and testing of the SHA-3 toy function, dataset preparation, neural network architecture, training methodology, and evaluation. Each section is devoted to explaining the steps taken to achieve the objectives and the results obtained.

3.1 Implementation of SHA-3 Toy Function

As part of the thesis a simplified version of SHA-3 has been implemented, referred to as a "*toy function*". This implementation is designed to focus on the fundamental principles of the algorithm, while omitting some of the complexities present in the full version.
The motivation behind implementation of this toy function was threefold: First, it provided a valuable opportunity to understand the internal mechanics of SHA-3, such as the absorption and squeezing phase, individual step mappings of the Keccak algorithm, and the role of the Keccak-p permutation step. Second, the toy function serves as a core building block that can be systematically expanded to a full SHA-3 implementation. This kind of modular approach makes the development process more manageable and also provides a clear pathway for transitioning from theoretical understanding to a complete implementation. Third, it provides an excellent way to generate "partial hashes", meaning one can clearly observe how individual rounds of Keccak-*p*, or even single-step mappings, are performed in our closed environment.

In order to transform these theoretical foundations, we developed a Python-based prototype that effectively captures the essence of the Keccak algorithm. Our toy implementation reduces the state size from 1600 bits to 25 bits, omits the suffixes, padding, and uses only 12 rounds compared to full version's 24 rounds, to simplify the demonstration of core concepts. These modifications are all available modifications, discussed in [16], [47], and are also marked in the first column of Table 2.3. In the toy implementation, a simplified state size of 25 bits is arranged as a 5×5 grid of lanes, preserving the essential mathematical principles of the original algorithm. Compared to standardized computational Algorithms 2.1, 2.2, 2.3, 2.4, 2.5, we consider parameter $z = 0$. All Keccak-*p* permutation steps have already been discussed in Section 2.2.2.

Below, we introduce the implementation of every permutation step on a reduced 25-bit state. Each step mapping in the code involves a dedicated function to perform specific bit manipulations and return an updated state. Individual step mapping functions are called to perform sequential transformation of the cryptographic state.

`theta_step` function, see Listing 1, first creates the intermediate arrays `C` and `D`, to store column parity and transformed parity values. It computes `C[x]` by XORing all elements in each column using a vertical iteration. Calculates `D[x]` by XORing adjacent rotated column parities, using modular arithmetic to handle wrap-around. Finally, a new state `a_prime` is created, by XORing each original lane with its corresponding `D[x]` value.

```
def theta_step(A):
    C = [0] * 5
    D = [0] * 5
    for x in range(5):
        C[x] = A[x][0] ^ A[x][1] ^ A[x][2] ^ A[x][3] ^ A[x][4]
    for x in range(5):
        temp = rotate_left(C[(x + 1) % 5], 1)
        D[x] = C[(x - 1) % 5] ^ temp
    a_prime = [[0] * 5 for _ in range(5)]
    for x in range(5):
        for y in range(5):
            a_prime[x][y] = A[x][y] ^ D[x]
    return a_prime
```

Listing 1. *Implementation of the θ step*

`rho_step` function initializes a new 5×5 state grid `a_prime` and iterates through each lane using nested `x` and `y` loops. For each lane, it retrieves the specific rotation offset for the current position, applies the `rotate_left()` function to rotate the lane's bits, and finally stores the rotated value in the corresponding position of `a_prime`. The rotation offsets are pre-defined in the `ROTATION_OFFSETS` dictionary, which maps each `(x, y)` coordinate to its specific rotation amount.

```
def rho_step(A):
    a_prime = [[0] * 5 for _ in range(5)]
    for x in range(5):
        for y in range(5):
            rotation = ROTATION_OFFSETS[(x, y)]
            a_prime[x][y] = rotate_left(A[x][y], rotation)
    return a_prime
```

Listing 2. *Implementation of the ρ step*

pi_step function initializes a new 5×5 state grid a_prime and iterates through each lane using nested x and y loops. For each lane, the permutation rule described in Algorithm 2.3, showcased in Listing 3, is used to wrap around grid boundaries.

```
1  def pi_step(A):
2      a_prime = [[0] * 5 for _ in range(5)]
3      for x in range(5):
4          for y in range(5):
5              a_prime[x][y] = A[(x + 3 * y) % 5][x]
6      return a_prime
```

Listing 3. *Implementation of the π step*

The non-linear chi_step function modifies each lane by XORing its current value with the result of logical ANDing the complement of a neighboring lane and a shifted version of another lane, as described in implementation Listing 4.

```
1  def chi_step(A):
2      a_prime = [[0] * 5 for _ in range(5)]
3      for x in range(5):
4          for y in range(5):
5              a_prime[x][y] = A[x][y] ^ ((~A[(x + 1) % 5][y]) & A[(x + 2) % 5][y])
6      return a_prime
```

Listing 4. *Implementation of the χ step*

As shown in the implementation Listing 5, the iota_step function modifies the value of the lane at position $[0, 0]$ in the state array A, by XORing it with a round-specific constant RC[i]. Constant depends on the current round number i, and the overall number of rounds n_r determined by the parameter b. In our toy function's implementation only the first column of Table 2.5 is used, corresponding to values from RC[0] to RC[11], since the $n_r = 12$.

```
1  def iota_step(A, round_index):
2      a_prime = [row[:] for row in A]
3      a_prime[0][0] ^= ROUND_CONSTANTS[round_index]
4      return a_prime
```

Listing 5. *Implementation of the ι step*

After defining the individual permutation step mappings, we apply them sequentially in a loop for 12 consecutive rounds. Initially, the code sets up a 5×5 grid A, and distributes the input bits across the cells. After that, a 12-round iterative process is applied, as seen in Listing 6. This iterative process implements the core functionality of the hashing algorithm, updating the state grid A round by round. The number of rounds, denoted by $n_r = 12$, corresponds to the specifications for a reduced-round implementation in our toy function. Upon completion of these rounds, the final state of the grid A is restructured into a 25-bit binary string, which constitutes the output of the algorithm.

```python
for round_index in range(12):
    A = theta_step(A)
    A = rho_step(A)
    A = pi_step(A)
    A = chi_step(A)
    A = iota_step(A, round_index)
```

Listing 6. *Single loop iterating over individual Keccak-p steps*

3.2 Dataset preparation

Building on the SHA-3 toy function implementation discussed in Section 3.1, the following steps include generating a dataset of inputs and their corresponding hash outputs. This dataset serves as a crucial resource for analyzing the hash function's properties and training a neural network, as described in the following sections.

To explore the toy function's behavior, we generated all possible 25-bit binary values. This approach ensures coverage, mapping every potential input within the function's 25-bit state space, which spans from 0 to 2^{25}. Generated values were then left padded with zeros, to fill up the state's 5×5 grid structure.

$$x_{\text{padded}} = \text{pad}(x) = \underbrace{0 \ldots 0}_{25-n} x \tag{3.1}$$

The resulting dataset was organized into two text-only files. The first containing the unhashed inputs and the second containing their corresponding hashed outputs. Generating all possible values allowed us to perform a comprehensive validation of the hash function implementation using *Dirichlet's drawer principle*. The Dirichlet's drawer principle also referred to as the *pigeonhole principle*, states that if more items are distributed than there are containers to hold them, at least one container must contain more than one item [47], [1]. By leveraging this principle, we could thoroughly examine every possible input to systematically check the function's reliability. Such a dataset could also be used to observe individual bit-wise mappings within individual rounds or steps of the implemented SHA-3 toy function. Generated values were stored in text-only files for further use. During the experimentation phase, storing generated values in separate files introduced computational overhead. This overhead was then minimized by implementing a dynamic routine that generated the desired unhashed and hashed pairs on demand.

3.3 Neural Network Architecture

Binary nature of hash function presents a significant challenge in their learning and generalization. The high-dimensional space of binary sequences makes learning the exact mapping very difficult, as even minor errors can lead to drastically different hash outputs. Moreover, ensuring the network generalizes to unseen binary sequences while avoiding memorization of specific input-output pairs is crucial for future applications. The neural network's architecture was designed to analyze and learn the learn *Keccak-p* step mappings based on the comprehensive dataset, discussed in Section 3.2.

The input layer of the neural network corresponds to the bit-length of the binary inputs used in the hash function. In our implementation, a 25-bit input is fed into the SHA-3 toy function. Hence, the input layer must contain 25 artificial neurons in order to capture all information provided on an input of the neural network. The output layer must align with the bit-length of the hash function's output, ensuring the network can produce a 25-bit representation of the hashed output. To bring the neural network to life, we buit upon PyTorch's modular approach. The network was constructed using `torch.nn` modules. `SimpleNN` class, inheriting from `nn.Module`, forms the backbone of our model, see Listing 7. It represents a sequence of fully connected layers, each composed of linear transformations `nn.Linear`, followed by non-linear activation functions `nn.ReLU`.

```
1  class SimpleNN(nn.Module):
2      def __init__(self, input_size, output_size, hidden_size):
3          super(SimpleNN, self).__init__()
4          self.fc = nn.Sequential(
5              nn.Linear(input_size, hidden_size),
6              nn.ReLU(),
7              nn.Linear(hidden_size, hidden_size),
8              nn.ReLU(),
9              nn.Linear(hidden_size, hidden_size),
10             nn.ReLU(),
11             nn.Linear(hidden_size, output_size)
12         )
13
14     def forward(self, x):
15         return self.fc(x)
```

Listing 7. *Modular FNN implementation with 3 hidden layers*

During training, backpropagation is used to efficiently compute gradients, allowing the model to learn and adapt through currently selected optimization algorithms such as SGD (`torch.optim.SGD`), Adam (`torch.optim.Adam`), RMSprop (`torch.optim.RMSprop`), Adagrad (`torch.optim.Adagrad`), Adadelta (`torch.optim.Adadelta`) or NAdam (`torch.optim.NAdam`). PyTorch's tensor operations make it simple to process binary input and output data by converting arrays and Python lists into `torch.tensor` objects.

This PyTorch implementation offers a flexible platform for experimenting with different network architectures, calculating loss using Mean Squared Error (MSE), and tracking performance metrics across various hyperparameter configurations, further discussed in following sections.

3.4 Training and Experimentation Process

With the architecture established, the next step involves training the network to model the *Keccak*-p mappings. Performance within training across various hypeparameters is evaluated. The training process along with the experimental results, is detailed in following sections.

3.4.1 Impact of Optimizer Algorithm on Loss

Focus of this section is to understand how the choice of optimizer affects network's learning process. The experiment evaluates the effectiveness of various optimizers, SGD (`torch.optim.SGD`), Adam (`torch.optim.Adam`), RMSprop (`torch.optim.RMSprop`), Adagrad (`torch.optim.Adagrad`), Adadelta (`torch.optim.Adadelta`) or NAdam (`torch.optim.NAdam`), in minimizing the MSE loss function.

The network is trained on a dataset of 256 sample pairs of unhashed and hashed data (equivalent to 2^8 bits), for 50 epochs, using a batch size of 32. Batch size refers to the number of training samples processed by the model at once before updating the weights of the model during training. Within the experimental setting, we explored a range of hidden layer sizes and learning rates to identify optimal network configurations. *Learning rate* (b) is a critical hyperparameter in the training of neural networks that determines the step size of the optimization algorithm when adjusting the network's weights to minimize the loss function. The *hidden layer size* (w) refers to the number of neurons in a hidden layer, which controls the network's capacity to learn patterns from the data. These insights are visualized using a heatmap, where the dimensions are determined by the hyperparameter ranges. Horizontal axis represents 15 values of the hidden layer size (w) evenly spaced from 1 to 128. Vertical axis represents 15 values of the learning rate (b) evenly spced between values of 0.001 and 0.5. This results in a 15×15 grid, that maps the relationship between w, b, and the corresponding loss values.

As can be observed in Figure 3.1, the choice of optimizer can significantly impact the model's performance. SGD and Adam proved to be robust to variations in learning rate and hidden layer size, offering a wider range of optimal configurations. However, SGD and Nadam were more sensitive to changes in learning rate values. This may require precise tuning to achieve optimal results. Adaptive optimizers like Adagrad and Adadelta showed a stable performance across different learning rates.

