DESIGN AND IMPLEMENTATION OF A PORTABLE HARDWARE DEVICE FOR ANALYZE SIDE CHANNEL ATTACKS

Prathapage Chamika Kusal Piyathillaka

IT21195716

Bsc (Hons) Degree In Information Technology
Specializing In Cyber Security

Department Of Computer System Engineering

Sri Lanka Institute of Information Technology
Sri Lanka

April 2025

DECLARATION

I declare that this is my own work, and this dissertation does not incorporate without acknowledgement any material previously submitted for a degree or diploma in any other university or Institute of higher learning and to the best of my knowledge and belief it does not contain any material previously published or written by another person except where the acknowledgement is made in the text.

Also, I hereby grant to Sri Lanka Institute of Information Technology the non-exclusive right to reproduce and distribute my dissertation in whole or part in print, electronic or other medium. I retain the right to use this content in whole or part in future works (such as articles or books).

Signature: [illegible] Date: 09/04/2025

Signature of the Supervisor: [illegible] Date: 09/04/2025

ABSTRACT

In modern days of cybersecurity, cryptographic systems are typically considered secure based on the strength of their algorithms. However implementation-level vulnerabilities typically render them vulnerable to practical attacks. This research explores the exploitation of time based side-channel information for deducing or approximating DES (Data Encryption Standard) encryption keys using two complementary methods. The first method targets the vulnerability presented by timestamp based key generation. It is shown that if DES keys are produced by pseudo-random number generators seeded from Unix timestamps an attacker with approximate guesses of the key generation time window can reproduce a pool of candidate keys. Using byte-wise and bit-wise similarity metrics such as Euclidean distances the closest matching keys can be identified from the reproduced pool and exposing inherent weaknesses of predictable entropy sources.

The second approach discusses the application of deep learning which is Long Short-Term Memory (LSTM) network in predicting the DES key bits from encryption timing and side-channel features. A database of encryption operations was created in a controlled setup, gathering mean encryption time, Hamming weight and Hamming distance for various key-plaintext pairs. The LSTM model trained showed remarkable ability in reconstructing DES key patterns from timing-driven inputs and establishing that fine differences in execution time can uncover useful cryptographic information.The second approach discusses the application of deep learning which is Long Short Term Memory (LSTM) network in predicting the DES key bits from encryption timing and side-channel features. A database of encryption operations was created in a controlled setup, gathering mean encryption time, Hamming weight and Hamming distance for various key plaintext pairs. The LSTM model trained showed remarkable ability in reconstructing DES key patterns from timing driven inputs and establishing that fine differences in execution time can uncover useful cryptographic information.

Both techniques were integrated in a portable demo system on a Raspberry Pi, prioritizing accessibility, real-time feedback and pedagogical usefulness. The results confirm that time based properties ,when not appropriately addressed can break the confidentiality of symmetric encryption, requiring constant-time implementations and secure randomness in cryptographic protocols.

Keywords: Side Channel Attack, Timing Analysis, DES, LSTM, Cryptography, Raspberry Pi, Key Prediction, Timestamp Vulnerability, Machine Learning, Information Security

ACKNOWLEDGEMENT

I would like to thank Mr. Kavinga Yapa my supervisor, first for his unfaltering guidance, support and encouragement during the course of this research project. His expertise, valuable comments and tireless motivation helped in steering the course of my work and negotiating difficulties at every stage.

I am also very grateful to my co-supervisor, Mr. Amila Senaratne for his creative suggestions, technical guidance and continuous support. His contributions were a key factor in helping me to formalize my thoughts and translate theoretical principles into practical applications.

Finally I would like to express my gratitude to the academic and technical staff of the department for creating a conducive and intellectually stimulating environment. I would like to express my special thanks to my fellow students and colleagues who have provided inspiration, support, and friendship throughout this journey.

This project would not have been possible without the contributions of all these individuals and I remain sincerely thankful to each of them.

LIST OF CONTENTS

DECLARATION **2**

ABSTRACT **3**

ACKNOWLEDGEMENT **4**

LIST OF CONTENTS **5**

LIST OF FIGURES **8**

LIST OF ABBREVIATIONS **9**

INTRODUCTION **10**

Overview **10**

Background Literature **11**

Research Gap **14**

Research Problem **17**

Research Objectives **19**

Main objective 19

Sub-objectives 19

METHODOLOGY **21**

Timestamp Based Analysis **21**

System diagram 21

Key generation 21

Step 1: choosing the right tools 21

Step 2: writing the code 22

Step 3: converting date and time to timestamp 22

Step 4: generating keys based on timestamps 22

Step 5: user input and execution flow 23

Analysis 24

Step 1: setting up the environment and libraries 24

Step 2: loading the generated keys 25

Step 3: comparing keys by byte match 25

Step 4: highlighting similarity in the output 26

Step 5: finding the closest key using binary euclidean distance 26

Step 6: finding the top 20 most similar keys 27

Step 7: find most closest key 28

Step 8: security warning and conclusion 28

Machine Learning Model To Predict Key Bits **29**

System diagram 29

Dataset generation....30
Step 1: environment setup....30
Step 2: timing data collection....30
Step 3: feature engineering....32
Step 4: dataset preparation....33
Training the model....33
Transition from random forest to lstm: motivation and results....33
Why lstm and the model selection rationale....33
Step 1: data loading and preprocessing....34
Step 2: feature normalization....35
Step 3: padding and masking....36
Step 4: train-test split....36
Step 5: model architecture....37
Step 7: compilation and training....37
Prediction phase of lstm-based des key bit....38
Step 1: loading trained model and scaling parameters....38
Step 2: preprocessing user inputs (plaintext, time, key)....38
Step 3: normalization of features....39
Step 4: model inference and prediction....40
Step 5: evaluating the prediction....40
Step 6: visualizing feature impact....41

Hardware Integration....42
Step 1: system setup and deployment workflow....42
Step 2: installing the operating system....42
Step 3: remote access configuration....42
Step 4: installing required libraries....43
Step 5: lcd screen integration....43

Commercialization Aspect....44
Timestamp-based key generation and analysis – commercialization aspect....44
Lstm-based key bit prediction – commercialization aspect....44

Testing & Implementation....45
Implementation....45
Timestamp-based key analysis implementation....45
Lstm-based key prediction implementation....46
Hardware integration on raspberry pi....46
Testing....47
Unit testing....47
Integration testing....47
Performance testing....48
Model feature evaluation....48

RESULTS & DISCUSSION....49

Results....49
Method 1: timestamp-based key generation and analysis....49

Method 2: lstm-based des key bit prediction 50

Research Findings 53

Effectiveness of timestamp-based generation & analysis 53

Role of side-channel timing in key inference 53

Application of deep learning to side-channel analysis 53

Practicality and integration in low-resource systems 54

Broader implications for cryptographic awareness and education 54

Discussion 55

Practical implications of time-based DES key inference 55

Issues and limitations 55

Ethical and security concerns in applied cryptanalysis 56

Future directions in timing-based side-channel research 56

SUMMARY OF STUDENT'S CONTRIBUTION 57

CONCLUSION 58

REFERENCES 60

LIST OF FIGURES

Figure 1-System diagrame for 1st method 21
Figure 2- Convert user input time and date to a timestamp 22
Figure 3-Generating DES keys based on Timestamp 23
Figure 4-Getting user input 24
Figure 5- Calculate character level similarity 25
Figure 6-Highlight matching characters 26
Figure 7-Convert byte to Binary 26
Figure 8-Convert key into Binary 26
Figure 9-Applying Euclidean Distance 27
Figure 10-Equation for Euclidean Distance 27
Figure 11-Finding top 20 keys 27
Figure 12-Finding most closes Key 28
Figure 13-Setting threshold for warning 29
Figure 14-System diagram for 2nd method 29
Figure 15-These two lines are from the pycryptodome library 30
Figure 16-Setting plaintext and encryption key 31
Figure 17-Measure time for encryption 31
Figure 18-Finding hamming weights and hamming distance 32
Figure 19-Imported libraries 34
Figure 20-Convert to lists of integers 35
Figure 21-Preparing features 35
Figure 22-Convert time to micro seconds 35
Figure 23-Padding 36
Figure 24-split dataset for training 37
Figure 25-Model architecture 37
Figure 26-Preprocessing inputs 38
Figure 27-Normalize the time 39
Figure 28-Combining features 40
Figure 29-Code for prediction 40
Figure 30-Generated DES keys 49
Figure 31-Selected top 20 keys 49
Figure 32-Final output (Closest key) 50
Figure 33-Generated dataset through control environment 50
Figure 34-Accuracies and losses 51
Figure 35-Graph for features 52
Figure 36-Final output (Predicted key) 52

LIST OF ABBREVIATIONS

Abbreviation	Full Form
SCA	Side-Channel Attack
DES	Data Encryption Standard
LSTM	Long Short-Term Memory
ML	Machine Learning
AI	Artificial Intelligence
PRNG	Pseudo-Random Number Generator
UI	User Interface
LCD	Liquid Crystal Display
CPU	Central Processing Unit
RNG	Random Number Generator
HWD	Hamming Weight and Distance
OSS	Open Source Software
VM	Virtual Machine
API	Application Programming Interface
IoT	Internet of Things
AES	Advanced Encryption Standard
CNN	Convolutional Neural Network
GUI	Graphical User Interface
ROC	Receiver Operating Characteristic
RNN	Recurrent Neural Network
CSV	Comma-Separated Values

INTRODUCTION

Overview

Symmetric encryption algorithms such as Advanced Encryption Standard (AES) and Data Encryption Standard (DES) are widely used today to protect data in various digital systems. These algorithms are widely used in applications ranging from secure communications to financial transactions. There these algorithms maintain their confidentiality and integrity [1]. As attack techniques have become more sophisticated by now, traditional cryptographic defenses are increasingly challenged by side-channel attacks, particularly exploiting time information [2].

Timing analysis-based side channel attack is a vulnerability that by measuring the time that take to perform a cryptographic operation then by analyzing that time trying to extract sensitive information like cryptographic keys and plaintexts [3]. Traditional timing analysis techniques often face limitations due to their reliance on static data sets and lack of integration with other physical leakages such as power consumption and can't do it easily [4]. As encryption algorithms continue to evolve, the need for more sophisticated tools that can perform multiple analyzes to uncover their subtle weaknesses becomes increasingly apparent.

As response for these challenges in this project propose development of a portable hardware device designed to do time analysis-based side channel attacks on Symmetric encryption. This device is trying to do this time analysis-based side channel attack in two different ways. The first method is to generate a time-critical data set and filter keys using Euclidean distance and train a machine learning model to predict key bits by analyzing time and other features like hamming weight and hamming distances. By integrating these approaches this device aims to provide a more accurate and comprehensive assessment of symmetric key strength thereby identifying exploitable vulnerabilities in real-world scenarios.

As response for these challenges in this project propose development of a portable hardware device designed to do time analysis-based side channel attacks on Symmetric encryption. This device is trying to do this time analysis-based side channel attack in 3 different ways. The first method is to generate a time-critical data set and filter keys using Euclidean distance, correlate time and power consumption data for improved analysis, and train a machine learning model to identify time-varying patterns. By integrating these approaches this device aims to provide a more accurate and comprehensive assessment of symmetric key strength, thereby identifying exploitable vulnerabilities in real-world scenarios.

Background Literature

Side-channel attacks (SCAs) have emerged as one of the most practical threats to cryptographic systems that exploiting their physical or observable implementation characteristics rather than attacking the mathematical foundations of encryption algorithms. Among the diverse types of SCAs like power analysis EM attacks and acoustic cryptanalysis, timing attacks are one of the most easily available and effective because they are non-invasive and can be performed remotely in software environments. Time-based side-channel attacks study the time required for the cryptographic computations to deduce secret information such as key material. This review of the literature presents the history of timing analysis during its initial years and its application to symmetric key algorithms such as DES and machine learning-based side-channel modeling and the nonexistence of functional, lightweight demonstration tools in the literature.

The idea of using execution time to extract cryptographic secrets was originally presented by Paul C. Kocher in his now classic paper, Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS and Other Systems [5]. In this paper Kocher showed how small timing differences during modular exponentiation operations would divulge private keys in public-key cryptosystems. This breakthrough led to a wave of interest in exploring similar vulnerabilities in other cryptographic primitives including symmetric ciphers such as DES and AES. Kocher's research opened the way for future research and placed timing attacks on the map as a viable and threatening attack vector in actual systems.

As the subject evolved, scientists started using timing analysis on block ciphers especially in cases where some execution branches, table lookups or loop iterations varied based on input data or a secret key. In particular Boneh and Brumley generalized the theory in 2003 by demonstrating a successful remote timing attack on OpenSSL's RSA implementation via a network [6]. Though their attention was still on public-key schemes, they used their results to strengthen the power of feasibility in timing leaks on network-accessible systems and highlight the practicality of Kocher's theoretical model. At the same time, work was also being done to investigate timing leaks in symmetric cryptoschemes with minimal actual demonstrations.

Symmetric encryption algorithms such as DES (Data Encryption Standard) and AES (Advanced Encryption Standard) are usually deterministic in nature. However depending on how they are implemented especially in software-based systems and variations in execution time can still occur. These can be due to variations in CPU caching, branch conditions or memory access delay particularly key scheduling or S-box substitution operations. In DES slight variations in the Hamming distance between plaintext or ciphertext inputs and between key and plaintext can alter the number of bit transitions and this alters slightly the encryption time. Though these are slight and hard to measure with typical tools, recent machine learning methods provide a strong tool for learning and detecting these patterns from noisy data.

The application of machine learning (ML) and deep learning to side-channel analysis has been on the rise in recent years. Traditional ML techniques had previously employed decision trees, k-nearest neighbors and support vector machines (SVM) as outlined by Lerman, Bontempi and Markowitch in their comparative analysis of ML algorithms used in power analysis attacks [7]. These performed well when it was feasible to work with high-quality side-channel traces, especially power traces. However with the progress of the domain, researchers began exploring deep learning models such as convolutional neural networks (CNNs) and recurrent neural networks (RNNs) like Long Short-Term Memory (LSTM) models, to analyze more intricate and sequential side-channel data.

Specifically Zaid et al. proposed the ASCAD dataset and proved that CNNs could be superior to traditional approaches in profiling AES implementation attacks [8]. Although CNNs are optimal for spatial data such as power traces and LSTMs are optimal when dealing with temporal data like sequences of timing in which input values possess a sequence and relationship with time. Most of the prior research has relied on power consumption traces acquired with high-resolution oscilloscopes which restricts their reproducibility and applicability. Very few attacks have been proposed based on timing information and deep learning and even fewer have investigated such attacks on DES particularly in embedded systems.

For deterministic key generation attacks the use of system clocks or timestamps as sources of randomness has been criticized for decades in cryptology for making systems insecure. Systems seeded by random number generators (RNGs) based on predictable factors like the current system time at the time have particularly simple windows to brute-force if the window of timestamps is guessable or known. Yilek et al. revealed one in the Debian OpenSSL whereby predictable entropy caused millions of identical keys to be produced in actual systems [9]. Biryukov et al. further examined predictable random number generation for TLS session keys [10]. However the previous work has the tendency to be focused on public-key schemes and secure socket implementations with our project being particularly extending the timestamp vulnerability technique to symmetric key generation for DES a less-used vector.

At the hardware level, platforms such as Raspberry Pi have become common across cybersecurity education, embedded system research and IoT applications with minimal resources. As extensively used as they are no evidence of side-channel demonstration research is established to prove live timing attacks on cryptographic processes executed on

Raspberry Pi. Most academic studies use high end lab equipment or simulated environments which are not representative of what students or entry-level attackers might encounter. In my integration of timestamp-based key generation analysis and LSTM-based timing prediction onto a Raspberry Pi platform is another contribution to the application of SCA learning and experimentation.

Further though there exist some research papers with promising results on artificial or clean data sets. There has been no effort to show live prediction of cryptographic keys from noisy, system-level timing data and machine learning in actual deployments. In my work it's recording timing data over numerous rounds of encryption, designing Hamming based features, training a deep LSTM model and deploying the model to predict DES key bits. This fills the void between academic research and actual, embedded system cryptanalysis.

Last but not least no or few teaching materials or research frameworks integrate both deterministic modeling (timestamp-based key extraction) and machine learning-based prediction from timing information in one easy to use framework. Available frameworks are either narrowly scoped (to AES under power side-channeling) or mostly theoretical without practical demonstrations. This research closes this gap by providing a modular, reproducible, and hardware-friendly framework that demonstrates two weak implementations which is weak randomness and timing leakage on DES encryption.

Research Gap

Side-channel analysis (SCA) has been for a long time one of the most prominent attack vectors on cryptographic systems and there have been a number of works showing how adversaries can use physical information like power consumption, electromagnetic (EM) emissions or timing behavior to reveal secret keys or internal states. Timing-based side-channel attacks are still one of the most accessible and subtle to carry out particularly in software-level attacks where no physical access to hardware is necessary. Despite being conceptually straightforward and existing since Kocher's seminal research on timing attacks back in 1996. The research community still lacks comprehensive, modern and practical frameworks that demonstrate real-time machine learning-based key recovery using time as the primary leakage vector and particularly in resource-constrained environments such as Raspberry Pi-based systems.

One of the earliest and most referenced works in timing analysis is by Paul C. Kocher who first demonstrated how variations in the time required to perform cryptographic operations could leak information about private keys in public key algorithms like RSA and Diffie-Hellman [5]. Although this research laid the groundwork for timing-based SCA which focus in subsequent research in the field turned to implementations of high value cryptosystems and public key system attacks. Later studies extended the technique to symmetric key algorithms such as AES and DES and although most efforts remained theoretical or focused on very specific implementations in controlled environments. Kocher's research revealed the possibility of timing data betraying cryptographic data yet there were relatively few systems that could practically take advantage of such weaknesses in contemporary software or embedded systems.

Most of the literature has been concerned with electromagnetic side-channel attacks and power analysis, like the ones proposed by Mangard et al. In their paper regarding Differential Power Analysis (DPA) and Simple Power Analysis (SPA) [11]. Though very effective in practice, these attacks tend to be susceptible to needing sophisticated hardware equipment, oscilloscopes, probes and extensive signal processing knowledge. This has made power based SCAs more suitable for advanced academic or nation state attackers but less so for novice researchers, students or attackers in constrained environments. On the other hand timing analysis particularly with software based tools remains more accessible to attackers with constrained resources. Yet there is a low degree of open source or educational software exploiting time based side-channel weaknesses in symmetric cryptography resulting in a need for practical training and illustrative research.

Some research has proposed the use of machine learning (ML) for side-channel analysis especially for power traces of AES implementations. For example Lerman et al. presented a comparative study of ML classifiers applied to side-channel traces. which shows that models like Support Vector Machines (SVM) and Random Forests can outperform traditional statistical attacks in some cases [7]. Likewise Zhang et al. explored the use of deep learning models such as Convolutional Neural Networks (CNNs) and Long Short-Term Memory (LSTM) networks for key extraction from side-channel information [10]. However much of these studies focus on high resolution power traces and custom signal acquisition setups. There is a clear lack of attention to the use of sequence models like LSTM on timing side-channel data. Which is typically noisier and harder to exploit due to lower resolution and additional environmental noise.

Another one is current timing analysis tools and methods are largely specialized, sophisticated, and expertise-intensive in cryptographic theory and side-channel attack techniques. They are usually for experts and are thus very challenging to manage by someone who does not possess wide knowledge in the area [12]. Formal attack trees have been used for example by some studies to analyze time-based attacks [13]. But this complexity diminishes the practical use of timing analysis in situations where ease of use is the most important. Moreover current timing analysis methods are not consistent in their accuracy, yielding inconsistent results. Data acquisition may be inaccurate because of filtering and matching issues which resulting in inconsistent results [6]. For instance earlier studies acquired confidential information by measuring how long it takes for a system to respond. In these instances the risk of measurement error in measuring these responses is present, and therefore better methods and greater proficiency are required when carrying out timing analysis.

In particular for DES encryption most timing attack research has since become obsolete as DES is already widely considered deprecated in high security applications. However DES remains a valuable teaching algorithm in cryptography education and is still present in legacy systems, embedded software and low-cost IoT devices. Therefore research updating attack techniques for DES using modern tools such as deep learning or data driven methods is still limited. Traditional attacks on DES use brute force or statistical cryptanalysis while side-channel attacks on DES are power/EM-based or theoretical implementations. This creates a gap for real-world, real-time and easily replicable experiments using only timing information to infer DES key bits particularly in a controlled but realistic setup.

Furthermore, most recent SCA research targets large datasets or laboratory-grade equipment. They rarely address how time based attacks could be deployed in lightweight or real world hardware such as Raspberry Pi, Arduino or low-cost microcontrollers. These platforms are increasingly used in IoT and edge computing where timing-based vulnerabilities are even more exploitable due to lack of secure hardware design. Most academic frameworks stop at theoretical demonstrations or simulations and do not provide hardware embedded proof-of-concept models that students or researchers can deploy to understand the threat landscape in real-time. This highlights a major research and educational gap in showing how real cryptographic key leakage can happen even in unsophisticated hardware due to poorly implemented time-dependent cryptographic operations.

Additionally though some timestamp-based key generation vulnerabilities have been addressed in the context of PRNGs and session token generation and few papers cover timestamp-based DES key generation as an independent vulnerability. For example, research such as that of Yilek et al. and Biryukov et al. has shown how low entropy in random number generators can result in predictable keys or session tokens [9] [14]. However these typically apply to RSA key generation or TLS session keys. My work reveals a novel and understudied vulnerability in symmetric key encryption by demonstrating how user defined or predictable timestamps can generate repeatable DES keys vulnerable to reconstruction.

Also though numerous researchers have created cryptographic simulation platforms but few of them feature real-time machine learning prediction, timestamp-based deterministic key generation and live visualization on resource constrained hardware such as Raspberry Pi. The idea of predicting DES key bits from encrypted timing data, Hamming weight/distance features and sequence based learning models and then visualizing results in real time via an LCD screen or web interface is still unexplored. This represents a practical gap in demonstrating attacks that not only work but are accessible to those with limited resources such as university students, entry-level penetration testers and security researchers.

Therefore the key research gap this project aims to address is the lack of practical, real-time frameworks and educational tools that demonstrate timing based side-channel attacks on symmetric key algorithms like DES using modern machine learning techniques. Although timing attacks are well understood in theory their practical realization especially in low-resource situations is often not addressed in current research. In addition the combination of deterministic vulnerability modeling (timestamp-based key generation) and deep learning-based inference models (LSTM prediction based on timing and Hamming features) in a single deployable system is new and addresses a critical gap in research and education.

By running the system on a Raspberry Pi and offering full functionality without the need for external monitors or expensive hardware this project serves immediately to further the applied cryptography, side-channel research and cybersecurity education communities. It not only advances research into real world exploitability of timing information but also provides a scalable, demonstrable and reproducible tool for classroom or lab use. The system's dual emphasis on both deterministic timestamp based vulnerability modeling and modern ML-based inference is a key step towards making side-channel attack research more practical, feasible and efficient.

Research Problem

Cryptographic systems they theoretically secure but often become vulnerable in real world applications due to implementation-level flaws. The most common category of such mistakes is side-channel leakage. Where the adversary acquires information based on the physical or behavioral properties of a system for example power consumption, electromagnetic radiation or running time to deduce secret information. Among these timing based side-channel attacks are particularly threatening because they can often be executed purely through software which not requiring physical access to the target device. Despite being found over two decades ago timing attacks remain an underestimated threat in contemporary cryptographic implementations particularly in low-cost, embedded and educational systems.

The research problem this work is set to address involves the applied exploitation of timing data towards recovering and analyzing DES encryption keys for two insecure cases. First one is deterministic key generation from timestamps and the second one is predictive key inference based on machine learning as well as encryption time and trained features. Both methods mention basically different attack vectors which one against poor key generation practices and the other against execution-time side-channel leakage by data driven modeling. However both converge on a common and critical issue. The failure to account for timing predictability as a viable attack surface in symmetric encryption systems.

In real-world deployments particularly on resource limited devices or older systems such as embedded controllers or Raspberry Pi nodes developers usually make use of easily predictable or low-entropy seeds such as system time to seed encryption keys. This introduces a dangerous predictability that undermines the core principle of cryptographic randomness. If an attacker can estimate or guess the time interval on which a key was generated they can resimulate the key generation process and significantly narrow the key search space. While this vulnerability is conceptually well-known there is little modern systemized implementation or tool that demonstrates how timestamp based DES keys can be regenerated and matched with high similarity especially in environments accessible to students, researchers or entry-level penetration testers.

At the same time another emerging issue is the increasingly observable relationship between encryption time and amount of data, particularly in software-based cryptosystems lacking sufficient countermeasures like constant-time operation. Although DES encryption execution time is generally consistent, subtle variations can still reveal information due to interactions between plaintext and key. These fluctuations though hard to be exploited by conventional statistical techniques could be acquired by current deep learning algorithms like Recurrent Neural Networks (RNNs) such as LSTM well designed to identify patterns in time-series or sequence data. However there are extremely few current best practice systems that apply LSTM-based algorithms to forecast DES key bits from side-channel timing information and Hamming-based features.

The absence of publicly accessible, educationally tractable and experimentally verified systems illustrating real-time key inference from timing behavior and weak randomness alone is a serious research and teaching gap. This project directly addresses that gap by developing and integrating two complementary systems one that simulates timestamp-based key generation vulnerabilities and another that uses an LSTM model to predict DES key bits from timing data into a lightweight Raspberry Pi-powered demonstration platform. This not only demonstrates the viability of timing-based attacks in resource constrained settings but also points to the relatively uncharted territory of the intersection of machine learning and side-channel analysis with time being the major source of leakage.

Therefore the overall research problem is How can time-domain side-channel information like timestamp-based key generation and execution time of encryption operations be manipulated using deterministic modeling and machine learning to guess or estimate DES encryption keys in real-time on low resource platforms? Resolving this issue has repercussions for the development of secure cryptographic implementations and strengthening side-channel resistance in symmetric primitives and constructing educational tools bridging theoretical cryptographic weaknesses with real-world exploitability.

Research Objectives

The main goal of this study is to investigate and determine the practicability of utilizing time-based side-channel data to recover or predict DES encryption keys through the use of two new methods. They are timestamp-based deterministic key generation analysis and deep learning-based key bit prediction using timing data. By integrating these techniques into a practical, lightweight system using a Raspberry Pi this research aims to bridge the gap between theoretical side-channel vulnerabilities and their real-world applications particularly in educational, research and lightweight cryptographic environments.

This project is targeting the essential but often underrated vulnerabilities of time based leakage in symmetric crypto schemes highly pertinent with the prevalence of insecure or outdated cryptographic implementations utilized in embedded and IoT devices. As these devices are increasingly presen and it becomes more important to understand how timing behaviors leak sensitive information. Our work not only shows experimental attacks that take advantage of these vulnerabilities but also delivers a modular, reusable experimentation, education and research platform for future work in this new field.