Increasing the hidden layer size generally reduced MSE loss, but the improvement in performance started to slow down, and eventually stopped after a certain point. The RMSprop optimizer seemed to perform better with smaller hidden layer sizes. Adam optimizer consistently performed better, with lower error rates, across a wider range of hyperparameter settings. This makes Adam an optimal choice for our specific dataset and model architecture. While SGD can work well, it is sensitive to the learning rate and requires careful adjustment to perform its best. Although, Adadelta and Adagrad provide a stable performance, they might require larger networks to achieve similar performance as Adam or Nadam. For most scenarios, Adam or Nadam are a popular choice due to their adaptability and consistent performance.

In subsequent experiments, the Adam optimizer will be used as the default optimization algorithm to leverage its robust learning capabilities, with an optimal learning rate equal to 0.001.

3.4.2 Impact of Input Complexity and Hidden Layer Size on Loss and Accuracy

In this section, the impact of input complexity, measured as the number of binary samples, and the hidden layer size on the network's performance is evaluated through loss and accuracy metrics.

Input complexity ranges from 2^1 to 2^{14}. Similarly, the hidden layer size ranges from 32 to 1024 neurons, increasing exponentially in powers of two. Heatmaps were generated to visualize the relationship between these hyperparameters and their impact on model's performance. Loss values highlight regions where the network has difficulty learning the step mappings between unhashed inputs and their corresponding hashed outputs. Accuracy values indicate the model's ability to correctly predict hashed outputs, based on unhashed inputs within the given training data. Each combination of input complexity and hidden layer size is trained for 50 epochs using the Adam optimizer with learning rate 0.001 and batch size set to 32.

In Figure 3.2, trade-off between model capacity and input complexity can be observed. Smaller hidden layers were insufficient to handle complex input mappings, while larger ones provided enough capacity. With hidden layer sizes of 512 and 1024, model performs well on both input samples of small and higher complexity.

The heatmap of accuracy in Figure 3.3, suggests that as the input complexity increases, the model accuracy generally improves, particularly for larger hidden layer size represented by the green cell color. The highest accuracy of 100% is achieved with the hidden layer sizes of 512 and 1024 neurons, indicating their ability to effectively learn complex input-output mappings.

Our findings show that as the width of the model increases, its performance improves. This suggests that models with higher dimensionality have the potential to achieve superior results. Overall, this experiment highlights the importance of balancing model capacity, as represented by the hidden layer size, with the complexity of the input data. Insights from this experiment can be used in further development of an effective neural network architecture for our application.

3.4.3 Model Performance in Training and Generalization Across Multiple Rounds

The primary goal is to analyze and evaluate the model's performance in learning and generalizing the input-output mappings as the complexity of the SHA-3 hashing function increases, with the increasing number of rounds, using our small model. By systematic increase in the number of rounds n_r of the SHA-3 toy function, we aim to understand the relationship between function complexity and model performance. In this context, we refer to one round as the *Keccak-p* permutation round, consisting of five consecutive steps, discussed in Section 2.2.2.

Initially, experiment begins with only one round, in literature also referred to as the Round 0. Round number is then systematically increased up to 12. As per training, the MSE loss function is used. Further metrics like optimizer, learning rate, and neural network structure remain unchanged. The hidden layer size (w) was fixed to 1024 neurons. 256 samples were used within this experiment in order to represent the scaled down complexity of the SHA-3 toy function on a rather smaller dataset. Dataset was divided into two sets using a fixed ratio, 95% for training and 5% for evaluation. However, this split can be adjusted to fit specific needs of the model training process within further experiments. Training accuracy is calculated during each epoch, comparing model's predictions with the ground truth values, actual hashed values, in the training dataset. Evaluation accuracy is calculated using the same principle as the training accuracy, but the predicted value is compared with a separate evaluation dataset. Model's learning progress is tracked over 100 epochs, tracking both training and evaluation accuracy. Training and evaluation accuracy can be compared in Figure 3.4 for the first 6 rounds, and Figure 3.5 for the subsequent $7-12$ rounds.

Graphs plotting model's performance across multiple rounds both during training and generalization showcase that the neural network can effectively learn the training data round mappings even with more function of more rounds. After the first five permutation steps, training and evaluation accuracy align relatively closely with both reaching near 100%. On the other hand, its ability to generalize decreases with the task complexity increase. Over the span of 12 rounds of the SHA-3 toy function, training accuracy of the model remains high, but evaluation accuracy stagnates near 50%.

This suggests several potential challenges, such as overfitting, which is evident by the widening gap between the training and evaluation accuracy. This may be also likely due to a combination of other factors. Secondly, the network's limited capacity of only 3 layers and 1024 neurons, bit mappings of 12 rounds might pose a far more complex challenge that may be obvious at first glance. Thirdly, choosing small dataset to observe the behavior of the bit mappings. Unhashed input uses only eight left-padded bits, whereas the hashed output is using complete 25-bit state. These factors may have resulted in model's hard time learning meaningful features, and random-like predictions in higher rounds.