Main objective

The main objective of this research is to design, implement, and validate a real-time side-channel analysis system capable of predicting or reconstructing DES encryption keys using timestamp-based key generation and timing analysis through deep learning model.

Sub-objectives

1. To apply a timestamp-based key generation model to mimic the way in which cryptographic systems utilizing deterministic or weakly seeded PRNGs produce predictable DES keys. The aim is to demonstrate that, if the time interval utilized in key generation is known or approximated, an attacker can replicate a very close approximation of the original encryption key.
2. To implement an engine that compares a user-provided DES key to a set of timestamp-based keys using byte-level matching and binary Euclidean distance. This module illustrates how attackers determine "close" keys and quantify key predictability in terms of statistical and visual measures.
3. To create a dataset of encryption times, keys, and plaintexts for simulated environments, to quantify properties like Hamming weights, Hamming distance, and average encryption time in several runs. The dataset is used to train a deep model to identify patterns of timing-based leakage.
4. To develop and train a deep learning model, namely an Long Short-Term Memory (LSTM) network, that is able to predict DES encryption key bits from timing data

and engineered side-channel features with high predictive performance and good generalization on unseen inputs.

5. To combine both the timestamp analysis and LSTM prediction models into a single hardware system utilizing a Raspberry Pi. The system should be able to accept users' plaintext, timing, and key input, conduct analysis or prediction, and provide the result in real-time using an LCD or GUI screen.
6. To test the performance, accuracy, and usability of the system through extensive testing, such as unit, integration, and performance testing. The aim is to make sure that the system performs well under limited environments and that the predictions are accurate and interpretable.
7. To illustrate the system's learnability and utilitarian significance by designing it to be light-weight, modular, and reproducible in order to facilitate extension in the future to teaching or research. This involves keeping procedures documented and readying the system for extended application to future encryption algorithms like AES.

METHODOLOGY

Timestamp Based Analysis

System diagram

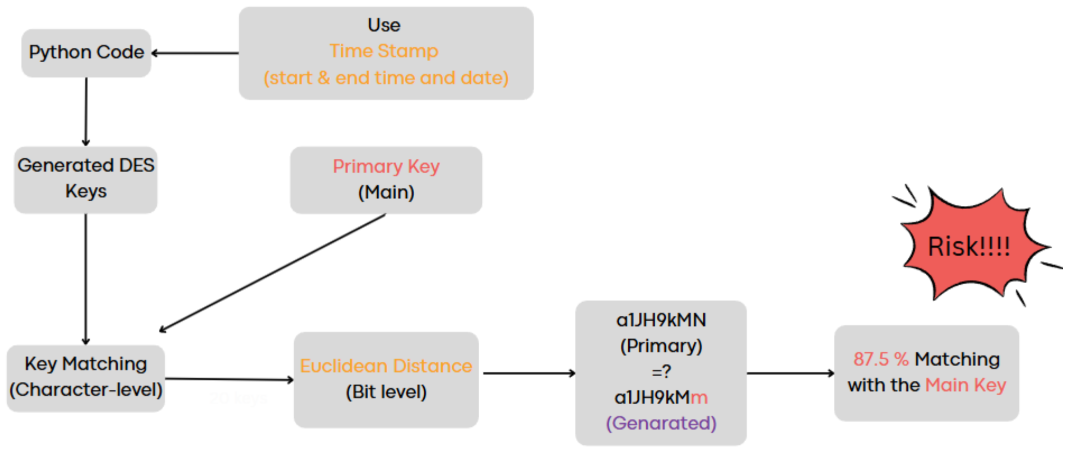


Figure 1-System diagram for 1st method

Key generation

In this project I am implemented a vulnerable key generation system using timestamps. The idea was to explore how predictable keys could become if generated on user defined or input time ranges. In here I describe in full detail how I implemented this method using Python and which libraires were I used and why and how the logic flow from input to key generation and file output.

Step 1: choosing the right tools

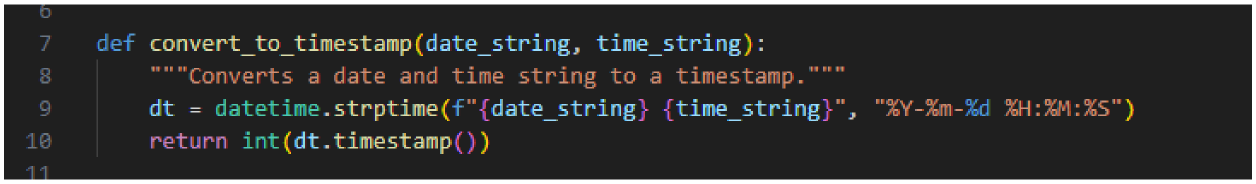
Before writing any code I had to decide on the tools and libraires that would support for this process. I chose python to implement the key generator due to its simplicity, readability and powerful built in libraries. First libraires that I used I *Random*. I used python's *random* module to generate pseudo-random numbers. Specifically I took advantage of *random.seed()* to initialize the generator using a timestamp. This choice makes the system intentionally flawed which is ideal for later vulnerability analysis. Another one is *Datatime*. Handling date and time input is a critical part of this project. I used the *datatime* module to parse human readable date time inputs and convert them into Unix timestamps. This conversion is vital because timestamps are numeric and can serve as seeds for the random key generator.

Step 2: writing the code

I started the code by defining some constants and setting up the environment for file output. This includes the key size and the output file path.

The key size is fixed to 8 bytes (64 bits). I set the key size to 8 bytes because I am conducting this analysis against DES encryption which uses a similar key length. This size allows for a balance of simplicity and effectiveness while being large enough to mimic a real encryption key and small enough to remain human readable when converted to hexadecimal.And the file path points to the location where the keys will be saved. I used a raw string with *r""* to handle windows style backslashes correctly. While not the most portable approach this was sufficient for a single system test.

Step 3: converting date and time to timestamp



```
def convert_to_timestamp(date_string, time_string):
    """Converts a date and time string to a timestamp."""
    dt = datetime.strptime(f"{date_string} {time_string}", "%Y-%m-%d %H:%M:%S")
    return int(dt.timestamp())
```

Figure 2- Convert user input time and date to a timestamp

So the next part is timestamp convention. This function (fig 2) takes two inputs *date_string* for date and a *time_string* to represent time. it first combines them into a single string and passes this string to *datetime.strptime()* to parse it into a *datetime* object. From there *dt.timestamp()* is used to convert this datetime object into a Unix timestamp. It is an integer representing the number of seconds that have passed since January 1, 1970 [15].

Step 4: generating keys based on timestamps

Once I got the start and end timestamps the next step is to generate cryptographic keys which mean DES encryption keys. This is where the actual time analysis happens in this method. For each timestamp between start and end values I create a random 8-byte key. Because the number of bytes is defined by the *keysize* variable. The *random.seed(t)* function initializes the random number generator with the timestamp *t* (fig 3). Its ensuring that for every timesamp a different sequence of random numbers is produced. This randomness is crucial for generating keys because the same timestamp always produces the same sequence. And it also introduces predictability if the attacker can guess or know the timestamp range.

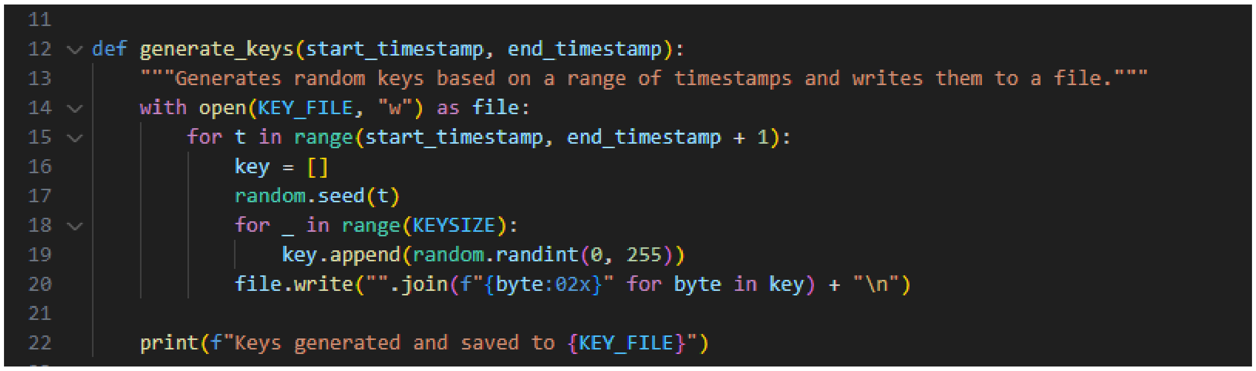

```python

def generate_keys(start_timestamp, end_timestamp):
    """Generates random keys based on a range of timestamps and writes them to a file."""
    with open(KEY_FILE, "w") as file:
        for t in range(start_timestamp, end_timestamp + 1):
            key = []
            random.seed(t)
            for _ in range(KEYSIZE):
                key.append(random.randint(0, 255))
            file.write("".join(f"{byte:02x}" for byte in key) + "\n")

    print(f"Keys generated and saved to {KEY_FILE}")
```

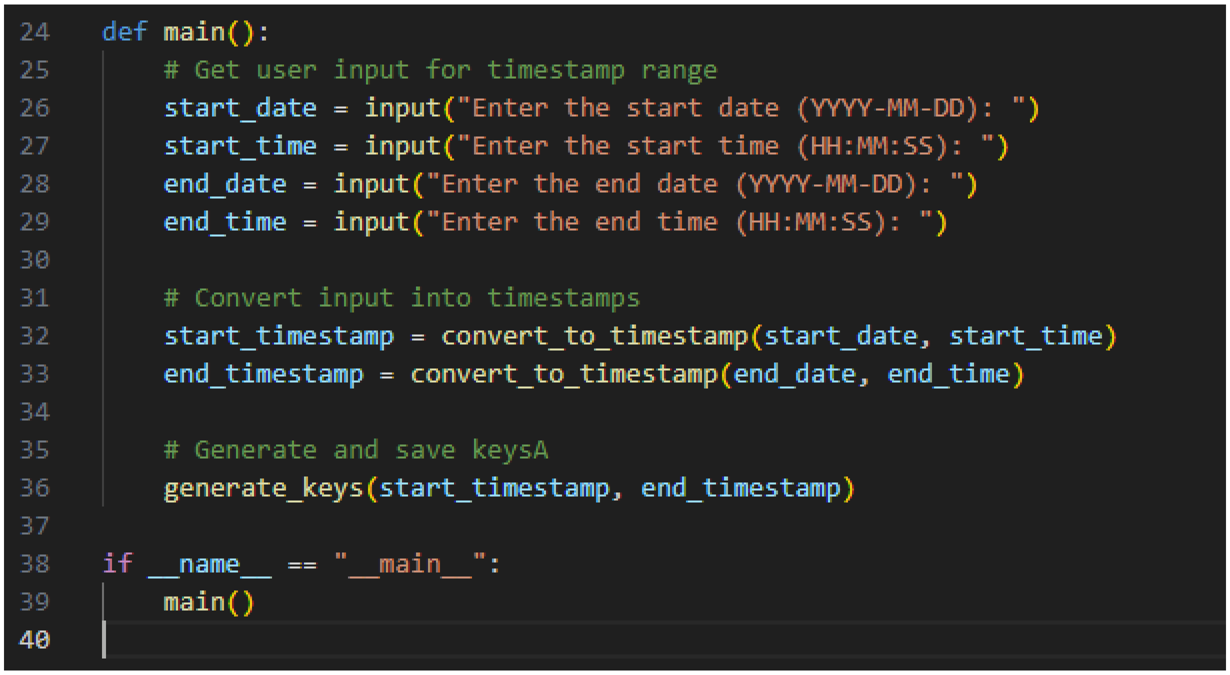
Figure 3-Generating DES keys based on Timestamp

I used the *random.randint(0, 255)* method to generate random bye values for the key. Each iteration of the loop produces a random bye (a number between 0 and 255) and these bytes are appended to the *key* list. After generating all the bytes for the key I converted them into a hexadecimal string using a formatted string: *f"{byte:02x}"*. This ensures that the byte values are represented in a two-digit hexadecimal format for an example if, a3 etc. Finally the generated key is written to a file. In here in my code it is *e.txt* with each key o a new line.

One of the crucial design choice here is how the generated keys are saved to a file. Because storing keys in a text file makes it easy to analyze them later. Its enables the comparison of generated keys against user input during the analysis process of the project. This step also aligns with the goal of simulating a system that relies on timestamp based key generations which can contain inside of a cryptographic systems.

Step 5: user input and execution flow

The final part (fig 4) of this process involves talking user input for the timestamp range and then calling the *generate_keys()* function to produce the keys. the user is prompted to enter the start and end dates and the corresponding times which are converted to Unix timestamps and passed to the key generation function.



```python
def main():
    # Get user input for timestamp range
    start_date = input("Enter the start date (YYYY-MM-DD): ")
    start_time = input("Enter the start time (HH:MM:SS): ")
    end_date = input("Enter the end date (YYYY-MM-DD): ")
    end_time = input("Enter the end time (HH:MM:SS): ")

    # Convert input into timestamps
    start_timestamp = convert_to_timestamp(start_date, start_time)
    end_timestamp = convert_to_timestamp(end_date, end_time)

    # Generate and save keysA
    generate_keys(start_timestamp, end_timestamp)

if __name__ == "__main__":
    main()

```

Figure 4-Getting user input

Int his *main()* function the user provides the start and end dates along with the times. And then it passed to the *convert_to_timestamp()* function. The resulting timestamps are used to generate the keys which are saved to the file specified by *KEY_FILE.*The reason this method of key generation based on timestamps was selected is primarily for simplicity and demonstration. However this approach is inherently flawed from a security standpoint. It is use predictable values (timestamps) to initialize the random number generator which exposes it to key regeneration attacks if the attacker can guess the timestamp range. This predictability is what makes the system insecure which is the focus on this project analysis phase.