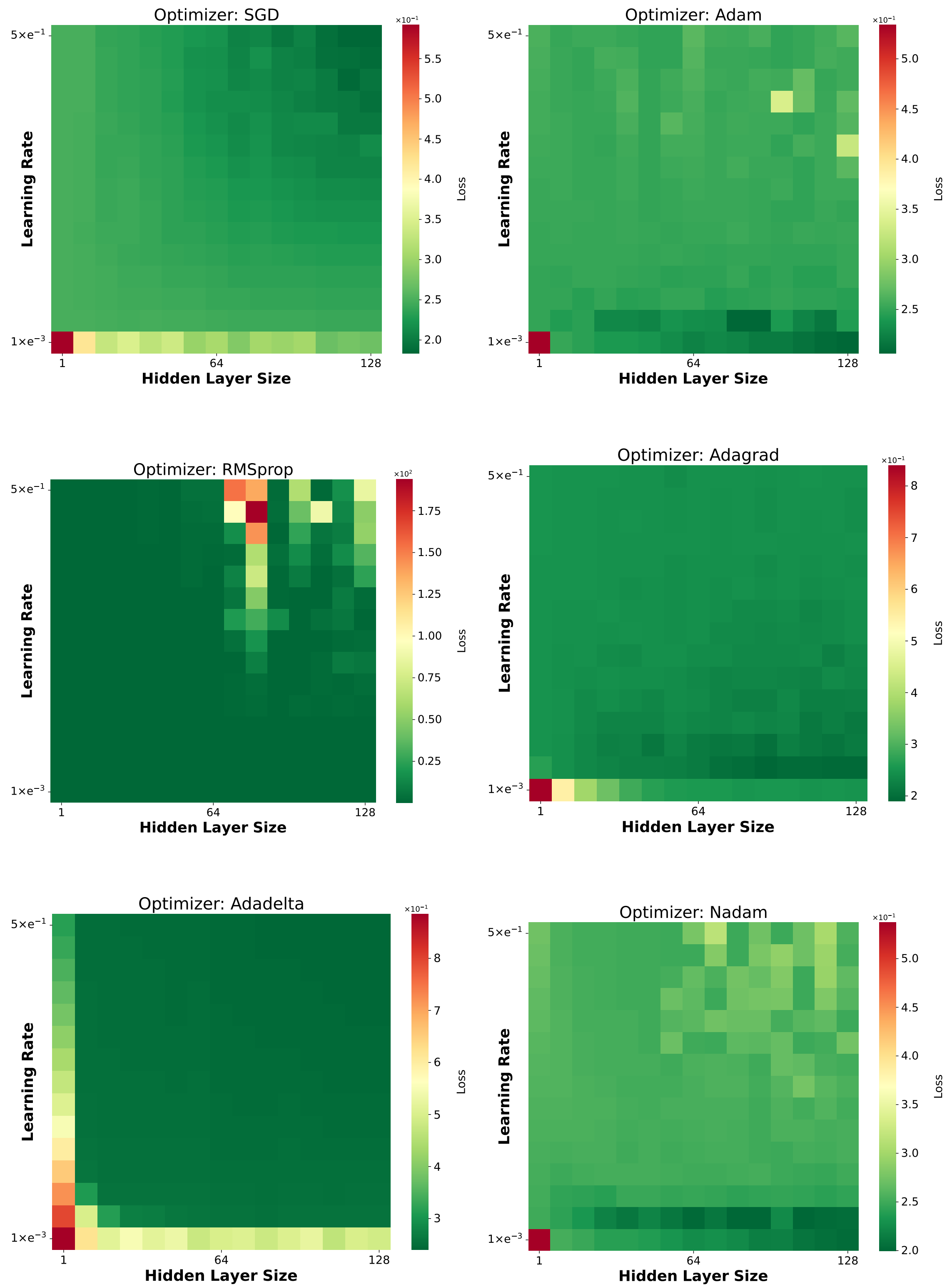

Figure 3.1. *Comparison of heatmaps selected optimizer algorithms on Loss*

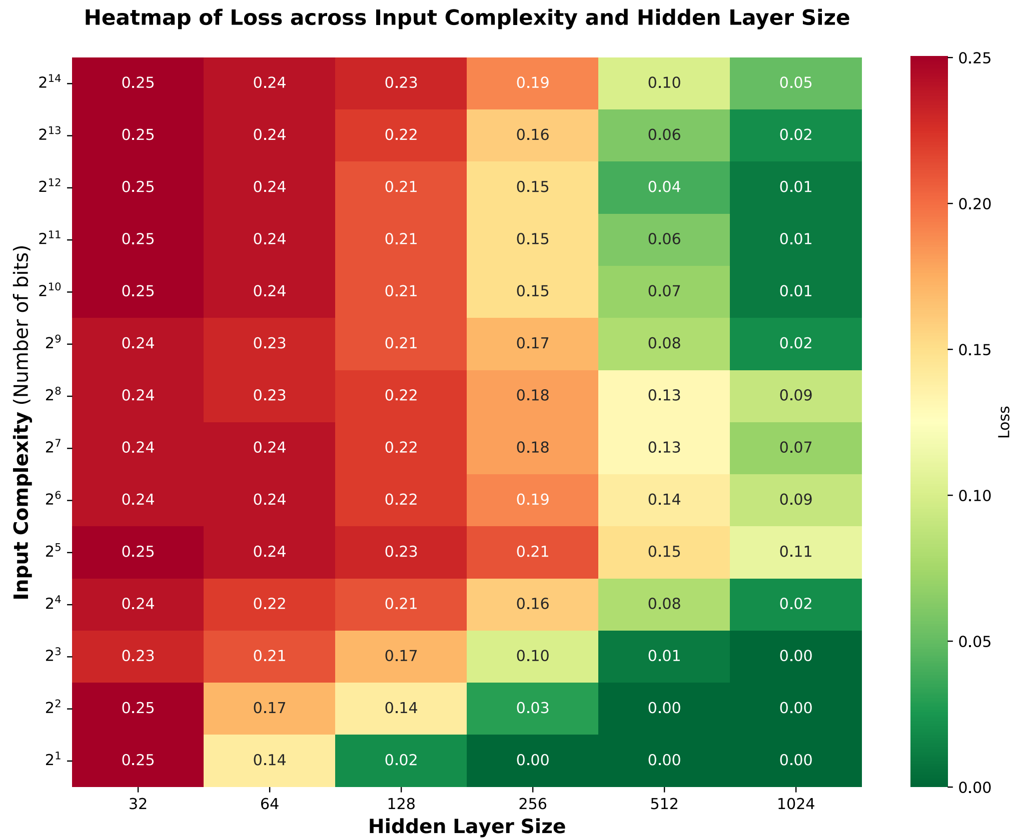

Figure 3.2. *Heatmap of Loss across Input Complexity and Hidden Layer Sizes*

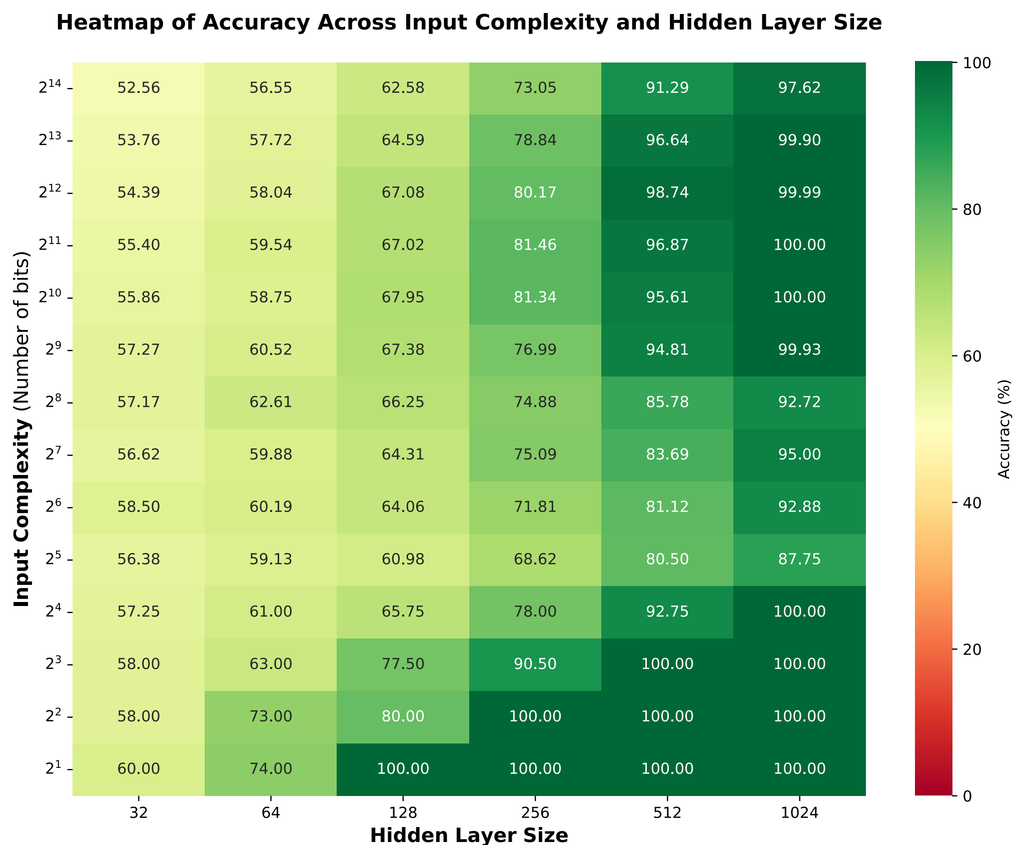

Figure 3.3. *Heatmap of Accuracy across Input Complexity and Hidden Layer Sizes*

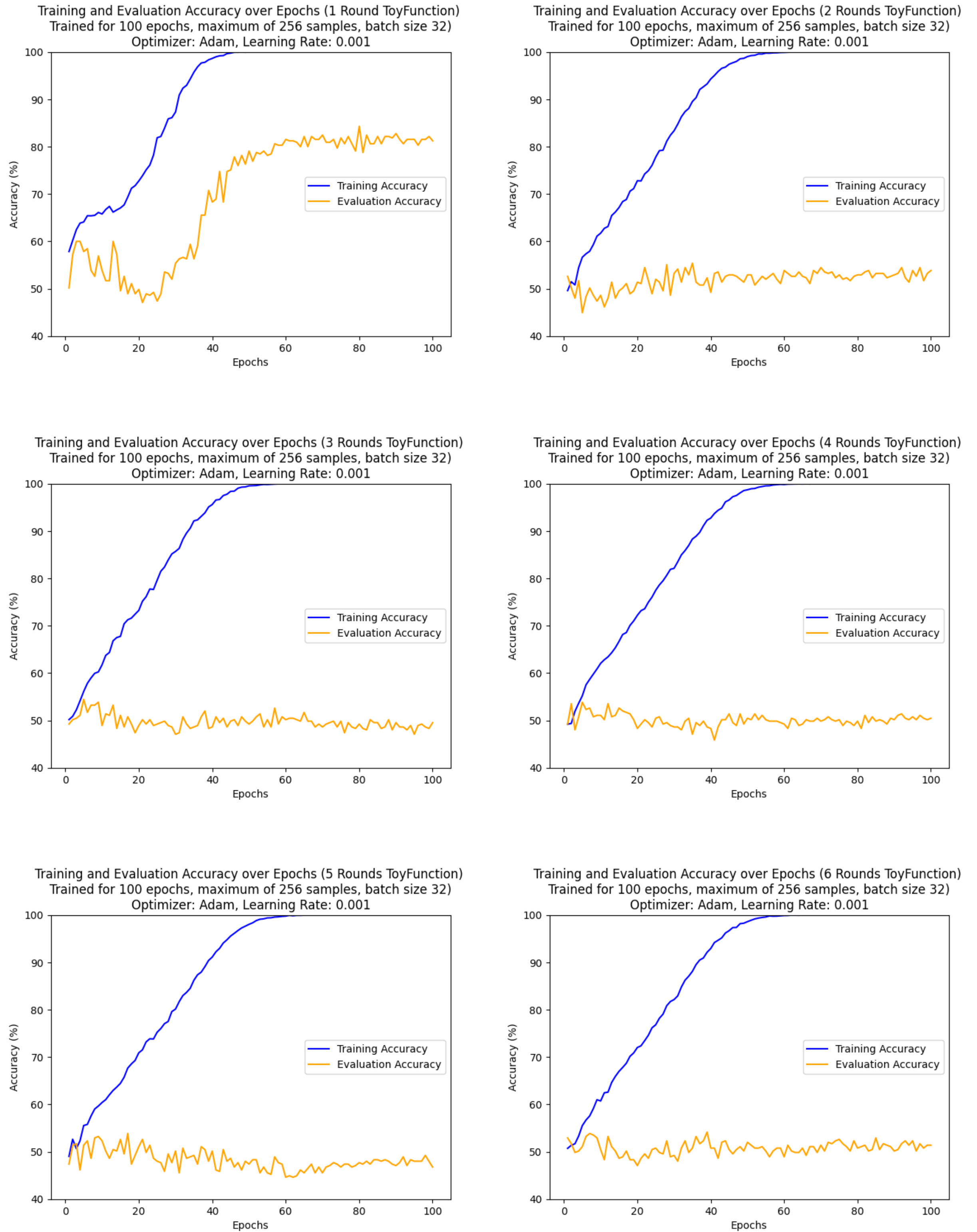

Figure 3.4. *Model Performance in Training and Generalization Across* $1-6$ *Rounds*

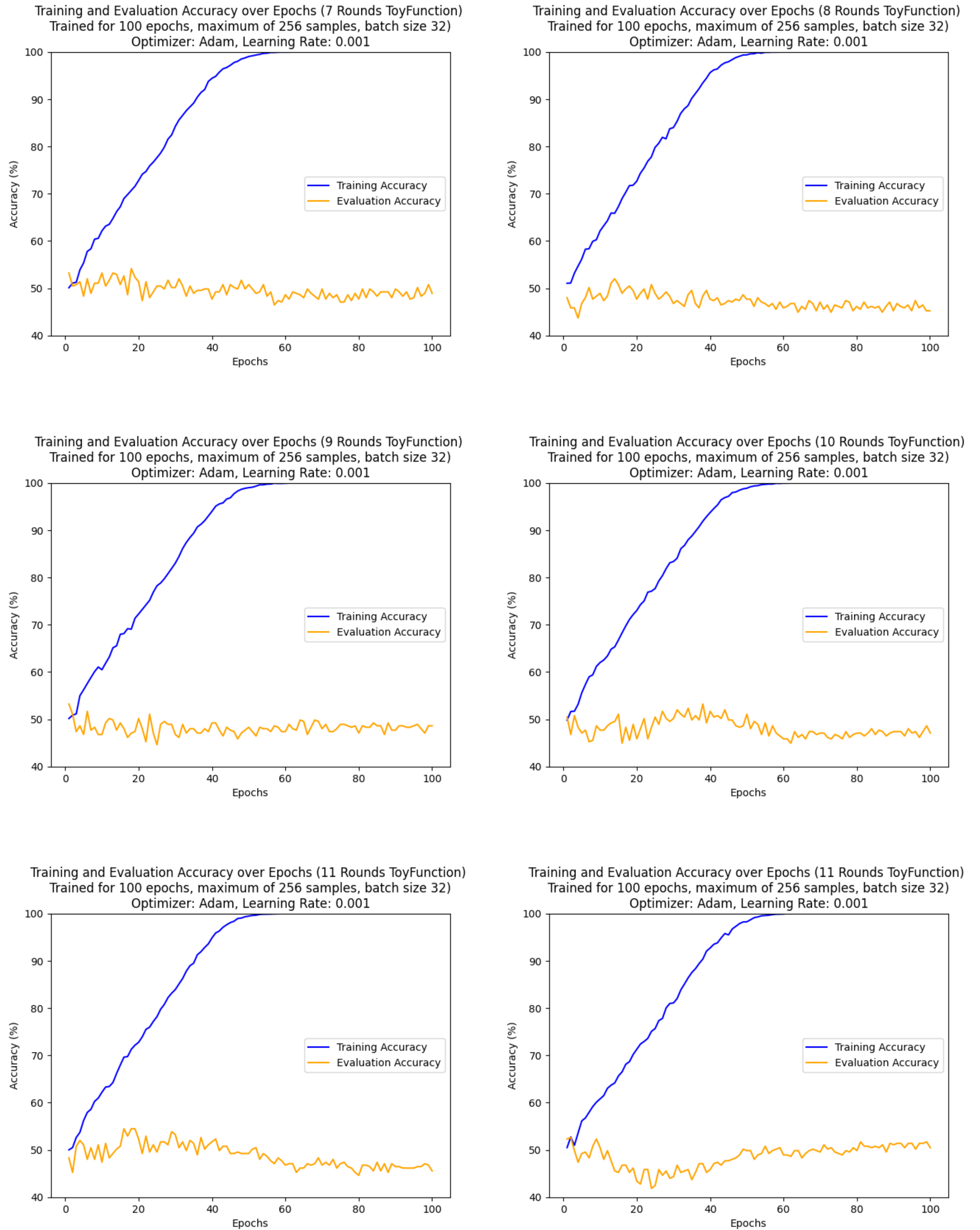

Figure 3.5. *Model Performance in Training and Generalization Across* $7-12$ *Rounds*

4. MACHINE LEARNING AIDED RANDOMNESS ASSESSMENT

This chapter details the implementation process, focusing on how pseudorandom number generators were used to create binary sequences. Following this, a model leveraging a Transformer architecture to assess randomness was implemented. This section revolves around the research question, *whether ML models could replace traditional statistical tests such as the* ***NIST SP 800-22*** *test suite, thanks to their advanced pattern recognition capabilities.* Each step of the implementation process is described, from dataset preparation and the Transformer's architecture to training methodology and evaluation.

4.1 Implementation of pseudorandom binary sequence generators

To evaluate the model's ability to detect patterns and classify randomness against NIST's standardized statistical test suite, several binary sequence generators were implemented. Codebase included two Python-based implementations of a Linear Congruential Generator, with underlying mathematics detailed in Section 2.3.2. The first LCG mode has fixed parameters, allowing the user to specify the initial seed x_0, multiplier a, increment c, and modulus p. With an initial seed x_0, this function accepts a `sequence_length` parameter, upon which it returns a computed sequence of desired length.

```
def lcg(x0, num_outputs):
    p = 2**31 - 1
    a = 45289
    c = 0
    current_x = x0
    outputs = []
    for _ in range(num_outputs):
        current_x = (a * current_x + c)  % p
        uniform_value = current_x / p
        binary_output = 0 if uniform_value <= 0.5 else 1
        outputs.append(binary_output)
    return outputs
```

Listing 8. *Implementation an LCG with fixed parameters*

The second mode provides more flexibility, enabling user to adjust the multiplier, increment while selecting a range of initial seeds to generate the sequence. Both LCG implementations produce sequences of raw integers. Binary sequences are created through a method known as *thresholding*. In this context, thresholding refers to converting normalized numerical outputs into binary values by assigning a 1 if the value exceeds a predefined threshold, usually 0.5, and 0 otherwise. Numerical outputs x_n are obtained by normalizing the raw values z_n by the modulus p. The thresholding rule is then:

$$b_n = \begin{cases} 1 & \text{if } x_n > 0.5 \\ 0 & \text{otherwise} \end{cases}$$

Beyond the LCG-based implementations, suite includes a Python implementation of the BBS generator, with mathematical foundation discussed in Section 2.3.3. To initialize the sequence, seed value x_0 must be selected such that $\gcd(x_0, n) = 1$, ensuring that the sequence remains within the multiplicative group modulo n. The generator updates its internal state by repeatedly squaring the current value, and the least significant bit (LSB) of each result is extracted and added into the resulting output sequence.

```
1  def bbs(x0, num_outputs):
2      p = 30000000091
3      q = 40000000003
4      n = p * q
5
6      current_x = pow(x0, 2, n)
7      outputs = []
8
9      for _ in range(num_outputs):
10         current_x = pow(current_x, 2, n)
11         binary_output = current_x % 2
12         outputs.append(binary_output)
13
14     return outputs
```

Listing 9. *Implementation of a BBS generator with fixed parameters*

Additonally, the suite included a set of deterministic sequence generators that serve as control examples with predictable patterns. These generators produce binary sequences with pre-defined structures to contrast with the pseudorandom outputs. One of the generators creates patterns by strategically positioning varying numbers of consecutive 1 bits across different positions in fixed-length sequences. Another implementation offers multiple structured pattern types, such as *alternating bits*, *edge flipping*, *center flipping*, *block patterns*, *incrementing values*, and *parity flipping*.