Analysis

In this section of the project I focused on analyzing the generated cryptographic keys based on their similarity to a user provided key. This analysis involves two components character level similarity and binary Euclidean distance. I used some libraires and mathematical concepts to measure how closely the keys match the user input and to determine the security level of a given key. Below is a comprehensive explanation of how the code functions and why I chose to implement it this way.

Step 1: setting up the environment and libraries

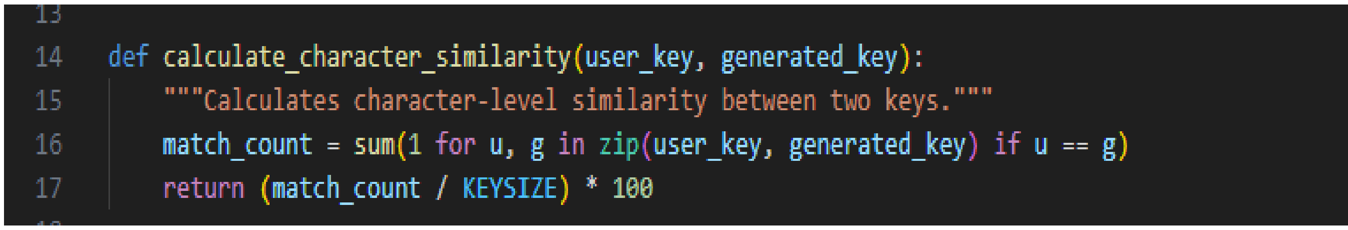
So first I imported three main libraries, *math, re and matplotlib.pyplot. math* library provides mathematical functions necessary for calculations like Euclidean distance and the calculation of binary similarities. The most important function I used from *math* is *sqrt()* which helps calculate the Euclidean distance between two binary representations of keys.And then *re* library is used for regular expressions matching. I used it to validate the format of the user provided key. The input should be exactly 16 hexadecimal characters corresponding to an 8-byte key. Regular expressions ensure that the input format strictly adheres to this specification and preventing errors during further analysis. *Mataplotlib.pyplot* this library is used for plotting visualizations. I used it to plot the character level similarities of the top 20 closest keys and provide a clear graphical representation of how similar the generated keys are to the user provided key. With these libraries in place I start to code the core functionality of the keys analysis.

Step 2: loading the generated keys

The first function I wrote is *load_keys()*. It loads the previously generated keys from the file *e.txt* in current code and converts them from hexadecimal stings to byte arrays. The keys in the file are written in hexadecimal format so I split each line into pairs of characters which means each representing a byte. Then convert each pair into an integer. I used a list comprehension to achieve this convention which makes the process both compact and efficient.

Step 3: comparing keys by byte match

The next task is to compute the character level similarity between the user input key and each generated key. This is done by comparing the corresponding byte of the two keys and counting how many bytes match.



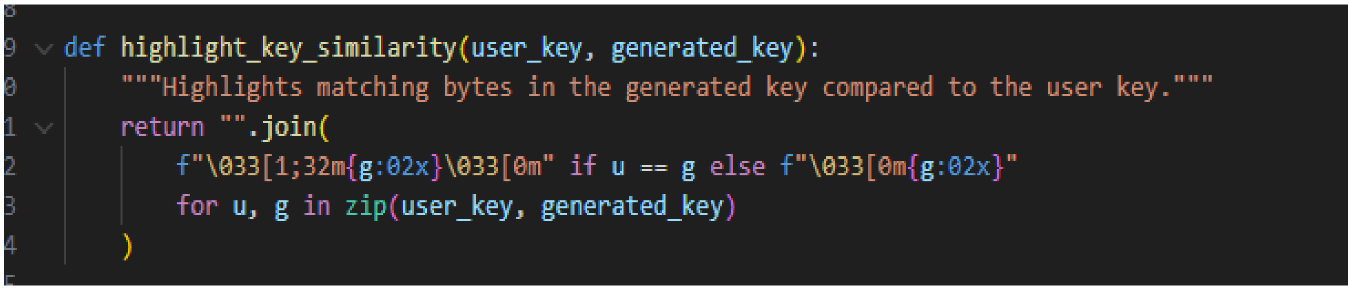
```python
def calculate_character_similarity(user_key, generated_key):
    """Calculates character-level similarity between two keys."""
    match_count = sum(1 for u, g in zip(user_key, generated_key) if u == g)
    return (match_count / KEYSIZE) * 100
```

Figure 5- Calculate character level similarity

In this code I used *zip()* to pair up the corresponding bytes from the two keys. Then I counted how many of those byes match using the *sum()* function which is a pythonic way of counting true conditions. Then the results similarity score is calculated of matching byes as the percentage relative to the total number of bytes in the key which is defined by *keysize.*

Step 4: highlighting similarity in the output

To make it more user friendly I implemented the *highlight_key_similarity()* function. This function highlights the matching byte in green using ANSI escape sequences and leaves the non-matching bytes in their default color.

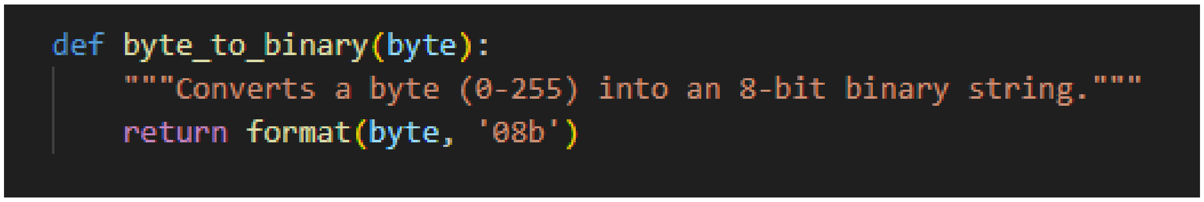


```python
def highlight_key_similarity(user_key, generated_key):
    """Highlights matching bytes in the generated key compared to the user key."""
    return "".join(
        f"\033[1;32m{g:02x}\033[0m" if u == g else f"\033[0m{g:02x}"
        for u, g in zip(user_key, generated_key)
    )
```

Figure 6-Highlight matching characters

Step 5: finding the closest key using binary euclidean distance

In the next step I needed to measure the similarity between the keys at the binary level. To do this I first converted each byte of a key into its 8 bit binary representation using the *byte_to_binary()* function as shown in figure 7.

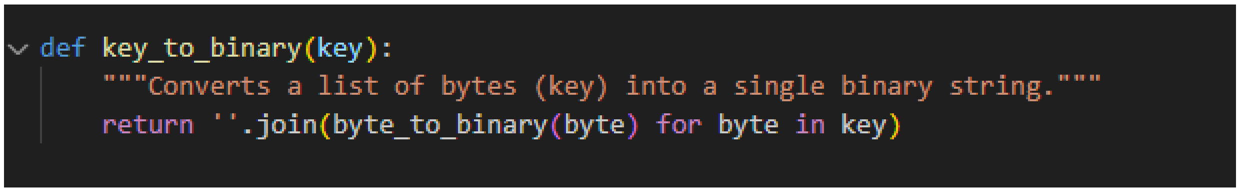


```python
def byte_to_binary(byte):
    """Converts a byte (0-255) into an 8-bit binary string."""
    return format(byte, '08b')
```

Figure 7-Convert byte to Binary

In here I used the *format()* function with the format specifier ‘08b’ to convert the byte into 8-bit binary string. So it ensures that each byte is represented with exactly eight digits.

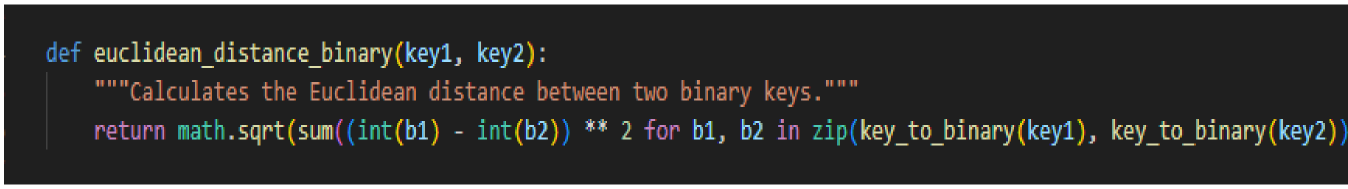
Once the bytes are converted into binary strings I combined them into a single binary string fo the entire key using the *key_to_binary()* function. It is shown in figure 8.



```python
def key_to_binary(key):
    """Converts a list of bytes (key) into a single binary string."""
    return ''.join(byte_to_binary(byte) for byte in key)
```

Figure 8-Convert key into Binary

So now I have binary representation of both keys with me so then I calculated the Euclidean distance between them to measure their similarity in binary level to improve the accuracy.



```python
def euclidean_distance_binary(key1, key2):
    """Calculates the Euclidean distance between two binary keys."""
    return math.sqrt(sum((int(b1) - int(b2)) ** 2 for b1, b2 in zip(key_to_binary(key1), key_to_binary(key2))
```

Figure 9-Applying Euclidean Distance

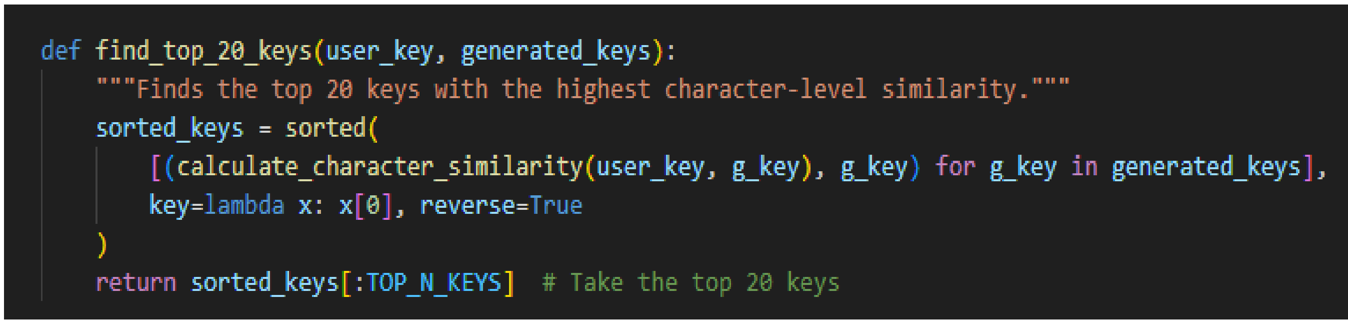
Euclidean distance is a standard measure of distance in multi dimensional spaces [16]. In my case it calculates how far apart the binary representations of the two keys are. A smaller distance indicates higher similarity while a larger distance suggests a more significant difference between the keys. The equation shows in figure 10.

$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

Figure 10-Equation for Euclidean Distance

The next function is *find_top_20_keys()*. I created this for identify the top 20 most similar keys based on character level similarity. And I used python's built in function called *sorted()* to sort the keys by similarity score in descending order and returned the top 20.

Step 6: finding the top 20 most similar keys



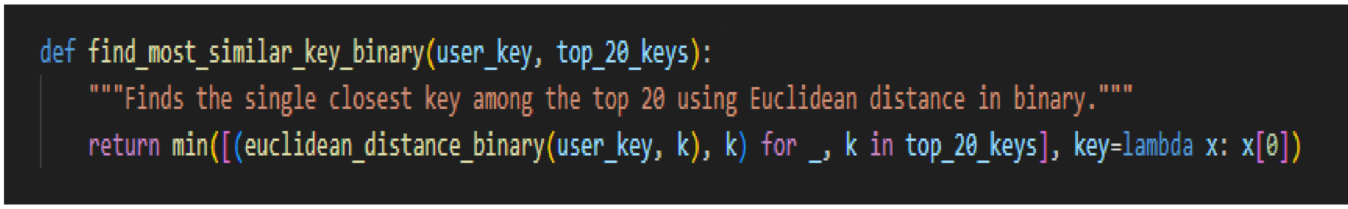
```python
def find_top_20_keys(user_key, generated_keys):
    """Finds the top 20 keys with the highest character-level similarity."""
    sorted_keys = sorted(
        [(calculate_character_similarity(user_key, g_key), g_key) for g_key in generated_keys],
        key=lambda x: x[0], reverse=True
    )
    return sorted_keys[:TOP_N_KEYS]  # Take the top 20 keys
```

Figure 11-Finding top 20 keys

After this to identify the most closest key to the main key or user input I wanted to go a step deeper and evaluate them at the binary level. For that I implemented *find_most_similar_key_binary()* function. My goal of this function is to take those 20 candidates and pinpoint the one that is closest to the user key when we compare their binary forms using Euclidean distance. The reason for doing this is that even if 2 keys have a similar number of matching characters bit their actual bit patterns might be quite different. Binary level checks give a more granular, accurate picture of similarity.