4.2 Sequnce-based Randomness Assessment

This script implements an automated framework on top of the NIST SP 800-22 statistical test suite to evaluate the randomness of individual binary sequences. Individual evaluations are later converted into a *one-hot encoded* format. The statistical test suite is a `C` program organized into subdirectories corresponding to each test. Each test outputs its results to a `results.txt` file, typically containing p-values that indicate whether the tested sequence passes the randomness criteria at a chosen significance level α.

The underlying principle inside the implemented script is similar to statistical hypothesis testing in the **SP 800-22** suite, discussed in Section 2.4. It compares resulting p-values against the specified significance level $\alpha = 0.01$ across `results.txt` files generated in each subdirectory. A sequence is considered random if it passes a given test with a p-value exceeding the significance level α. Otherwise, it is considered non-random.
The random sequence is assigned label `1`, label `0` otherwise. Tests that produce more than one p-value for each sequence are handled with particular care. For example, *Serial* and *Cumulative Sums* tests require all relevant p-values to surpass the threshold. The *Non-Overlapping Template* test presents additional automation complexity. The assessment framework collects results across 148 templates, requiring a sequence to pass at least 90% of its 148 template tests to receive either a random or non-random label.
The script then combines all the results from the complete set of tests and generates a final `randomLabels.txt` file. This file contains a single definitive label per sequence, with one label per line, where `1` indicates a sequence passed all selected statistical tests, and `0` indicates that a given sequence failed at least one of the tests.

This sequence-based evaluation framework objectively assesses which sequences pass or fail the standardized tests included in the NIST SP 800-22 statistical test suite. Examining which sequences fail the standardized battery of tests reveals insights into the statistical properties that distinguish non-random sequences from random ones.

4.3 Dataset preparation

Binary sequences produced by both pseudorandom number generators and deterministic pattern-based generators, outlined in Section 4.1, were stored as text-only files. These sequences underwent evaluation through the NIST SP 800-22 test suite and were subsequently classified using the automated assessment framework.

Each sequence was assigned a binary label based on whether it passed or failed the NIST SP 800-22 battery of tests, as described in Section 4.2. This resulted in a split dataset between two text-only files, first containing binary sequences of equal length, and the second one containing labels of `0` for non-random, or `1` for random sequences.

A methodology that ensured an equal class distribution by generating a balanced number of sequences from both pseudorandom generators and deterministic pattern-based generators was utilized to mitigate potential biases arising from unequal class representation between random and non-random sequences.

4.4 Classification Transformer Architecture

The binary nature of sequence randomness detection presents unique challenges, requiring a model capable of capturing complex patterns in high-dimensional binary spaces. While feed-forward neural networks can capture relationships to some extent, their limited ability to model sequential dependencies becomes a limiting factor. Moreover, binary vocabulary introduces sparsity in token representation, and the need for positional awareness becomes important in distinguishing structurally meaningful patterns.
The `BitSequenceTransformer` represents a specialized application of transformer architecture optimized for binary sequence analysis, designed to classify binary sequences based on randomness properties.

```
class BitSequenceTransformer(nn.Module):
    def __init__(self):
        super(BitSequenceTransformer, self).__init__()
        vocab_size = 2

        config = BertConfig(
            vocab_size=vocab_size,
            hidden_size=Config.HIDDEN_SIZE,
            num_hidden_layers=Config.NUM_HIDDEN_LAYERS,
            num_attention_heads=Config.NUM_ATTENTION_HEADS,
            intermediate_size=Config.HIDDEN_SIZE * 4,
            max_position_embeddings=Config.MAX_POSITION_EMBEDDINGS,
            type_vocab_size=1,
        )

        self.bert = BertModel(config)
        self.dropout = nn.Dropout(0.1)
        self.classifier = nn.Linear(Config.HIDDEN_SIZE, 1)
        self.sigmoid = nn.Sigmoid()

    def forward(self, x):
        batch_size, seq_length = x.shape
        attention_mask = torch.ones(batch_size, seq_length, device=x.device)
        outputs = self.bert(input_ids=x, attention_mask=attention_mask)
        pooled_output = outputs.pooler_output
        pooled_output = self.dropout(pooled_output)
        logits = self.classifier(pooled_output)
        probs = self.sigmoid(logits)
        return logits, probs
```

Listing 10. *BitSequenceTransformer for randomness assessment task based on BERT architecture*

To effectively address these challenges, model's architecture is based on a modified BERT implementation. In contrast to conventional sequence models that handle inputs step-by-step, this architecture operates on the entire sequence in parallel, which enables it to capture complex dependencies between bits across all positions. Input layer utilizes a vocabulary of two tokens, representing 0 and 1, and projects them into a hidden embedding dimension of 128. Input sequences are then transformed into learned embeddings of shape (`batch_size, sequence_length, 128`), which are paired with positional encodings that are capable of handling sequences of up to one million tokens.

The encoder consists of two transformer layers, each featuring four self-attention heads and an intermediate feed-forward network with 512 neurons. Approach, that corresponds to a standard BERT configuration. To prevent the model from overfitting, a dropout layer with a 10% rate is applied to the pooled vector. This regularization technique, implemented as `nn.Dropout(0.1)`, works by randomly deactivating a tenth of model's features during the training process. Lastly, a sigmoid activation function is applied to the raw output logit in order to convert it into a probability score. This probability score ranges from 0 to 1, representing the likelihood of the input belonging to the positive class, i.e., the sequence classified as random.

4.5 Training and Experimentation Process

During training, the model uses the AdamW optimizer (`torch.optim.AdamW`) with a configurable learning rate, ranging from 0.0001 to 0.1. The model is trained to minimize the Binary Cross-Entropy with Logits (`nn.BCEWithLogitsLoss`), a well-suited loss function for the binary classification problem. Training loop also includes comprehensive evaluation metrics offering a more detailed view of how well the model performs, especially in cases when dealing with potential class imbalance. Within our setup, a *positive* prediction refers to the model classifying a sequence as random. Thus, a *true positive* means the model correctly identified a truly random sequence. *Precision* measures the proportion of all correct predictions, calculated as:

$$Precision = \frac{TruePositive}{TruePositive + FalsePositive}$$

Recall measures model's ability to correctly identify all positive cases:

$$Recall = \frac{TruePositive}{TruePositive + FalseNegative}$$

F1 score is the harmonic mean of *Precision* and *Recall*, offering a balanced metric:

$$F1 = 2 \cdot \left(\frac{Precision \cdot Recall}{Precision + Recall} \right)$$

4.5.1 Hyperparameter Search and Optimization

This section measures and evaluates the influence of key architectural and training parameters on model performance. The experiment performs a systematic hyperparameter grid search to identify optimal configuration for transformer-based model binary sequence randomness assessment. The search covered a range of values for learning rate spanning from 0.0001 to 0.1 across an increasing number of transformer layers, a growing number of attention heads, embedding dimensions, and varying training step durations. Each configuration was evaluated based on its F1 score and accuracy to ensure a fair comparison under potential class imbalance.

Heatmap in Figure 4.1 visualizes validation accuracy across an increasing number of hidden layers and attention heads. Interestingly, the highest accuracy 0.9754 was achieved with only two hidden layers and four attention heads. This suggests a smaller architecture provides enough capacity to capture the relevant structure in binary sequences.
In contrast, deeper models with a higher number of layers did not outperform smaller ones, which suggests the model's overfitting, insufficient regularization, or the simplicity of the classification task. A notable drop in performance can be observed in high-layer, low-head configurations, indicating that model complexity must be appropriately scaled to fit the specific requirements of the task. The best performing combination of hidden layers and number of attention heads was used for subsequent experiments.

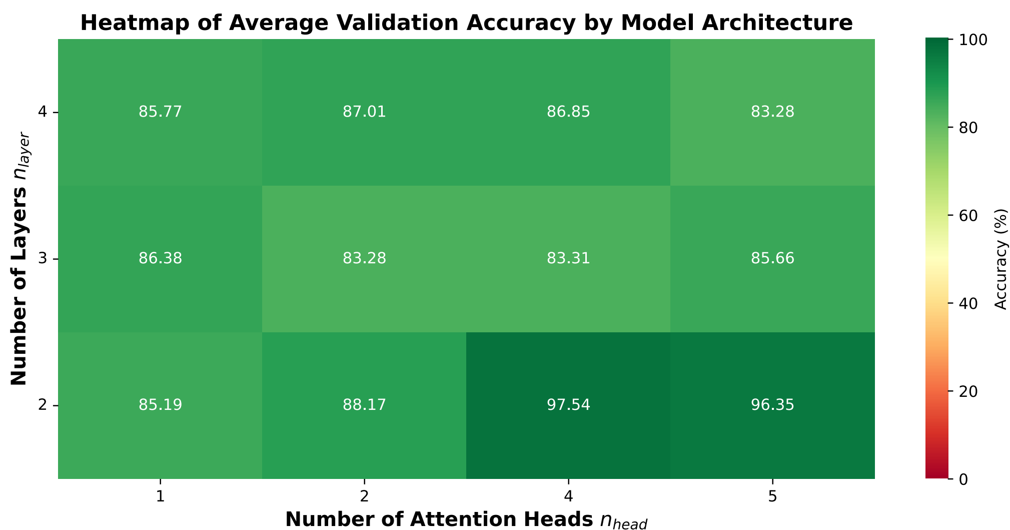

Figure 4.1. *Heatmap of Average Validation Accuracy by Model Architecture*

The second heatmap presented in Figure 4.2, highlights the effect of training hyperparameters on how the selected learning rate and batch size affect the validation accuracy. Results reveal that the choice of learning rate significantly affects the model's performance.

Both 0.0001 and 0.001 yield consistently high accuracy across all batch sizes, while a higher learning rate of 0.005 leads to a significant decrease in the model's performance, as low as 0.6155. This effect is referred to as the *optimization stability trade-off*. While using a larger learning rate promises quicker training, it can make gradient updates unstable, leading to a decrease in the model's performance. However, batch size had a slight impact when the learning rate was appropriately chosen, which aligns with the *noise gradient hypothesis*.