Step 7: find most closest key

In here for each key *k* I called the function *euclidean_distance_binary(user_key, k)* which gives me the binary level distance between the user key and that candidate key. I paired each distance with tis corresponding key as a tuple like *(distance, key)*. So then by using *min(..., key=lambda x: x[0])* I selected the tuple with the smallest distance which mean the most similar key athe the binary level.



```
def find_most_similar_key_binary(user_key, top_20_keys):
    """Finds the single closest key among the top 20 using Euclidean distance in binary."""
    return min([(euclidean_distance_binary(user_key, k), k) for _, k in top_20_keys], key=lambda x: x[0])
```

Figure 12-Finding most closes Key

Step 8: security warning and conclusion

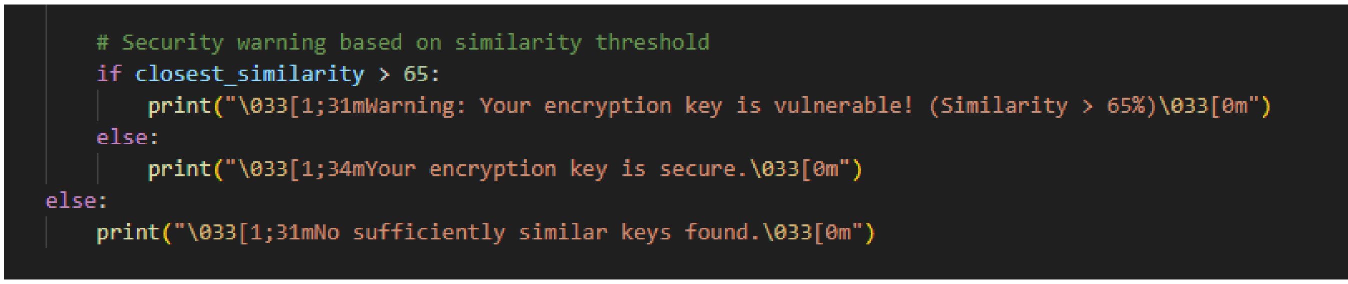
Finally I visualized the similarities using a bar chart to show the relative similarity of the top 20 keys. This was done using *matplotlib.pyplot* which is a powerful plotting library. This function generated a bar chart where each bar represents the character level similarity of one of the top 20 keys providing a clear visual representation of how close each key is to the users input.

Then the *main()* function is the heart of the script. This is where the entire key comparison process happened. It handles everything from loading keys to visualizing similarities and giving security feedback.

I started by this function by loading all previously generated keys from a file in this example its *e.txt.* And converting them form hex strings into lists of integers in bytes. This is essential for meaningful comparison later on. Then I asked the user to enter their own encryption key in hexadecimal format (16 hex characters = 8 bytes). This input serves as the baseline for our similarity check. To ensure the integrity of the input I validated the format using a regular expression. Its checks for exactly 16 valid hexadecimal characters. If the user enters anything else a red error message is shown and the function exits.

Now I know the input is valid. Next I convert the hex string into a list of byes. This allows byte wise and bitwise operations later in the comparison. Then I used the earlier defined function to find the 20 keys that are most similar to the user's key based on character level similarity. Then for each of these 20 keys I printed a line showing the key itself with matching bytes highlighted in green and the percentage of how much it matches the user's key. This helps visually understand which keys are dangerously similar. Then I calculated the closest key at the binary level using Euclidean distance. This is more fine grained than the character level check because it considers differences in individual bits. And to make this distance more understandable I converted it into a percentage similarity. I compared the actual distance to the maximum possible binary distance to get a normalized value.

Finally based on a threshold of 65% this output a security warning. If the similarity is above that the key seemed unsafe because it's too close to an already generated one. Otherwise the user gets a green signal.



```
    # Security warning based on similarity threshold
    if closest_similarity > 65:
        print("\033[1;31mWarning: Your encryption key is vulnerable! (Similarity > 65%)\033[0m")
    else:
        print("\033[1;34mYour encryption key is secure.\033[0m")
else:
    print("\033[1;31mNo sufficiently similar keys found.\033[0m")
```

Figure 13-Setting threshold for warning

Machine Learning Model To Predict Key Bits

System diagram

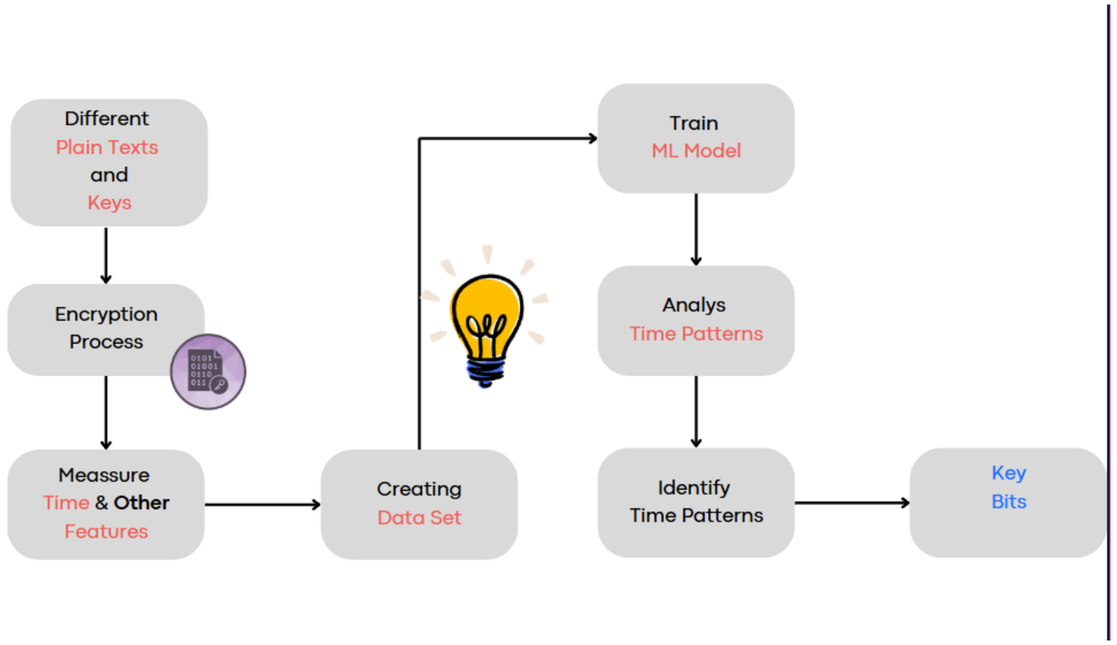


Figure 14-System diagram for 2nd method

Dataset generation

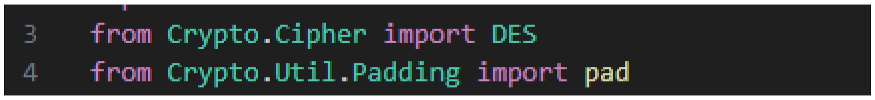
This method investigates the feasibility of predicting key bits of a DES encryption system using machine learning model. So first approach involves generating a dataset under tightly controlled experimental conditions. And extracting relevant cryptographic and statistical features and traing a machine learning model to learn the underlying patterns that relate execution time to encryption key properties. In here this methodology divided into four main stages. They are environment setup, timing data collection, feature engineering and dataset preparation.

Step 1: environment setup

So first part is for setup the control environment. To ensure consistency and reduce noise in timing measurements all data collection was performed in a highly control virtual environment. For this I used a windows 10 virtual machine with a Ryzen 5 processor and limit the RAM to 8GB and a dedicated NVIDIA GTX 1660TI Gpu. I intentionally isolated this environment. During the execution of the encryption scripts all background processes and non-essential services were stopped. Only the python encryption process was allowed to run. This minimized the impact of scheduling anomalies, caching effects and operating system noise on the timing data. So thereby improving the fidelity of the dataset.

Step 2: timing data collection

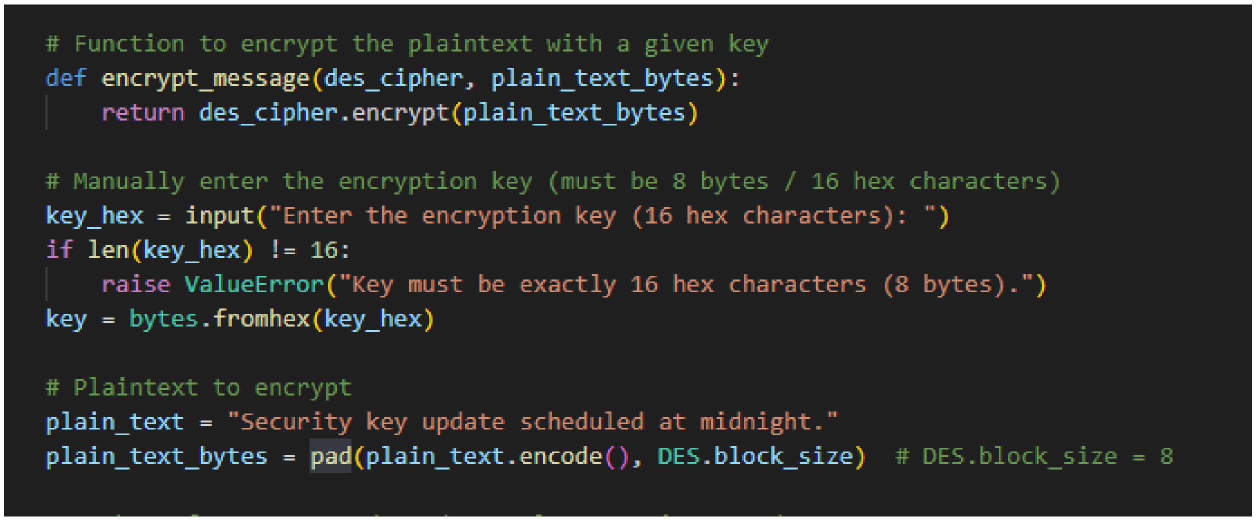
So the next part is collection of timing data. To measure encryption times I developed a python based script using *pycryptodome* library for DES encrtyption.



```
from Crypto.Cipher import DES
from Crypto.Util.Padding import pad
```

Figure 15-These two lines are from the pycryptodome library

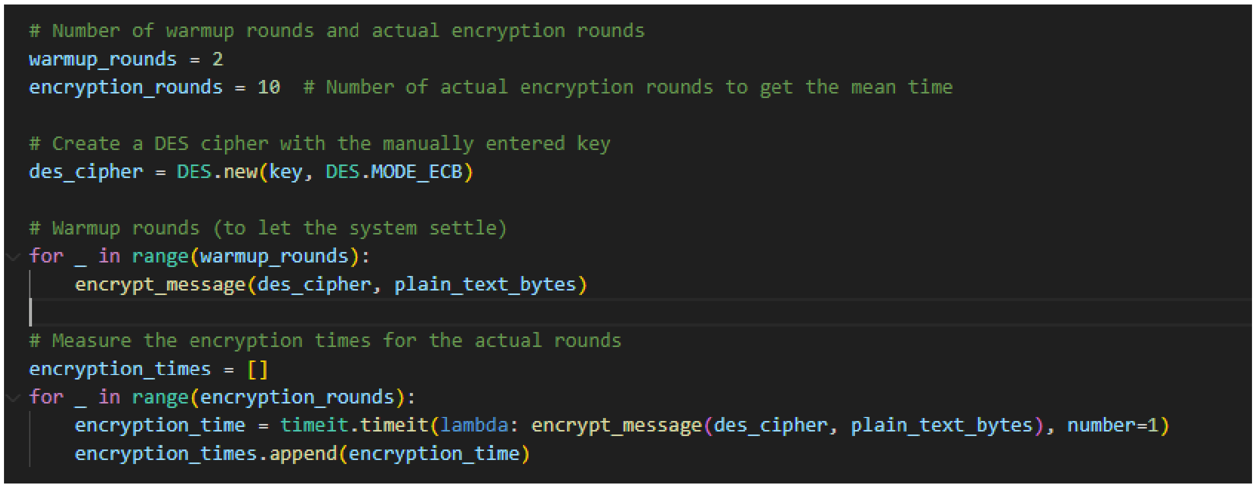
Each encryption round used a manually entered 8-byte key in hexadecimal format. In this code currently used a fixed message called “Security key update scheduled at midnight.”. This is shown in figure 16. This was padded to match the DES block size of 8 bytes using *PKCS5* padding. In here I took 100 different plaintext and 100 different unique keys and each plaintext were encrypted by using these 100 keys.



```python
# Function to encrypt the plaintext with a given key
def encrypt_message(des_cipher, plain_text_bytes):
    return des_cipher.encrypt(plain_text_bytes)

# Manually enter the encryption key (must be 8 bytes / 16 hex characters)
key_hex = input("Enter the encryption key (16 hex characters): ")
if len(key_hex) != 16:
    raise ValueError("Key must be exactly 16 hex characters (8 bytes).")
key = bytes.fromhex(key_hex)

# Plaintext to encrypt
plain_text = "Security key update scheduled at midnight."
plain_text_bytes = pad(plain_text.encode(), DES.block_size)  # DES.block_size = 8
```

Figure 16-Setting plaintext and encryption key

And I code this script to perform 2 warmup encryption rounds before timing begins. This is done because to allows the interpreter, cipher initialization and any cache mechanism to stabilize. Then ten actual encryption rounds are timed using Python's *timeit* module.



```python
# Number of warmup rounds and actual encryption rounds
warmup_rounds = 2
encryption_rounds = 10  # Number of actual encryption rounds to get the mean time

# Create a DES cipher with the manually entered key
des_cipher = DES.new(key, DES.MODE_ECB)

# Warmup rounds (to let the system settle)
for _ in range(warmup_rounds):
    encrypt_message(des_cipher, plain_text_bytes)

# Measure the encryption times for the actual rounds
encryption_times = []
for _ in range(encryption_rounds):
    encryption_time = timeit.timeit(lambda: encrypt_message(des_cipher, plain_text_bytes), number=1)
    encryption_times.append(encryption_time)
```

Figure 17-Measure time for encryption

Then I took the mean encryption time across these 10 rounds as the representative value for that key-plaintext pair. This process ensures that outliers caused by temporary CPU fluctuations or context switches are smoothed out by multi-run normalization.