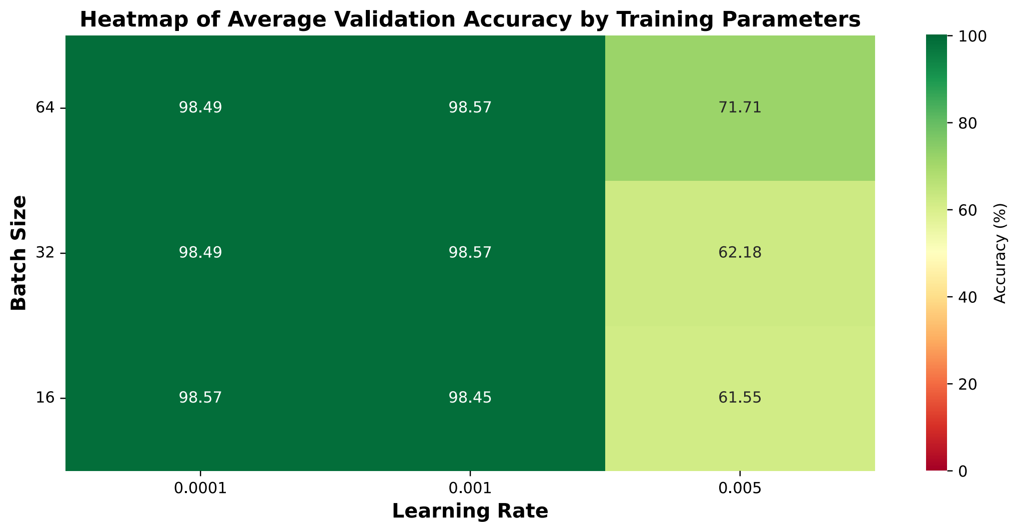

Figure 4.2. *Heatmap of Average Validation Accuracy by Training Parameters*

Experimental findings reveal that architectural and optimization hyperparameters are critical for optimal model performance. Careful tuning is essential when deploying transformer-based models for binary sequence randomness assessment. Alongside consistent performance across various configurations, transformer-based models showed strong potential for learning the structural and statistical properties of binary sequences.

Notably, the model maintained high accuracy despite variations in sequence source and training parameters, which higlights model's robustness and adaptability for such a task. Given above, it is plausible to consider transformer models as a potential data-driven alternative, or at least a complementary approach, to NIST SP 800-22 standardized statistical test suite. Unlike traditional tests that rely on handcrafted statistical properties, these models can learn complex, non-linear dependencies directly from data. This capacity is valuable for evaluating randomness in settings where classical assumptions do not hold, or more advanced, pattern-based detection is required. Presented results demonstrate that transformer architectures are effective and generalize well to broader scenarios of binary sequence assessment.

5. MACHINE LEARNING DRIVEN NEXT-BIT PREDICTION

This chapter details the implementation stages for next-bit prediction sequence modeling. The central hypothesis states that *if a binary sequence passes **NIST SP 800-22** statistical tests for randomness, predicting the next bit from preceding bits should not be computationally feasible.* The chapter covers the development of a dynamic LCG pseudorandom number generator with period optimization, dataset preparation, transformer model architecture, training methodology, and evaluation metrics with results.

5.1 Implementation of Linear Congruential Generator

Following the implementation of the linear congruential generator in Section 4.1, this second implementation demonstrates a more sophisticated approach, enhanced by a dynamic parameter selection. Rather than relying on correctly hard-coded values, the code programmatically generates optimal values for parameters a and c to remain coprime with the selected modulus p. Dynamic parameter selection is based on the *Hull-Dobell theorem*, discussed in Section 2.3.2.

```
1  def find_coprimes(p, count: int = 5,
2                        low: int = 3,
3                        high: int = 1_000,
4                        seed: int = 42) -> list[int]:
5      torch.manual_seed(seed)
6      coprimes = []
7      while len(coprimes) < count:
8          candidate = int(torch.randint(low, high, (1,)).item())
9          if candidate % 2 == 1:
10             coprimes.append(candidate)
11     return coprimes
```

Listing 11. *Generating odd integers coprime with modulus p*

These conditions are directly embedded into the LCG generation pipeline using `find_coprimes` and `find_a` functions, shown in Listing 11 and Listing 12, respectively. This ensures that only valid full-period sequences are used, minimizing possible repetition. Additionally, it guarantees statistical completeness of the generated sequence.

Such approach allows the model to be tested not just on specific patterns but on a wide range of structurally valid sequences, providing a better measure of its generalization and robustness.

```
def find_a(p, limit=None):
    if limit is None:
        limit = p
    factors = sympy.primefactors(p)
    if p % 4 == 0:
        factors = [4 if f == 2 else f for f in factors]
        factors = [f for f in factors if f != 2]
    unique_factors = set(factors)
    lcm = 1
    for factor in unique_factors:
        lcm *= factor
    result = []
    for k in range(1, limit // lcm + 1):
        a = k * lcm + 1
        if a < limit:
            result.append(a)
    return result
```

Listing 12. *Finding suitable parameter a for maximizing period of the LCG*

5.2 Dataset Preparation

The dataset creation methodology employed a multi-method approach to extract binary sequences from linear congruential generator, enabling comprehensive analysis of predictable patterns across different binary representations. Five distinct extraction methods were implemented to capture various aspects of LCG output characteristics. These characteristics span from the least significant bits, which are cryptographically easy to break and do not exhibit much statistical importance, to the most significant bits (MSBs), which are statistically stronger and show more reliable structure.

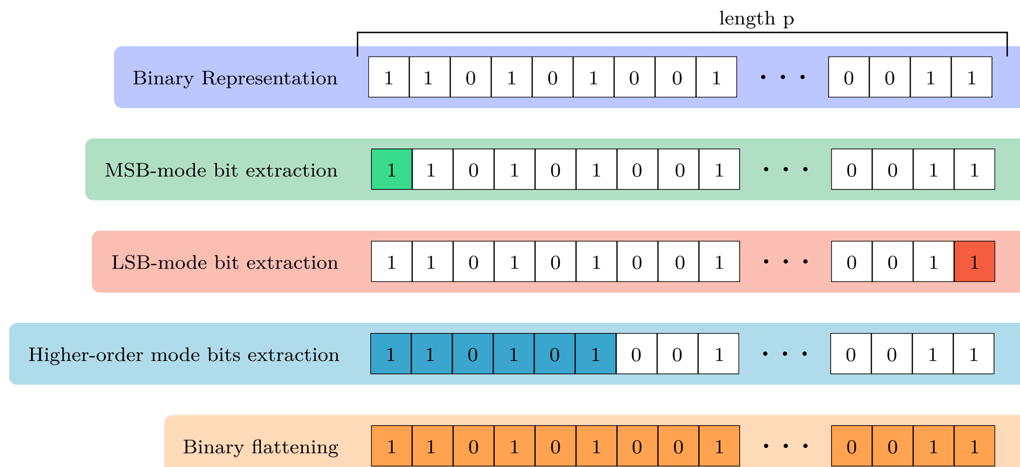

Figure 5.1. *Bit position extraction for binary sequence creation*

Implemented extraction methods cover both single-bit and multi-bit approaches, with careful consideration of the mathematical properties in mind. The LSB extraction method targets the most cryptographically vulnerable component of LCG output, as these bits typically exhibit the poorest statistical properties and are most vulnerable to prediction attacks. For instance, when both the multiplier a and increment c are odd, and the modulus p is of the form 2^k, the LSB-extracted sequence can degrade into a simple alternating pattern of zeros and ones. In contrast, the MSB extraction leverages the upper-order bits, which demonstrate excellent statistical distribution, while remaining potentially predictable due to the linear nature of the underlying mathematical principles of the generator. The binary flattening approach keeps all the information by turning each LCG integer into its complete binary form. This allows to analyze patterns across all bit positions at the same time. A binary sequence creation through *uniforming*, in literature interchangeably also referred to as thresholding, has been employed. The process of thresholding is discussed in Section 4.1.

The primary dataset creation method focuses on higher-order bits extraction with configurable flattening. This represents a more balanced approach between information preservation and the created sequence's statistical significance. This approach was influenced by the methodology proposed in Java's `java.util.Random` class and the `POSIX/glibc` `rand48` family of functions [29], which operate on 48-bit internal values but return only the 32 most significant bits at each iteration. Similarly, the implemented method extracts the user-specified number of the most significant bits (MSBs). Extraction is defined from a single 1 MSB, which simplifies to MSB-only extraction mode, to $(p-1)$ MSBs from each LCG output. Extracted bits are then flattened into continuous binary sequences.

Each extraction method produces sequences formatted for next-bit prediction tasks. Input sequences contain $n-1$ bits and targets representing the nth bit to be predicted. Here, n represents the user-defined sequence length. Resulting datasets maintain consistent sequence lengths and batch structures across methods, allowing a fair transformer performance comparison.

5.3 Next-Bit Transformer Architecture

Traditional approaches to binary sequence prediction have mostly relied on statistical methods and classical machine learning techniques. However, the problem with these approaches is their inability to cope with dependencies that extend across arbitrarily long sequences. The `BinaryGPT` model represents a specialized decoder-only transformer architecture designed for next-bit prediction in pseudorandom number generators.

The model employs a causal self-attention mechanism to predict the next bit in a binary sequence, with particular application to analyzing the predictability of LCG-generated outputs.

Model processes sequences of binary tokens 0 and 1 through single-bit embedding, which is mapped to a learned embedding vector. The model then predicts the probability of the next bit being either 1 or 0. Token embeddings are combined with learned positional encodings, which help the model understand both the content and sequence position, crucial for detecting patterns in pseudorandom sequences that may have positional dependencies.

```
class CausalSelfAttention(nn.Module):
    def __init__(self, config):
        super().__init__()
        assert config.n_embd % config.n_head == 0
        self.c_attn = nn.Linear(config.n_embd, 3 * config.n_embd, bias=False)
        self.c_proj = nn.Linear(config.n_embd, config.n_embd, bias=False)
        self.c_proj.NANOGPT_SCALE_INIT = 1
        self.n_head = config.n_head
        self.n_embd = config.n_embd

    def forward(self, x, output_attentions=False):
        B, T, C = x.size()
        qkv = self.c_attn(x)
        q, k, v = qkv.split(C, dim=2)
        k = k.view(B, T, self.n_head, C // self.n_head).transpose(1, 2)
        q = q.view(B, T, self.n_head, C // self.n_head).transpose(1, 2)
        v = v.view(B, T, self.n_head, C // self.n_head).transpose(1, 2)

        att = (q @ k.transpose(-2, -1)) * (1.0 / math.sqrt(k.size(-1)))
        att = att.masked_fill(torch.triu(torch.ones(T, T, device=x.device, dtype=torch.bool)
                                                    diagonal=1), float('-inf'))
        att = F.softmax(att, dim=-1)

        y = att @ v

        y = y.transpose(1, 2).contiguous().view(B, T, C)
        y = self.c_proj(y)

        if output_attentions:
            return y, att
        return y
```

Listing 13. *Causal Self-Attention inside the Next-Bit transformer architecture*

The architecture of `BinaryGPT` consists of three primary components: a causal self-attention mechanism, Multi-Layer Perceptron (MLP), and transformer blocks that integrate these components with normalization and residual connections. This design incorporates layer normalization and residual connections to stabilize the training process and ensure efficient gradient flow.

Implemented attention mechanism operates through a multi-head structure where the input embeddings are linearly transformed into three distinct representations Query (Q), Key (K) and Value (V). Instead of three individual transformations, the model applies only one linear transformation `c_attn` to the `n_embd`-sized input embeddings, converting them into a combined `3 * n_embd` space.

Attention mechanism uses several parallelly working attention heads, with each attention head focusing on a different part of the data's overall dimension. Multi-head attention design allows the model to capture number of relationships and patterns within the sequence simultaneously. During training, each head learns to focus on different types of dependencies within the sequence. The reshaping operation
`view(B, T, self.n_head, C // self.n_head).transpose(1, 2)` separates individual attention heads, allowing them to process information in parallel while maintaining the sequence structure. The causal masking feature creates a triangular mask that blocks future positions by setting their values to negative infinity before processing. This ensures the model only considers past and current information when making predictions, meaning the next bit prediction is based solely on previous bits. Attention weights are calculated using *scaled dot-product* attention, which involves computing similarities, scaling, applying the causal mask, normalization, and weighted combination of values.

The `MLP` component, shown in Listing 14, applies non-linear transformations to each position in the sequence independently. Its implementation follows a standard two-layer design, combining an expansion layer `c_fc`, an activation function, and a contraction layer `c_proj`. Thanks to this structure we enable the model to perform complex transformations on the attended representations from each sequence position.

```
1  class MLP(nn.Module):
2      def __init__(self, config):
3          super().__init__()
4          self.c_fc = nn.Linear(config.n_embd, 4 * config.n_embd, bias=False)
5          self.act_fn = config.act_fn
6          self.c_proj = nn.Linear(4 * config.n_embd, config.n_embd, bias=False)
7          self.c_proj.NANOGPT_SCALE_INIT = 1
8
9      def forward(self, x):
10         return self.c_proj(self.act_fn(self.c_fc(x)))
```

Listing 14. *Multi-layer perceptron implementation inside the Next-Bit transformer architecture*

Lastly, `Block` represents a complete layer within a transformer decoder, with implementation shown in Listing 16. It combines both attention mechanism and feed-forward MLP components with pre-normalization, residual connections and layer normalization features that help with stable model training. This design choice also offers a possibility to train transformer model with more layers and ensures that gradients flow more effectively throughout the network. Residual connections are a cornerstone of the architecture's success. They not only provide direct paths for gradients to move backward through the network, but also help with preserving the original input information even after number of transformations.

```python
class Block(nn.Module):
    def __init__(self, config):
        super().__init__()
        self.ln_1 = nn.LayerNorm(config.n_embd, bias=False)
        self.attn = CausalSelfAttention(config)
        self.ln_2 = nn.LayerNorm(config.n_embd, bias=False)
        self.mlp = MLP(config)

    def forward(self, x, output_attentions=False):
        if output_attentions:
            attn_out, attn_weights = self.attn(self.ln_1(x), output_attentions=True)
            x = x + attn_out
            return x, attn_weights
        else:
            x = x + self.attn(self.ln_1(x))

        x = x + self.mlp(self.ln_2(x))
        return x
```

Listing 15. *Transformer Block implementation*

5.4 Training and Experimentation Process

To evaluate the architectural design and performance of the `BinaryGPT` model, several experiments were conducted. These experiments measured how accurately model could predict the next bit in binary sequences generated by an LCG. Each model was trained for a fixed number of epochs using the same dataset and hyperparameter configuration. The number of transformer layers (`n_layer`) and the number of attention heads (`n_head`) served as the primary experimental variables.

The implementation utilizes a learning rate scheduling strategy that combines linear warmup with cosine annealing. During the initial training phase, the learning rate gradually increases from a low starting point to its highest value. This warmup period is essential because it prevents the model from making big, potentially harmful weight updates when the parameters are still randomly initialized.

After the warmup phase, the learning rate gradually decreases from its peak to a minimum value following a cosine decay schedule. The `AdamW` optimizer with weight decay regularization to further stabilize training was selected.

```python
def linear_warmup_cosine(it, lr_init, lr_min, lr_max, warmup_steps, max_steps):
    if it < warmup_steps:
        return (lr_max - lr_init) * (it+1) / warmup_steps + lr_init
    if it > max_steps:
        return lr_min
    decay_ratio = (it - warmup_steps) / (max_steps - warmup_steps)
    coeff = 0.5 * (1.0 + math.cos(math.pi * decay_ratio))
    return lr_min + coeff * (lr_max - lr_min)
```

Listing 16. *Implementation of linear warmup cosine annealing learning rate scheduler*

In addition to evaluating the model's prediction accuracy, additional metrics that provide deeper insight into the randomness of the input sequences are calculated. One of these metrics is the randomness score, which interprets the model's accuracy in the context of randomness. In other words, the closer the prediction accuracy is to 50%, the more random the sequence is assumed to be. This score offers a normalized measure to quantify unpredictability. An additional feature is the use of prediction entropy to determine how confident the model is in its predictions. High entropy suggests the model is uncertain, indicating a random sequence, while low entropy implies the model has identified predictable patterns. Model evaluation also incorporates context window analysis to examine how prediction accuracy varies with input length, revealing potential periodicity, long-range dependencies, and the optimal context size. Standard classification metrics, such as Receiver Operating Characteristic (ROC) curves, precision-recall curves, and Area Under the Curve (AUC) scores, are used to evaluate the model's predictive performance.

5.4.1 Hyperparameter Search and Optimization

The experimentation process for the `BinaryGPT` model followed a systematic approach to optimize performance on next-bit binary sequence prediction. The process began with a hyperparameter search across multiple dimensions, varying the number of Transformer layers, attention heads, and embedding dimensions. Initially, the embedding dimension was held constant while the number of layers and attention heads was varied. Subsequently, experiments explored different combinations of number of attention heads and embedding dimensions.

The model was trained on progressively more complex datasets generated by a dynamic LCG implementation, using various bit extraction strategies. Early experiments focused on LSB-mode and binary-flattened bit extraction to evaluate the model's pattern recognition capabilities across a spectrum of sequence complexities.

`BinaryGPT` successfully identified patterns ranging from simple alternating sequences of zeros and ones to the more sophisticated task of predicting the final bit in flattened binary outputs from the generator.

To assess the statistical properties of the training and test datasets, all generated binary sequences were evaluated using the NIST SP 800-22 statistical test suite. Results showed that a significant portion of the sequences exhibited non-random, pattern-based behavior. While some sequences passed as statistically indistinguishable from random, the majority did not. Despite these variations, the model consistently achieved 100% accuracy on both training and test sets within a small number of epochs. This, along with secondary analysis, confirmed that the sequences contained easily learnable structures that `BinaryGPT` was able to exploit.

Among the various bit extraction methods, MSB-only extraction produced statistically superior sequences. However, the model's accuracy dropped to around 50%, indicating performance equivalent to random guessing. These results, coupled with high randomness scores, approaching 1.00, indicate that MSB-only sequences lacked exploitable structure for next-bit prediction.

Interestingly, a higher-order bit extraction approach, using a modulus of $p = 32$ to extract the 15 most significant bits, produced sequences strong enough to pass the NIST SP 800-22 tests. On these sequences, the `BinaryGPT` model achieved up to 78% accuracy with a configuration of one layer and eight attention heads, as shown in Figure 5.2. This represents a significant improvement over the purely random performance on single MSB extraction, but still falls short of the near-perfect accuracy achieved on highly structured, non-random sequences.

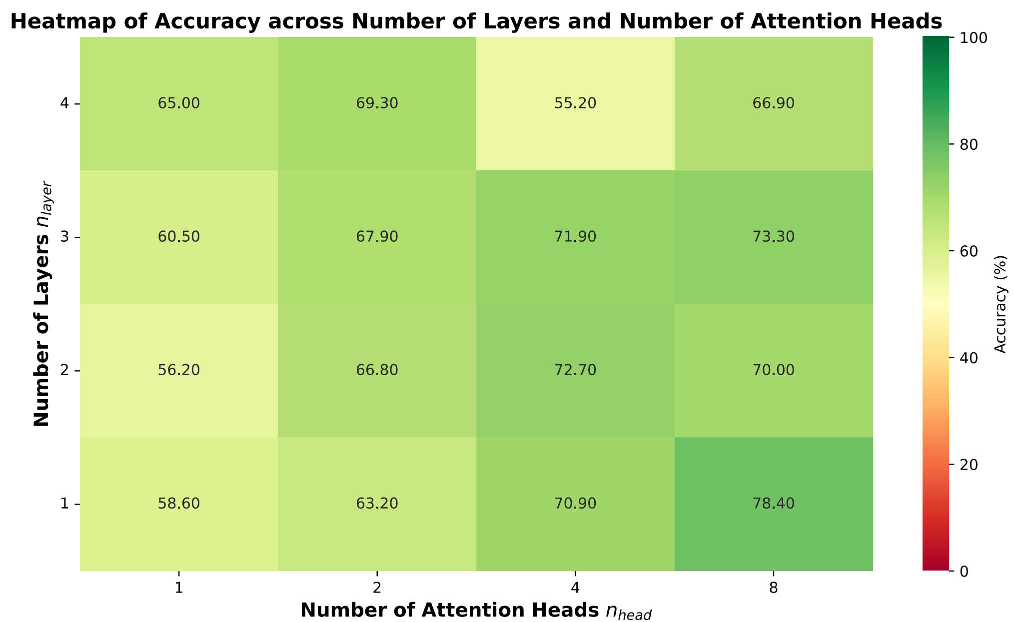

Figure 5.2. *Heatmap of Prediction accuracy across varying n_{head} and n_{embed}*

This intermediate performance suggests that while the higher-order MSBs preserved some degree of learnable structure, the increased statistical randomness limited the model's ability to extract and exploit predictable patterns.

Similar performance was observed using a modulus of $p = 16$ to extract 7 most significant bits. Binary sequences created by appending 7 MSBs into a binary sequence generated statistical p-values larger than the threshold α, resulting in most sequences being assessed as random. With training over 50 epochs, both training and test loss maintained a downward trend, as can be seen in Figure 5.3 Figure 5.4 visualizes an ROC curve for next-bit prediction, with the AUC score of 0.672.

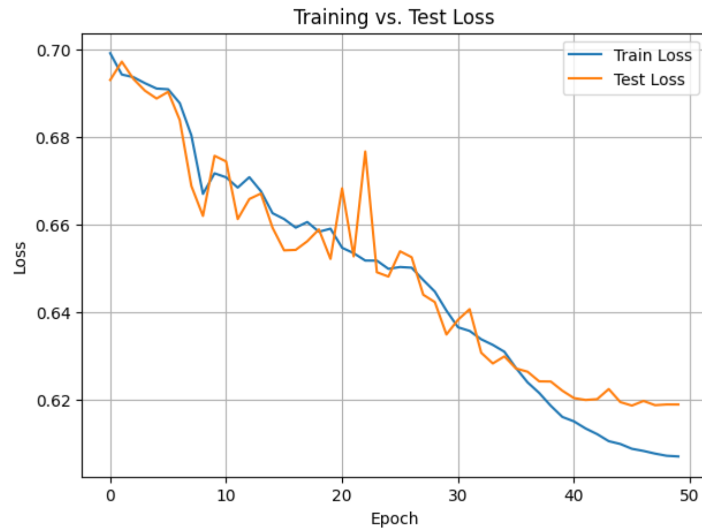

Figure 5.3. *Train and Test loss curves*

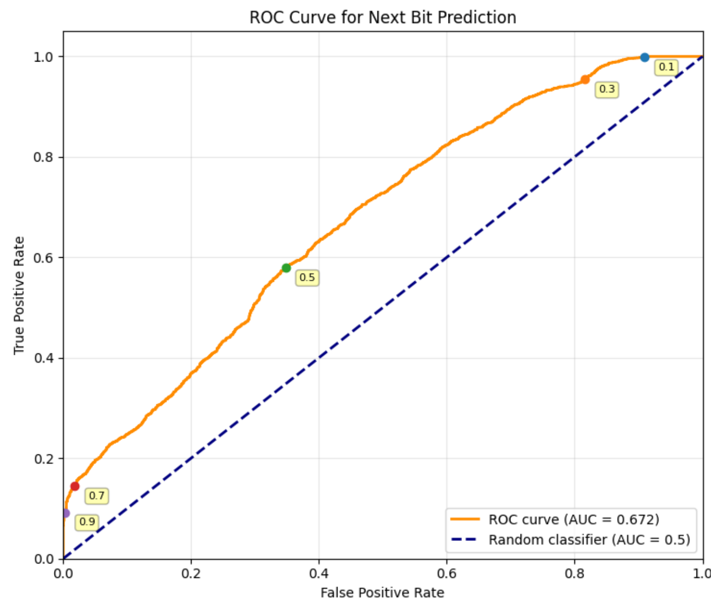

Figure 5.4. *ROC curve for next bit prediction*

This suggests the model has a moderate ability to distinguish between the positive and negative classes, corresponding to the 60% accuracy discussed above. This moderate success on partially structured, yet still random sequences can be attributed to the transformer's attention mechanism, which adapts dynamically to the available data patterns. The multiple attention heads in the optimal configuration appeared capable of identifying and focusing on the subtle structural artifacts within the higher-order bit sequences, even when these patterns were insufficient to guarantee deterministic prediction. The attention mechanism's ability to selectively weight different positions in the input sequence enabled the model to capture limited existing dependencies, resulting in performance significantly above random chance while acknowledging the constraints imposed by increased sequence randomness.

CONCLUSION

In this thesis, we explored machine learning approaches in high-entropy cryptographic settings. Firstly, by questioning whether teaching a neural network the underlying cryptographic transformations inside the hash function is feasible. To systematically explore the feasibility of learning cryptographic hash functions with neural networks, we implemented a SHA-3 toy function. This approach enabled us to understand the fundamental components of the SHA-3 algorithm and design an adjustable version with the option to create a full-size implementation.