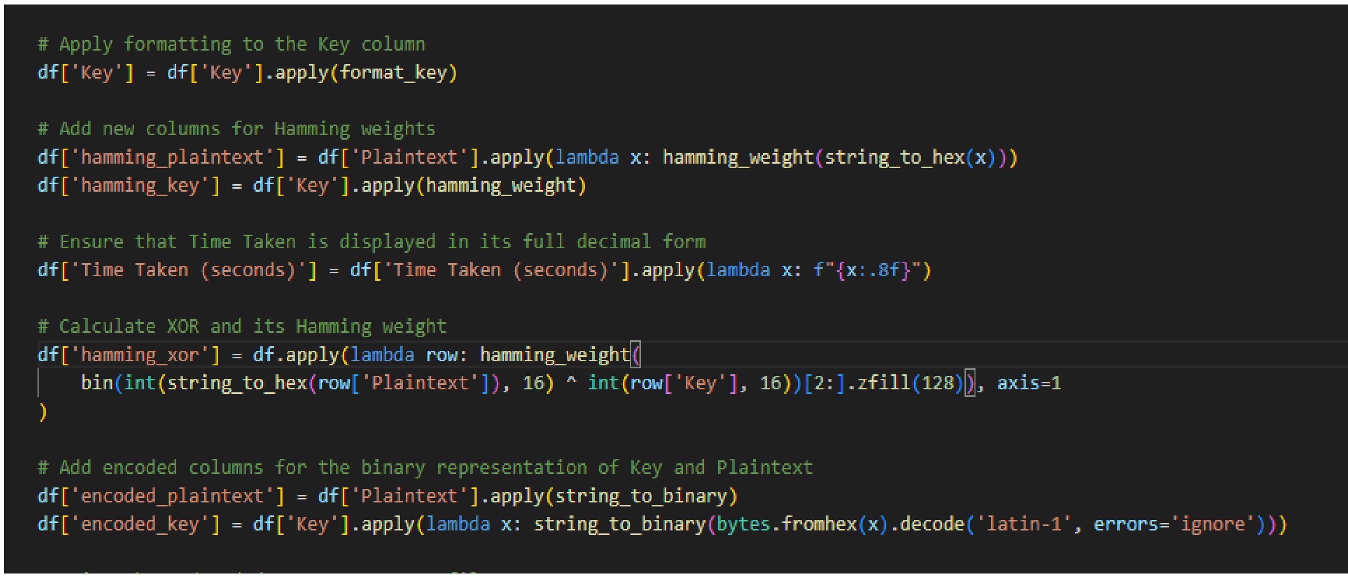
Then the plain text, key and corresponding mean encryption time are then saved to a CSV file to train the model. The output includes:

- Plaintext (human readable)

- Key in hexadecimal string
- Mean encryption time in seconds

Step 3: feature engineering

And the next part is feature engineering. In this step is to prepare the data for machine learning. In here additional features were derived from the plaintext and key using separate python scripts.



```python
# Apply formatting to the Key column
df['Key'] = df['Key'].apply(format_key)

# Add new columns for Hamming weights
df['hamming_plaintext'] = df['Plaintext'].apply(lambda x: hamming_weight(string_to_hex(x)))
df['hamming_key'] = df['Key'].apply(hamming_weight)

# Ensure that Time Taken is displayed in its full decimal form
df['Time Taken (seconds)'] = df['Time Taken (seconds)'].apply(lambda x: f"{x:.8f}")

# Calculate XOR and its Hamming weight
df['hamming_xor'] = df.apply(lambda row: hamming_weight(
    bin(int(string_to_hex(row['Plaintext']), 16) ^ int(row['Key'], 16))[2:].zfill(128)), axis=1
)

# Add encoded columns for the binary representation of Key and Plaintext
df['encoded_plaintext'] = df['Plaintext'].apply(string_to_binary)
df['encoded_key'] = df['Key'].apply(lambda x: string_to_binary(bytes.fromhex(x).decode('latin-1', errors='ignore')))
```

Figure 18-Finding hamming weights and hamming distance

These features include:

- Hamming weight of the Key and Plaintext: in here this measures the number of bits set to 1. Since power consumption and timing are often correlated with bit transitions. This feature captures the density of the data being processed.
- Hamming Distance: The bitwise difference between encoded key and plaintext was calculated. This provides a measure of how different two inputs are at the bit level which can influence the internal state transitions of the cipher, plaintext and thus its timing.
- Encoded representation: Both plaintext and keys were further encoded into binary or numerical formats suitable formats suitable for feeding into machine learning models. This included bit level decomposition for an example *0x1f* to *00011111*. And mapping characters to ASCII or ordinal values to facilitate training.

These features help the model to interpret the binary structure of inputs and better associate them with timing variations. The rich dataset which includes the original inputs, encryption time, and derived features for training and evaluation.

Step 4: dataset preparation

So the final dataset store as *encoded_data1.csv* which consists of rows where each entry represents a single encryption event with all engineered features. The target variable is a selected subset of DES bits and the input features include timing, hamming weights and hamming distances and encoded plaintext/key features. The data was preprocessed with normalization and split into training and testing sets in future model training.

Training the model

Transition from random forest to lstm: motivation and results

So in the early stages of this research I used Random Forest as a baseline model to predict the encryption key bits of DES encryption. I chose this model because of its ease of use, interpretability and strong performance on tabular data. Random Forests are ensemble learning methods that aggregate predictions from multiple decisions tree and reducing overfitting and improving generalization [17]. However encryption key predictions is inherently a sequential task especially when inputs like binary encoded plaintext are involved. These sequences contain or carry temporal dependencies that Random Forest models are inherently incapable of capturing. Because they trat each input independently without regard for sequence order or interdependencies between bits.

So by recognizing these limitations then I used a Recurrent Neural Network (RNN) architecture specifically Long Short Team Memory (LSTM) networks. Basically LSTMs are designed to process and learn from sequential data by maintaining a memory of previous time steps which allows them to model complex dependencies [18]. So this capability made LSTMs good for encryption key prediction problem because both the order and relationships among bits in plaintext and side channel timing features can influence key prediction. By implementing the LSTM model I observed a consistent improvement in predictive accuracy compared to the Random Forest model. The ability of LSTM to perfectly process sequences led to more reliable and generalizable results and validating the decision to adopt this architecture.

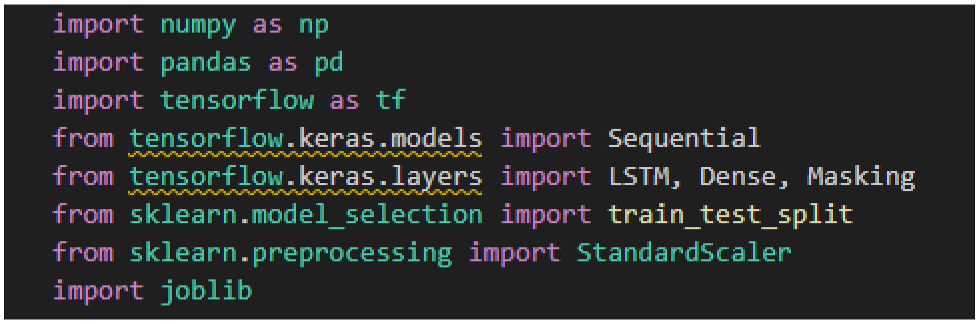
Why lstm and the model selection rationale

I use LSTM because LSTM networks are a specific type of RNN that us able to learn long term dependencies in data sequences. They are particularly effective at avoiding the vanishing gradient issue that might trouble standard RNNs and making them suitable for tasks involving long sequences such as natural language processing and time series predictions. In the context of this project predicting encryption key bits based on binary plaintext sequences, timing data and hamming weight and hamming distance metrics naturally lends itself to sequences modeling.

In here the goal was to learn the mapping between sequences of input features and sequences of output key bits. LSTMs have memory cells and gates. And they are capable of learning to identify patterns in input sequences that correspond to specific output bit sequences. In addition LSTM model can learn temporal dependencies and context dependent patterns for example how the hamming weight or timing of some plaintext byte can be associated with the related key bits. This makes them highly suitable for side-channel analysis where these types of indirect correlations are essential to infer the key.

Step 1: data loading and preprocessing

So the next part is data loading and preprocessing. I took approach for this process by importing necessary python libraires first such as NumPy for numerical computations, Pandas for data manipulation, TensorFlow and Keras for building and training the neural network and Scikit-learn for data normalization and model evaluation in figure 19.

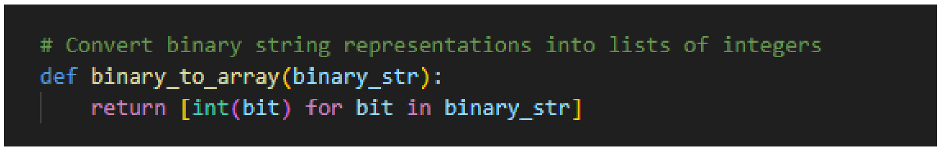


```
import numpy as np
import pandas as pd
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import LSTM, Dense, Masking
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
import joblib
```

Figure 19-Imported libraries

So the dataset is stored in a CSV file named *encoded_data1.csv* which includes binary-encoded plaintexts, binary-encoded keys, timing data for each encryption operation, and precomputed Hamming features.

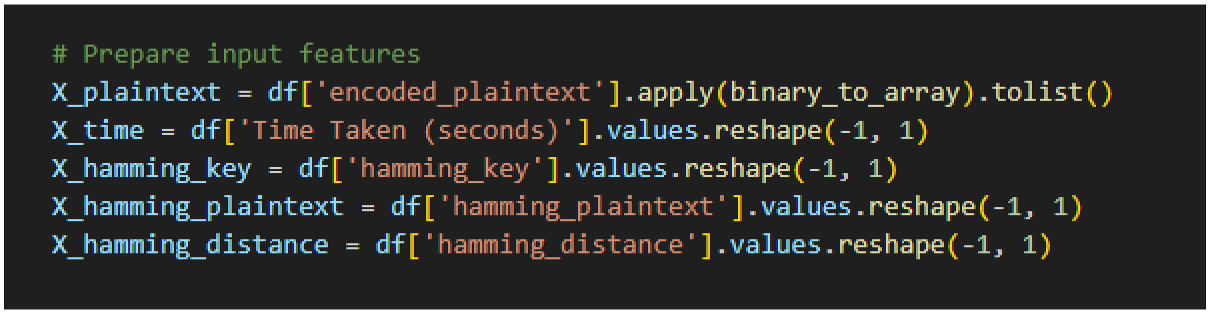
The first preprocessing step involves converting binary strings into numerical arrays. This is done to transform the the encoded plaintext and key form their stings forms for example like 101010011 into arrays of integers like [1,0,1,0,1,0,0,1,1]. This transformation important because machine learning models operate numerical input. Without this conversion the model would not be able to interpret the binary data meaningfully. This process shows in figure 20.



```python
# Convert binary string representations into lists of integers
def binary_to_array(binary_str):
    return [int(bit) for bit in binary_str]
```

Figure 20-Convert to lists of integers

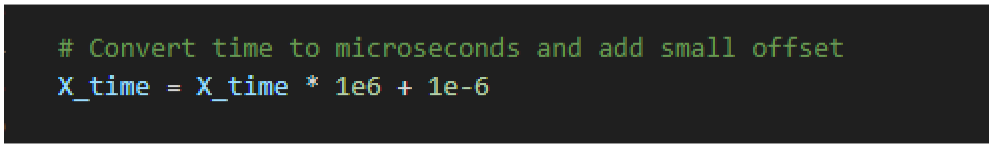
Once converted the plaintexts are extracted using the *encoded_plaintext* column and converted to a list of arrays. Like this encoded key are transformed and stored for later use as output labels. Additional features such as encryption time, hamming weight of the plaintext/key and hamming distance are extracted and reshaped into 2D arrays to prepare for normalization fig (21).



```python
# Prepare input features
X_plaintext = df['encoded_plaintext'].apply(binary_to_array).tolist()
X_time = df['Time Taken (seconds)'].values.reshape(-1, 1)
X_hamming_key = df['hamming_key'].values.reshape(-1, 1)
X_hamming_plaintext = df['hamming_plaintext'].values.reshape(-1, 1)
X_hamming_distance = df['hamming_distance'].values.reshape(-1, 1)
```

Figure 21-Preparing features

Then the data gathering phase timing data initially recorded in seconds. Then I converted it into microseconds by multiplying by *1e6.* A very small value *(1e-6)* is added to avoid zeros entries which could interfere with scaling algorithms or cause division errors fig (20).



```python
# Convert time to microseconds and add small offset
X_time = X_time * 1e6 + 1e-6
```

Figure 22-Convert time to micro seconds

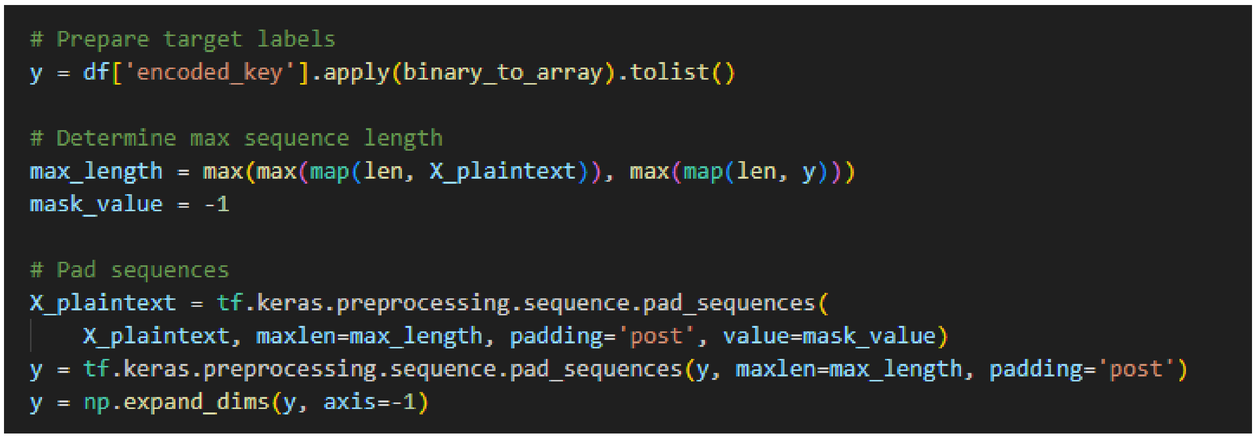
Step 2: feature normalization

So the next part is feature normalization and saving part. To ensure all inputs features are on a comparable scale I applied normalization by using the *StandardScaler.* This scaler standardizes feature by removing the mean and scaling to unit variance. By normalizing the features it improves model convergence and stability during training. Each feature is normalized independently to preserve its unique characteristics while making it numerically compatible with the training process.