Conducted experiments found that the choice of optimizer can significantly impact the model's performance. Across various learning rates and hidden layer sizes, the Adam optimizer consistently excelled compared to other optimizers. Larger neural networks, with 512 and 1024 neurons, achieved high accuracy even on higher complexity inputs, suggesting that neural networks can potentially learn Keccak-*p* permutation step mappings quite effectively. However, these models struggled to generalize across multiple Keccak-*f* function rounds. Although they performed well during training, their accuracy dropped significantly when tested on more challenging inputs. This indicates various limitations in capturing the patterns of cryptographic hash functions. Results from the first part opened several promising avenues for further exploration in subsequent parts of the thesis.

In the second part, the research question of whether machine learning models could replace traditional statistical tests, such as the NIST SP 800-22 test suite, has been explored. The focus shifted to using machine learning to assess binary sequences' randomness. Notably, the highest validation accuracy of 97% was achieved with a relatively small model architecture with only two hidden layers and four attention heads, which indicates that a smaller model can provide sufficient capacity to capture the relevant structure in binary sequences. Moreover, overly complex models may not be necessary for practical randomness assessment. Given the results, transformer-based models, with their advanced pattern recognition capabilities, can serve as a potential alternative or complementary tool to traditional statistical methods.

Lastly, the `BinaryGPT` model represents an advancement in the application of transformer architectures to next-bit prediction in binary sequences, particularly those generated by pseudorandom number generators like the LCG. Employing a causal self-attention mechanism, combined with the MLP component and transformer blocks, the model processed binary tokens and predicted the probability of the next bit with high accuracy.

Experiments conducted also demonstrated the model's robustness and adaptability. Notably, even when NIST classified sequences as random, our model predicted the next bit with nearly 80% accuracy, specifically when predicting the next bit in a sequence created by appending the 15 most significant bits within a 32-bit binary representation of the LCG. This result underscores the potential of Transformer-based models in high-entropy environments and their possible role as a complementary tool for randomness assessment within the NIST SP 800-22 Statistical Test Suite. If a model can reliably predict the next bit in a sequence based solely on preceding bits, this implies that the sequence contains an exploitable structure and thus may not be truly random despite passing traditional statistical randomness tests.

Future work could explore several promising research directions grounded in this thesis. Firstly, extending experiments to full-scale SHA-3 implementations could verify neural networks' ability to handle genuine cryptographic complexities. Secondly, one could investigate hybrid model architectures that combine statistical knowledge with transformer architectures to enhance the generalization and robustness of the model across diverse pseudorandom number generators. Lastly, further integration of interpretability methods into transformer-based models could significantly contribute to understanding how models recognize and predict structures in the binary domain, thus bridging theoretical cryptographic analysis with machine learning interpretability frameworks.

REFERENCES

[1] Miklós Ajtai. “The Complexity of the Pigeonhole Principle”. In: *29th Annual Symposium on Foundations of Computer Science (FOCS 1988)*. 1988, pp. 346–355. DOI: 10.1109/SFCS.1988.21951.

[2] L. Beinborn and Y. Pinter. “Analyzing Cognitive Plausibility of Subword Tokenization”. In: *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*. 2023. DOI: 10.48550/arXiv.2310.13348. URL: https://arxiv.org/pdf/2310.13348.

[3] Manabendra Nath Bera et al. “Randomness in Quantum Mechanics: Philosophy, Physics and Technology”. In: *Reports on Progress in Physics* 80.12 (2017), p. 124001. DOI: 10.1088/1361-6633/aa8731. URL: https://arxiv.org/abs/1611.02176.

[4] Guido Bertoni et al. *The KECCAK Reference, Version 3.0*. Tech. rep. Jan. 2011. URL: http://keccak.noekeon.org/Keccak-reference-3.0.pdf.

[5] Lenore Blum, Manuel Blum and Michael Shub. “Comparison of Two Pseudo-Random Number Generators”. In: *Advances in Cryptology: Proceedings of CRYPTO ’82*. Plenum Press, 1982, pp. 61–78. URL: https://link.springer.com/chapter/10.1007/978-1-4757-0602-4_5.

[6] *Byte-Pair Encoding Tokenization*. Hugging Face. 2023. URL: https://huggingface.co/learn/llm-course/chapter6/5.

[7] B. Carnahan. *Probability and Randomness*. Lecture notes. Carnegie Mellon University, Dept. of Statistics & Data Science, 2024. URL: https://www.stat.cmu.edu/~brian/201/week09/class/mon.pdf.

[8] S. Chintala. *PyTorch Strengthens Its Governance by Joining the Linux Foundation*. Sept. 12, 2022. URL: https://pytorch.org/blog/PyTorchfoundation/ (visited on 12/04/2024).

[9] Eric Conrad, Seth Misenar and Joshua Feldman. *CISSP Study Guide*. Elsevier Syngress, 2012. ISBN: 978-1-59749-961-3. DOI: 10.1016/C2011-0-07337-4.

[10] J. Devlin et al. “BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding”. In: *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics*. Minneapolis, MN, 2019, pp. 4171–4186. DOI: 10.18653/v1/N19-1423. URL: https://aclanthology.org/N19-1423/.

[11] Yevgeniy Dodis. “Randomness and Cryptography”. In: *Courant Newsletter* (2019). URL: https://cs.nyu.edu/~dodis/courant-article.pdf.

[12] A. K. Dubey and V. Jain. “Comparative Study of Convolution Neural Network’s ReLU and Leaky-ReLU Activation Functions”. In: *Applications of Computing, Automation and Wireless Systems in Electrical Engineering*. Ed. by S. Mishra, Y. Sood and A. Tomar. Vol. 553. Lecture Notes in Electrical Engineering. Singapore: Springer, 2019, pp. 863–873. DOI: 10.1007/978-981-13-6772-4_76.

[13] Raphaël Féraud and François Clérot. “A Methodology to Explain Neural Network Classification”. In: *Neural Networks* 15.2 (2002), pp. 237–246. ISSN: 0893-6080. DOI: 10.1016/S0893-6080(01)00127-7. URL: https://www.sciencedirect.com/science/article/pii/S0893608001001277.

[14] H. Ferraiolo and A. Regenscheid. *NIST SP 800-78-5: Cryptographic Algorithms and Key Sizes for Personal Identity Verification*. Tech. rep. SP 800-78-5. July 2024. DOI: 10.6028/NIST.SP.800-78-5. URL: https://doi.org/10.6028/NIST.SP.800-78-5.

[15] *FIPS PUB 180-4: Secure Hash Standard (SHS)*. Tech. rep. FIPS 180-4. Aug. 2015. DOI: 10.6028/NIST.FIPS.180-4. URL: https://doi.org/10.6028/NIST.FIPS.180-4.

[16] *FIPS PUB 202: SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions*. Tech. rep. FIPS 202. Aug. 2015. DOI: 10.6028/NIST.FIPS.202. URL: http://dx.doi.org/10.6028/NIST.FIPS.202.

[17] David Gerault et al. “How to Securely Implement Cryptography in Deep Neural Networks”. In: *Cryptology ePrint Archive* 2025.288 (2025). URL: https://eprint.iacr.org/2025/288.

[18] Aurélien Géron. *Hands-on Machine Learning with Scikit-Learn, Keras, and TensorFlow*. 2nd ed. Sebastopol, CA: O’Reilly Media, 2019. ISBN: 978-1-492-03264-9.

[19] A. Gohr. “Improving Attacks on Round-Reduced Speck32/64 Using Deep Learning”. In: *Advances in Cryptology – CRYPTO 2019*. Ed. by A. Boldyreva and D. Micciancio. Vol. 11693. Lecture Notes in Computer Science. Cham: Springer, 2019, pp. 150–179. ISBN: 978-3-030-26950-0. DOI: 10.1007/978-3-030-26951-7_6. URL: https://link.springer.com/chapter/10.1007/978-3-030-26951-7_6.

[20] Ian Goodfellow, Yoshua Bengio and Aaron Courville. *Deep Learning*. Online version. MIT Press, 2016. URL: http://www.deeplearningbook.org.

[21] *Google DeepMind*. Google DeepMind. 2025. URL: https://deepmind.google/.

[22] Gus Gutoski. *Lecture 1: Random Numbers, Coin Tossing*. Lecture notes. Perimeter Institute for Theoretical Physics, 2013. URL: https://perimeterinstitute.ca/personal/ggutoski/crypto-lec-1.pdf.

[23] A. Hambitzer et al. “NNBits: Bit Profiling with a Deep Learning Ensemble Based Distinguisher”. In: *Topics in Cryptology – CT-RSA 2023*. Ed. by M. Košílek. Vol. 13871. Lecture Notes in Computer Science. Cham: Springer, 2023, pp. 493–523. ISBN: 978-3-031-30871-0. DOI: 10.1007/978-3-031-30872-7_19. URL: https://link.springer.com/chapter/10.1007/978-3-031-30872-7_19.

[24] Alexander K. Hartmann. “Introduction to Randomness and Statistics”. In: *arXiv preprint arXiv:0910.4545* (2009), pp. 1–95. DOI: 10.48550/arXiv.0910.4545. URL: https://arxiv.org/abs/0910.4545.

[25] *How Does the Merkle-Damgård Construction Operate in the SHA-1 Hash Function?* EITCA Institute. June 15, 2024. URL: https://eitca.org/cybersecurity/eitc-is-acc-advanced-classical-cryptography/hash-functions/sha-1-hash-function/examination-review-sha-1-hash-function/how-does-the-merkle-damgard-construction-operate-in-the-sha-1-hash-function-and-what-role-does-the-compression-function-play-in-this-process/ (visited on 12/04/2024).

[26] T. E. Hull and A. R. Dobell. “Random Number Generators”. In: *SIAM Review* 4.3 (1962), pp. 230–254. ISSN: 0036-1445. DOI: 10.1137/1004061. URL: https://epubs.siam.org/doi/10.1137/1004061.

[27] L. E. Bassham III et al. *A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications*. Tech. rep. SP 800-22 Rev. 1a. Gaithersburg, MD, Apr. 2010. DOI: 10.6028/NIST.SP.800-22r1a. URL: https://doi.org/10.6028/NIST.SP.800-22r1a.

[28] S. Islam et al. “A Comprehensive Survey on Applications of Transformers for Deep Learning Tasks”. In: *arXiv preprint* (2023). DOI: 10.48550/arXiv.2306.07303. eprint: 2306.07303. URL: https://arxiv.org/pdf/2306.07303.

[29] *java.util.Random — Java Platform SE 8 Documentation*. Oracle. 2025. URL: https://docs.oracle.com/javase/8/docs/api/java/util/Random.html (visited on 05/27/2025).

[30] D. Johnston et al. *Public Comments on the Decision Proposal for SP 800-22 Rev. 1a*. Tech. rep. Comment period: 12 Jan – 14 Feb 2022. Feb. 2022. URL: https://csrc.nist.gov/csrc/media/Projects/crypto-publication-review-project/documents/decision-proposal-comments/sp800-22r1a-decision-proposal-comments-2022.pdf.

[31] Leslie P. Kaelbling, Michael L. Littman and Andrew W. Moore. “Reinforcement Learning: A Survey”. In: *Journal of Artificial Intelligence Research* 4 (1996), pp. 237–285. URL: https://www.jair.org/index.php/jair/article/view/10166.

[32] Burt Kaliski. “Quadratic Residuosity Problem”. In: *Encyclopedia of Cryptography and Security*. Ed. by H. C. A. van Tilborg and S. Jajodia. Springer, 2011, p. 1003. ISBN: 978-1-4419-5905-8. DOI: 10.1007/978-1-4419-5906-5_429. URL: https://link.springer.com/referenceworkentry/10.1007/978-1-4419-5906-5_429.

[33] *Keras: The Python Deep Learning API*. Keras Team. 2024. URL: https://keras.io (visited on 12/04/2024).

[34] C. Khanna. *Word, Subword, and Character-Based Tokenization: Know the Difference*. 2021. URL: https://towardsdatascience.com/word-subword-and-character-based-tokenization-know-the-difference-ea0976b64e17.

[35] Donald E. Knuth. *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*. 3rd ed. Reading, MA: Addison-Wesley, 1997, pp. 10–26. ISBN: 978-0-201-89684-8.