After normalization the scalers are saved using *joblib*. This allows the same transformation to be applied during in the key bit prediction and ensuring consistency between training and deployment.

Step 3: padding and masking

Since input sequences are in different length I applied padding to make all sequences the same length. This is necessary because neural networks require inputs of uniform shape. The maximum sequence length among all inputs and output is determined and shorter sequences are padded with a special mask value (-1) to indicate they should be ignored during training. I used -1 because plaintext is in binary format it already contains 0,1 and so to pad it I use -1. The target outputs (key bits) are also padded to match this maximum length and an extra dimension is added to match the expected shape for the model.



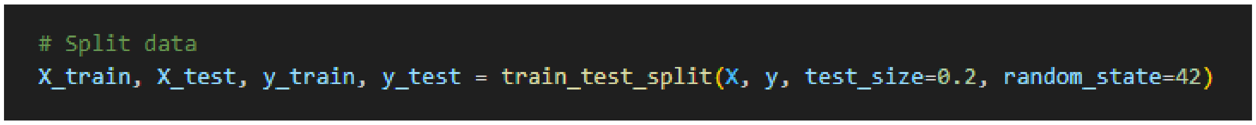
```python
# Prepare target labels
y = df['encoded_key'].apply(binary_to_array).tolist()

# Determine max sequence length
max_length = max(max(map(len, X_plaintext)), max(map(len, y)))
mask_value = -1

# Pad sequences
X_plaintext = tf.keras.preprocessing.sequence.pad_sequences(
    X_plaintext, maxlen=max_length, padding='post', value=mask_value)
y = tf.keras.preprocessing.sequence.pad_sequences(y, maxlen=max_length, padding='post')
y = np.expand_dims(y, axis=-1)
```

Figure 23-Padding

So next non sequential features like timing and hamming metrics need to be repeated across all time steps to align with the sequential data. I did this because to ensure that at every timestep the model receives the same side channel feature values to effectively give it complete context. And all repeated features are then stacked with the plaintext array to form a 3D tensor where each sample contains *max_length* timesteps and 5 feature per timestep.

Step 4: train-test split

So the next step involves splitting data for training and testing. To evaluate the generalization of the model the dataset is split into training and testing subsets using an 80/20 ratio. This ensures the model is evaluated on data it hasn't seen during training (fig 24).

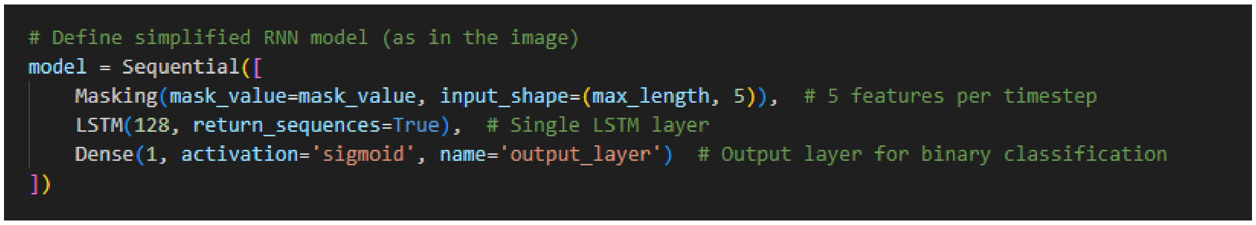


```python
# Split data
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

Figure 24-split dataset for training

Step 5: model architecture

The next part is for model architecture. The model consists of a sequential neural network with three primary layers. They are masking layer, LSTM layer and a dense output layer. The masking layer tells the model to ignore padded values (-1) to ensure that they do not affect learning.And this model followed by an LSTM layer with 128 memory units or neurons. This layer processes each sequence and captures dependencies between different parts of the sequence. The *return_seqnces=true* parameter ensures that the LSTM outputs a prediction at every timestep which is necessary for sequence-to-sequence tasks like bit prediction.Fianally a dense layer with a sigmoid activation function outputs a binary prediction for each timestep. The sigmoid function is appropriate here because the task is binary classification for each bit.



```python
# Define simplified RNN model (as in the image)
model = Sequential([
    Masking(mask_value=mask_value, input_shape=(max_length, 5)),  # 5 features per timestep
    LSTM(128, return_sequences=True),  # Single LSTM layer
    Dense(1, activation='sigmoid', name='output_layer')  # Output layer for binary classification
])
```

Figure 25-Model architecture

In my previous experiment deeper models with multiple LASTM layers were tested. However these models did not yield significant accuracy improvements and increased the risk of overfitting and longer training times. As a result a single layer architecture was chosen for its simplicity and comparable performance.

Step 7: compilation and training

The final step of this training process is compilation, training and evaluation. The model is complied with binary cross entropy loss and the Adam optimizer. Binary cross entropy is suitable for binary classification task and Adam is an efficient gradient based optimization algorithm. And training is performed over 50 epochs with a batch size of 32. Validation data is used to monitor the model's performance on unseen data during training. Finally the model is evaluated on the test set to report the final accuracy and the train model and relevant metadata are saved for future use.

Prediction phase of lstm-based des key bit

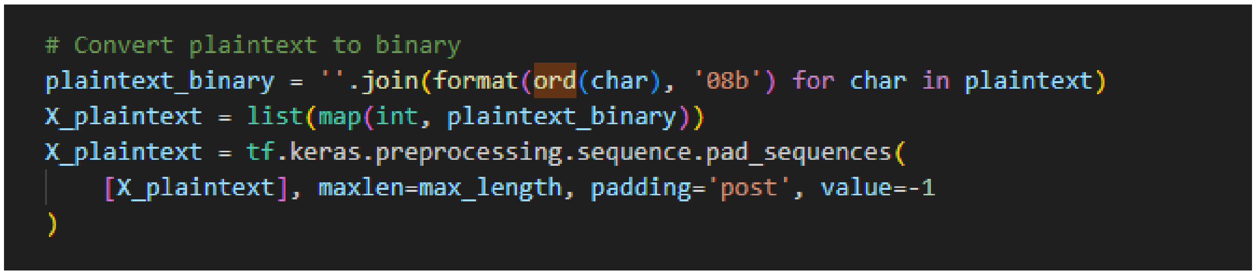
After training and validation of an Long Short-Term Memory (LSTM) model able to predict encryption key bits based on timing data and Hamming metrics. Prediction is part of the system evaluation process to see how well the system works in real life. The goal in this stage is to take actual user input (plaintext, encryption time, and starting key), preprocess relevant features and use the trained LSTM model to predict the key on a bit-by-bit basis. This section describes and explains every component of the prediction pipeline, feature processing, input preparation, model prediction and evaluation.

Step 1: loading trained model and scaling parameters

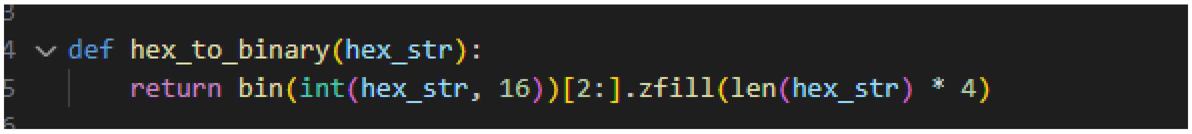
We first load the artifacts generated while training. They include the trained LSTM model in encryption_key_rnn_simple.h5 and four StandardScaler objects for normalizing input features at train time: scaler_time, scaler_hamming_key, scaler_hamming_plaintext, and scaler_hamming_distance. The scalers are necessary to maintain consistency between training and inference so that input features at prediction time can be normalized just like they were when the model was training. Also loaded is max_length of sequences, which was used in padding during training to achieve a consistent input shape.

Step 2: preprocessing user inputs (plaintext, time, key)

In here three significant inputs are provided by the user. They are plaintext message, DES encryption time in seconds and the original DES key in hexadecimal. These are then processed and encoded into binary representations for model inference. So then plaintext is first converted from its string form into an 8-bit binary representation using Python's ord() function and formatted with zfill(8) to ensure every character is encoded to 8 bits. This bitstream serves as input data to the DES encryption function and will be the primary sequence input to the LSTM.



```python
# Convert plaintext to binary
plaintext_binary = ''.join(format(ord(char), '08b') for char in plaintext)
X_plaintext = list(map(int, plaintext_binary))
X_plaintext = tf.keras.preprocessing.sequence.pad_sequences(
    [X_plaintext], maxlen=max_length, padding='post', value=-1
)
```



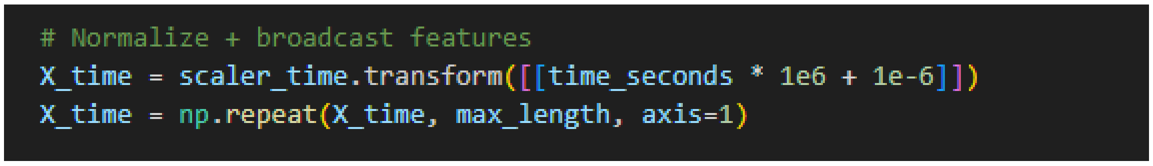
```python
def hex_to_binary(hex_str):
    return bin(int(hex_str, 16))[2:].zfill(len(hex_str) * 4)
```

Figure 26-Preprocessing inputs

Then the original key is also converted into a binary string using the *hex_to_binary* function. Which ensures that each hexadecimal character is accurately translated into 4 binary bits to preserve the exact bit representation of the key. This binary values is used to calculate various Hamming related features like the Hamming weight of the key (the number of 1s), the Hamming weight of the plaintext and the Hamming distance between the first 64 bits of the plaintext and the key. These metrics were chosen because they represent bit level complexity and transformation effort. Which key indicators of computational load during encryption may influence timing and ultimately leak through side channels.

Step 3: normalization of features

Then I worked on normalizing the features to prepare them for the model. First I took each feature and scaled them using the scalers I had trained earlier. But before that I had to handle the encryption time carefully. So I multiplied it by 10^6 to convert it into microseconds. Then I added a tiny epsilon (10^-6) to avoid any zero values which could cause issues later. Normalization was important because the raw features were all over the place in terms of scale. For example hamming weight range form 0 to 64 while timing measurements are microseconds. If I hadn't normalized them the features with larger values like timing would have dominated the model's learning process throwing off the predictions (figure 27).



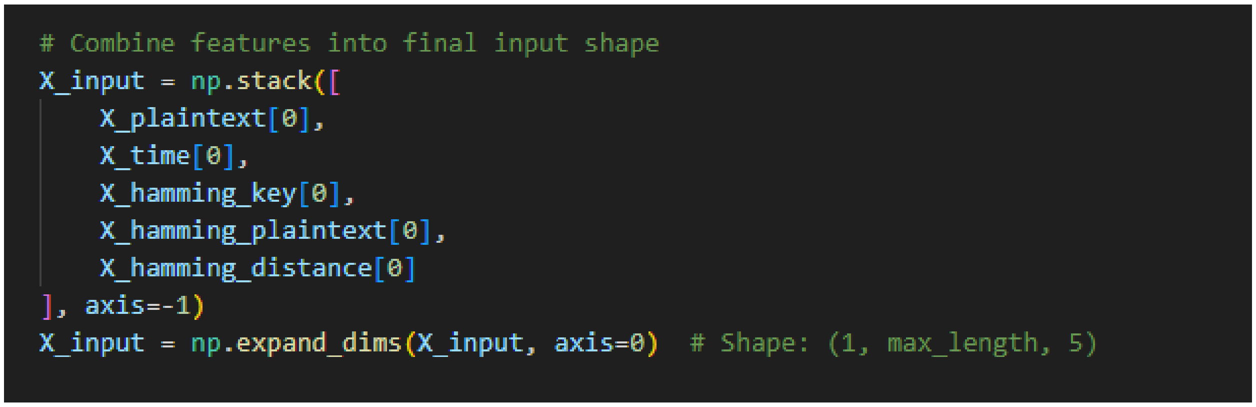
```
# Normalize + broadcast features
X_time = scaler_time.transform([[time_seconds * 1e6 + 1e-6]])
X_time = np.repeat(X_time, max_length, axis=1)
```

Figure 27-Normalize the time

Then I had to adapt the scalar features like encryption time and Hamming metrics. So they could work with the LSTM's sequence based input format. Since the model expects sequences I broadcasted each scaler value across all time steps by repeating it to match the maximum sequence length. For example if the plaintext was padded to 128 bits I expanded the time, hamming key/plaintext weight and hamming distance into arrays of length 128 where every timestep contained the same repeated value. This way each timestep in the LSTM now gets five features. They are the encoded plaintext bit, the repeated time value, the repeated Hamming key weight, the repeated plaintext Hamming weight and the repeated Hamming distance between them. By aligning everything like this the LSTM can process the full sequence while still considering those crucial scalar features at every step.

Finally with all the features properly aligned and broadcast. Then I stacked them along a new axis to create the final input for the model. The result was a clean 3D tensor with shape

(1, max_length, 5) where "1" represents our single example in the batch and *"max_length"* covers all the time steps. And "5" corresponds to our feature channels at each step: the plaintext bit, encryption time, Hamming key weight, Hamming plaintext weight, and their Hamming distance. This structure perfectly matches what the LSTM was trained on and giving it the right temporal format to process sequences effectively. Now the model can analyze both the bit patterns and their associated computational fingerprints in one cohesive input.

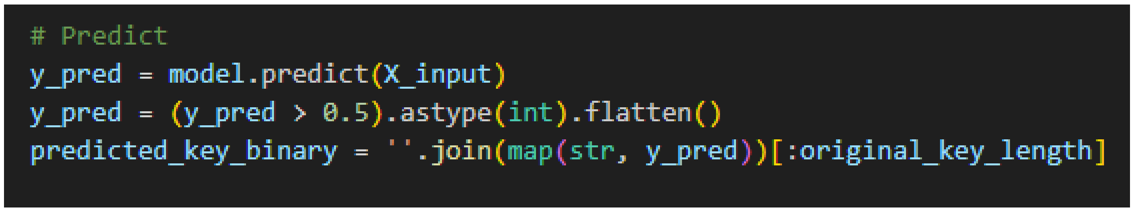