[36] Pierre L'Ecuyer and Richard Simard. "TestU01: A C Library for Empirical Testing of Random Number Generators". In: *ACM Transactions on Mathematical Software* 33.4 (2007), p. 22. DOI: 10.1145/1268776.1268777. URL: https://doi.org/10.1145/1268776.1268777.

[37] *Learning*. Cambridge Dictionary. 2024. URL: https://dictionary.cambridge.org/dictionary/english/learning (visited on 11/02/2024).

[38] Shiguo Lian, Jianhua Sun and Zhen Wang. "Secure Hash Function Based on Neural Network". In: *Neurocomputing* (2006). DOI: 10.1016/j.neucom.2006.04.003.

[39] Shiguo Lian et al. "Hash Function Based on Chaotic Neural Networks". In: *Proceedings of the 2006 IEEE International Symposium on Circuits and Systems (ISCAS)*. 2006, p. 4. DOI: 10.1109/ISCAS.2006.1692566.

[40] C. John Mann. "Randomness in Nature". In: *GSA Bulletin* 81.1 (1970), pp. 95–104. DOI: 10.1130/0016-7606(1970)81[95:RIN]2.0.CO;2. URL: https://pubs.geoscienceworld.org/gsa/gsabulletin/article/81/1/95/6744/Randomness-in-Nature.

[41] George Marsaglia. *Diehard: A Battery of Tests of Randomness*. Florida State University, Department of Statistics. 1995. URL: https://ani.stat.fsu.edu/diehard/.

[42] *math — Mathematical Functions — Python Documentation*. Python Software Foundation. 2025. URL: https://docs.python.org/3/library/math.html (visited on 05/27/2025).

[43] A. J. Menezes, P. C. van Oorschot and S. A. Vanstone. *Handbook of Applied Cryptography*. Boca Raton: CRC Press, 1996. ISBN: 978-0-8493-8523-0. URL: https://cacr.uwaterloo.ca/hac/.

[44] S. J. Mielke et al. "Between Words and Characters: A Brief History of Open-Vocabulary Modeling and Tokenization in NLP". In: *arXiv preprint* (2021). DOI: 10.48550/arXiv.2112.10508. eprint: 2112.10508. URL: https://arxiv.org/pdf/2112.10508.

[45] T. M. Mitchell. *Machine Learning*. Online edition. McGraw-Hill (digitized archive), 1997. URL: https://archive.org/details/machine-learning-tom-mitchell_202402.

[46] *NumPy — The Fundamental Package for Scientific Computing with Python*. NumPy. 2025. URL: https://numpy.org (visited on 05/27/2025).

[47] Christof Paar and Jan Pelzl. *Understanding Cryptography: A Textbook for Students and Practitioners*. 2nd ed. Heidelberg: Springer-Nature, 2024. ISBN: 978-3-662-69006-2. DOI: 10.1007/978-3-662-69007-9.

[48] S. K. Park and K. W. Miller. "Random Number Generators: Good Ones Are Hard to Find". In: *Communications of the ACM* 31.10 (Oct. 1988), pp. 1192–1201. ISSN: 0001-0782. DOI: 10.1145/63039.63042. URL: https://www.firstpr.com.au/dsp/rand31/p1192-park.pdf.

[49] A. Patil and M. Rane. "Convolutional Neural Networks: An Overview and Its Applications in Pattern Recognition". In: *Information and Communication Technology for Intelligent Systems*. Ed. by T. Senjyu et al. Vol. 195. Smart Innovation, Systems

and Technologies. Singapore: Springer, 2021, pp. 31–46. ISBN: 978-981-15-7077-3. DOI: 10.1007/978-981-15-7078-0_3.

[50] W. H. Payne, J. R. Rabung and T. P. Bogyo. “Coding the Lehmer Pseudo-Random Number Generator”. In: *Communications of the ACM* 12.2 (Feb. 1969), pp. 85–86. ISSN: 0001-0782. DOI: 10.1145/362946.362960. URL: https://www.firstpr.com.au/dsp/rand31/p85-payne.pdf.

[51] I. Provilkov. *BPE-Dropout: Official Implementation*. 2019.

[52] I. Provilkov, D. Emelianenko and E. Voita. “BPE-Dropout: Simple and Effective Subword Regularization”. In: *arXiv preprint* (2020). DOI: 10.48550/arXiv.1910.13267. eprint: 1910.13267. URL: https://arxiv.org/pdf/1910.13267.

[53] *Python Programming Language*. Python Software Foundation. 2024. URL: https://www.python.org (visited on 12/04/2024).

[54] *PyTorch: An Open Source Machine Learning Framework*. PyTorch. 2024. URL: https://pytorch.org (visited on 12/04/2024).

[55] *Randomness*. Cambridge Dictionary. 2025. URL: https://dictionary.cambridge.org/dictionary/english/randomness.

[56] A. W. Schrift and A. Shamir. “On the Universality of the Next Bit Test”. In: *Advances in Cryptology – CRYPTO ’90*. Vol. 537. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 1991, pp. 394–408. DOI: 10.1007/3-540-38424-3_29. URL: https://link.springer.com/chapter/10.1007/3-540-38424-3_29.

[57] *Scikit-learn: Machine Learning in Python*. scikit-learn. 2024. URL: https://scikit-learn.org/stable/ (visited on 12/04/2024).

[58] *Summary of the Tokenizers*. Hugging Face. 2025. URL: https://huggingface.co/docs/transformers/en/tokenizer_summary.

[59] *SymPy — Symbolic Mathematics in Python*. SymPy. 2025. URL: https://www.sympy.org/ (visited on 05/27/2025).

[60] *TensorFlow: An End-to-End Open Source Machine Learning Platform*. Google Brain Team. 2024. URL: https://www.tensorflow.org (visited on 12/04/2024).

[61] *Tensors — PyTorch Documentation*. PyTorch. 2024. URL: https://pytorch.org/docs/stable/tensors.html (visited on 12/04/2024).

[62] *torch.nn — PyTorch Documentation*. PyTorch. 2024. URL: https://pytorch.org/docs/stable/nn.html (visited on 12/04/2024).

[63] *torch.nn.BCELoss — PyTorch Documentation*. PyTorch. 2024. URL: https://pytorch.org/docs/stable/generated/torch.nn.BCELoss.html (visited on 12/04/2024).

[64] *torch.nn.BCEWithLogitsLoss — PyTorch Documentation*. PyTorch. May 27, 2025. URL: https://pytorch.org/docs/stable/generated/torch.nn.BCEWithLogitsLoss.html.

[65] *torch.nn.CrossEntropyLoss — PyTorch Documentation*. PyTorch. 2024. URL: https://pytorch.org/docs/stable/generated/torch.nn.CrossEntropyLoss.html (visited on 12/04/2024).

[66] *torch.nn.Embedding — PyTorch Documentation*. PyTorch. May 27, 2025. URL: https://pytorch.org/docs/stable/generated/torch.nn.Embedding.html.

[67] *torch.nn.KLDivLoss — PyTorch Documentation*. PyTorch. 2024. URL: https://pytorch.org/docs/stable/generated/torch.nn.KLDivLoss.html (visited on 12/04/2024).

[68] *torch.nn.LayerNorm — PyTorch Documentation*. PyTorch. May 27, 2025. URL: https://pytorch.org/docs/stable/generated/torch.nn.LayerNorm.html.

[69] *torch.nn.Linear — PyTorch Documentation*. PyTorch. 2024. URL: https://pytorch.org/docs/stable/generated/torch.nn.Linear.html (visited on 12/04/2024).

[70] *torch.nn.ModuleList — PyTorch Documentation*. PyTorch. May 20, 2025. URL: https://pytorch.org/docs/stable/generated/torch.nn.ModuleList.html.

[71] *torch.nn.MSELoss — PyTorch Documentation*. PyTorch. 2024. URL: https://pytorch.org/docs/stable/generated/torch.nn.MSELoss.html (visited on 12/04/2024).

[72] *torch.nn.NLLLoss — PyTorch Documentation*. PyTorch. 2024. URL: https://pytorch.org/docs/stable/generated/torch.nn.NLLLoss.html (visited on 12/04/2024).

[73] *torch.nn.ReLU — PyTorch Documentation*. PyTorch. 2024. URL: https://pytorch.org/docs/stable/generated/torch.nn.ReLU.html (visited on 12/04/2024).

[74] *torch.nn.SmoothL1Loss — PyTorch Documentation*. PyTorch. 2024. URL: https://pytorch.org/docs/stable/generated/torch.nn.SmoothL1Loss.html (visited on 12/04/2024).

[75] *torch.optim.Adadelta — PyTorch Documentation*. PyTorch. 2024. URL: https://pytorch.org/docs/stable/generated/torch.optim.Adadelta.html (visited on 12/04/2024).

[76] *torch.optim.Adagrad — PyTorch Documentation*. PyTorch. 2024. URL: https://pytorch.org/docs/stable/generated/torch.optim.Adagrad.html (visited on 12/04/2024).

[77] *torch.optim.Adam — PyTorch Documentation*. PyTorch. 2024. URL: https://pytorch.org/docs/stable/generated/torch.optim.Adam.html (visited on 12/04/2024).

[78] *torch.optim.AdamW — PyTorch Documentation*. PyTorch. May 27, 2025. URL: https://pytorch.org/docs/stable/generated/torch.optim.AdamW.html.

[79] *torch.optim.NAdam — PyTorch Documentation*. PyTorch. 2024. URL: https://pytorch.org/docs/stable/generated/torch.optim.NAdam.html (visited on 12/04/2024).

[80] *torch.optim.RMSprop — PyTorch Documentation*. PyTorch. 2024. URL: https://pytorch.org/docs/stable/generated/torch.optim.RMSprop.html (visited on 12/04/2024).

[81] *torch.optim.SGD — PyTorch Documentation*. PyTorch. 2024. URL: https://pytorch.org/docs/stable/generated/torch.optim.SGD.html (visited on 12/04/2024).

[82] M. Turčaník and M. Javurek. “Hash Function Generation by Neural Network”. In: *Proceedings of the 2016 New Trends in Signal Processing (NTSP)*. 2016, pp. 1–5. DOI: 10.1109/NTSP.2016.7747793.

[83] *Unigram Tokenization*. Hugging Face. 2023. URL: https://huggingface.co/learn/llm-course/en/chapter6/7.

[84] Ivan Vasilev et al. *Python Deep Learning*. 2nd ed. Packt, 2019. ISBN: 978-1-78934-846-0.

[85] A. Vaswani et al. “Attention Is All You Need”. In: *Advances in Neural Information Processing Systems 30 (NIPS 2017)*. Long Beach, CA, 2017. DOI: 10.48550/arXiv.1706.03762. URL: https://arxiv.org/pdf/1706.03762.

[86] *What Is a Neuron?* Queensland Brain Institute. 2024. URL: https://qbi.uq.edu.au/brain/brain-anatomy/what-neuron (visited on 12/04/2024).

[87] *What Is a Transformer Model?* IBM Think Blog. Mar. 28, 2025. URL: https://www.ibm.com/think/topics/transformer-model.

[88] *What Is Backpropagation*. IBM. 2024. URL: https://www.ibm.com/think/topics/backpropagation (visited on 12/04/2024).

[89] *What Is Supervised Learning?* IBM. 2024. URL: https://www.ibm.com/topics/supervised-learning (visited on 11/10/2024).

[90] *What Is Unsupervised Learning?* Google Cloud. 2024. URL: https://cloud.google.com/discover/what-is-unsupervised-learning (visited on 11/10/2024).

[91] *WordPiece Tokenization*. Hugging Face. 2023. URL: https://huggingface.co/learn/llm-course/en/chapter6/6.

[92] A. C. Yao. “Theory and Applications of Trapdoor Functions”. In: *23rd Annual Symposium on Foundations of Computer Science (FOCS 1982)*. IEEE, 1982, pp. 80–91. URL: https://www.di.ens.fr/users/phan/secuproofs/yao82.pdf.

[93] G. Yenduri et al. “Generative Pre-trained Transformer: A Comprehensive Review on Enabling Technologies, Potential Applications, Emerging Challenges, and Future Directions”. In: *arXiv preprint* (2023). DOI: 10.48550/arXiv.2305.10435. eprint: 2305.10435. URL: https://arxiv.org/pdf/2305.10435.