```
# Combine features into final input shape
X_input = np.stack([
    X_plaintext[0],
    X_time[0],
    X_hamming_key[0],
    X_hamming_plaintext[0],
    X_hamming_distance[0]
], axis=-1)
X_input = np.expand_dims(X_input, axis=0)  # Shape: (1, max_length, 5)
```

Figure 28-Combining features

Step 4: model inference and prediction

Now I ran inference using the predict method and it gave me a sequence of probabilities one for each bit position. Each value was a float between 0 and 1 basically the model's confidence that the corresponding key bit should be a 1. To turn these into actual predictions I thresholded everything at 0.5. So anything above became a 1, anything below a 0. Then I flattened the predictions and trimmed them down to match the original key length. The binary prediction sequence is joined into a string to represent the predicted encryption key.



```
# Predict
y_pred = model.predict(X_input)
y_pred = (y_pred > 0.5).astype(int).flatten()
predicted_key_binary = ''.join(map(str, y_pred))[:original_key_length]
```

Figure 29-Code for prediction

Step 5: evaluating the prediction

Then to assess the accuracy of the prediction a bit-wise comparison is performed between the original key and the predicted one. This is expressed as a bit match percentage the proportion of key bits that were correctly predicted. This provides an intuitive performance metric that reflects how successful the model was in identifying the correct bit values from timing and side-channel-like features. For enhanced interpretability the results are color coded to visualize which bits were correctly or incorrectly predicted. Green (\033[92m) highlights matching bits while red (\033[91m) highlights mismatches. The binary strings are chunked into 8-bit groups to improve readability and mimic the structure of standard DES key blocks.

Step 6: visualizing feature impact

Lastly to gain further insights into the model's input and its behavior I plot the values of all five features across the sequence using Matplotlib. These plots show how each feature varies across time steps which helps visualize the relationship between the plaintext structure and other input metrics. The encoded plaintext sequence reflects the information content of the message while the time, Hamming key, Hamming plaintext and Hamming distance sequences show constant values due to broadcasting. Although these features are scalar per message broadcasting allows the LSTM to consider their influence at every bit position during temporal modeling.

Hardware Integration

Step 1: system setup and deployment workflow

To place the timestamp based DES key generation/prediction and the LSTM-based DES key prediction system in a real-world context, an end-to-end deployment workflow was created on the Raspberry Pi 4 Model B. This was the shift from pure software based simulations to an embedded in real-time hardware setup. Raspberry Pi was chosen as it provided the perfect combination of low cost, small form factor and Python based machine learning library support that qualified it to be an extremely strong contender for prototyping cryptographic systems on constrained environments. Both these approaches one depicting key generation in compromised environments and the other one demonstrating key recovery through deep learning were easily integrated to build a standalone side channel analysis tool capable of running independently without the need for a complete computer system.

Step 2: installing the operating system

To place the timestamp based DES key generation and the LSTM based DES key prediction system in a real world context an end to end deployment workflow was created on the Raspberry Pi 4 Model B. This was the shift from pure software based simulations to an embedded, real-time hardware setup. Raspberry Pi was chosen as it provided the perfect combination of low cost, small form factor and Python based machine learning library support that qualified it to be an extremely strong contender for prototyping cryptographic systems on constrained environments. Both these approaches one depicting key generation in compromised environments and the other one demonstrating key recovery through deep learning were easily integrated to build a standalone side channel analysis tool capable of running independently without the need for a complete computer system.

Step 3: remote access configuration

With the operating system set up remote access was set up to give full control of the Raspberry Pi without needing external peripherals such as a keyboard or monitor. Both laptop and Raspberry Pi shared the same home Wi-Fi network which ensured smooth communication. An official browser based tool called Raspberry Pi Connect was used to control GUI access and terminal based communication. This utility allowed researchers to run scripts, install packages, observe model performance and modify code without ever having to lay hands on the Raspberry Pi. This configuration was beneficial to try out the timestamp generation process, experiment with various time intervals and observe the LSTM model predict DES key bits from side channel timing behavior all from one remote console. This low overhead approach not only lowered overheads but also showed that side channel analysis tools could be utilized efficiently in resource scarce settings.

Step 4: installing required libraries

Subsequently the necessary software platform was installed to facilitate timestamp based key analyze and deep learning inference. Python 3 base acted as the shared program framework which was complemented with the needed libraries. Python datetime, random and bitarray libraries were used to create and encode time sourced DES keys to create timestamps for the timestamp generation module. For the LSTM model, NumPy, pandas and scikit-learn were used to allow for data processing, model loading and prediction. If PyTorch or Keras were employed to train the LSTM model. They were also utilized to deal with on device inference. Such integration permitted the Pi to generate realistic DES keys and further conduct machine learning predictions on said keys from their trace properties. The whole procedure was modular in nature that is the scripts could be extended or reworked without modifying the system core. The lightweight Python-oriented environment was very flexible even on the constrained hardware footprint of the Raspberry Pi.

Step 5: lcd screen integration

To give portability and immediacy to the application currenlty a 16x2 LCD display was integrated into the Raspberry Pi setup as a sole output device for the key generation and the prediction outputs. In the time stamp approach the screen's output showed the DES key closest to the entered key by calculating the Hamming distances. In the machine learning based solution the outputted key bits of the LSTM model were directly rendered onto the screen for real time viewing of the model's inference. The LCD connected via GPIO pins and controlled by specially written Python scripts was thus capable of providing real time, readable feedback without requiring an external interface or secondary monitor. This unified integration transformed the system into a completely standalone and user-friendly sidechannel analysis tool which is extremely easy to use for class demonstrations, security labs and testbeds.

Commercialization Aspect

Timestamp-based key generation and analysis – commercialization aspect

The timestamp based key generation and analysis approach is an effective teaching tool for informing academic and research communities about how weak randomness and predictable inputs compromise encryption schemes. This method is especially suitable for universality students and entry level cybersecurity learners. It visually and practically demonstrates the dangers of using low entropy or time-based seeds in key generations. By allowing users to input a specific time range and the system generates DES keys in a deterministic way. Which can then be analyzed to show their predictability and repetition. This makes the method an ideal educational tool in cryptography and secure system design courses.

From a researcher's perspective this technique offers a practical way to replicate and test real world scenarios where key generation is poorly implemented such as in legacy systems or resource constrained IoT devices. It enables effective side channel vulnerability modeling by demonstrating how attackers could regenerate keys through brute force guessing when weak randomness is used. The tool is both lightweight and user friendly and requiring only a standard Python environment with minimal setup. Generated keys are automatically saved in a human readable format which simplifying analysis and integration with other security tools. Its portability and ease of use make it ideal for classroom demonstrations, security workshops, or cryptographic auditing training programs. This approach helps foster a deeper understanding of weak key vulnerabilities and underscores the importance of secure randomization practices in cryptography.

Lstm-based key bit prediction – commercialization aspect

The LSTM based key prediction model is designed for more advanced users and institutions seeking to explore the intersection of machine learning and cryptographic analysis. By using encryption timing and statistical features such as Hamming weight and distance the model is trained to predict bits of the encryption key. This method provides a highly valuable platform for researchers and security analysts to investigate side channel vulnerabilities that arise from subtle patterns in system behavior. It is a very good example of how apparently harmless leakag like encryption time, can be exploited using deep learning making it a fascinating case study for both attack and defense research.

From a business perspective this approach maps onto graduate school studies, research lab courses and post graduate cryptography training. Its application of LSTM networks introduces a practical element to abstract discussion of side-channel attacks, and it creates opportunities for experimentation with model training, feature engineering and countermeasures. While being technically sophisticated the system is made modular and easy to use with well documented processes for dataset creation, preprocessing, model training and prediction. The code can be easily modified to accommodate any key sizes and encryption algorithms and hence it is a handy experimental tool. It can be used as a benchmark system in security research laboratories to evaluate countermeasures such as masking, constant-time execution or noise injection. Such a methodology while facilitating effective analysis of crypt breaking weaknesses and also inspires innovation in machine learning based countermeasures hence filling the gap between theoretical academia and real-world utility in cybersecurity.

Testing & Implementation

Implementation

The application of the DES encryption key analysis tool took place in two complementary stages. Both characterized by a distinctive approach to analyzing cryptographic security through the use of time-based methods. The first approach used a timestamp-based key generation and similarity analysis method. So it's designed to reveal the vulnerabilities associated with poor randomization techniques. The second approach used deep learning in particular a Long Short Term Memory (LSTM) model trained on side channel timing data to predict encryption key bits. Both methods were developed independently in software environments and subsequently integrated into a portable and practical hardware based solution using a Raspberry Pi 4 Model B.

Timestamp-based key analysis implementation

The timestamp based approach was fully developed using Python and involved approximating weak cryptographic systems through deterministic key generation. Here the user is asked to input a time range through human readable date and time formats. These values are translated into Unix timestamps and utilized as seeds for a pseudo random number generator. An 8-byte DES style key is produced for every timestamp within the input time interval. The hexadecimal keys are stored in a plain text file so that they may be examined and compared later easily. After the key set has been generated the system conducts two stage similarity comparisons between the DES key supplied by the user and the keys generated. The first is a byte by byte match percentage which ensures the number of same bytes from both keys. The second level is more detailed in its examination by translating the keys to their complete binary form and calculating the binary level Euclidean distance. This produces a more accurate similarity measurement and is utilized to demonstrate how predictable and susceptible time seeded keys are. The outcome of the examination is graphed with color coded similarity markers and an alarming message if the nearest key is over a specified similarity threshold usually 65%. Not only was the module created to be a demonstration of vulnerability but also for educating on why entropy is a critical component for secure key creation.

Lstm-based key prediction implementation

The second implementation phase was to construct a deep learning based prediction model with an LSTM network. The model needed to restore DES key bits from timing information and cryptographic features. A 10,000 sample dataset with a controlled scenario was established wherein each sample constituted a single plaintext, a single key and the resultant mean encryption time derived by utilizing Python's pycryptodome library. To ensure consistency in the dataset and minimize environmental noise all the timing measurements were carried out in a virtual machine with minimal background processes. Multiple runs of each plaintext key pair were encrypted and only the mean of numerous runs of execution time was recorded to adjust for variation.

Feature engineering was time consuming where inputs were converted to machine learning ready formats. The Hamming weight of plaintext and key, Hamming distance between them and binary-encoded representations of both inputs were all features computed and normalized. The LSTM model architecture was then created to accept these sequences. The network consisted of a masking layer to process variable length input sequences, a single LSTM layer with 128 memory cells and a dense sigmoid activated output layer to predict individual bits of the DES key. The model was compiled with binary cross entropy loss and the Adam optimizer during training. At 50 training epochs the LSTM model peaked at 96% accuracy. Quite different from the basic random forest model which only I had 56%.

Hardware integration on raspberry pi

In the interest of demonstrating practical deployment of both approaches the system was completely deployed on a Raspberry Pi 4 Model B. Deployment was achieved through installing the Raspberry Pi with Python and supporting libraries such as TensorFlow, NumPy, pandas and matplotlib. All scripts for the generation of keys, analysis and prediction were pushed into the device. The system was also extended by incorporating a 16x2 LCD display using GPIO pins. This visualization enabled the presentation of the most similar timestamp generated key or the LSTM-generated key bits to be presented in real time. In addition the system waspoised for headless operation via Raspberry Pi Connect to enable remote access and control of the system without the need for an external display or keyboard. A common script was developed to allow the user to select between the timestamp based or LSTM-based analysis approach, provide required inputs and receive results visually as well as in log files. The architecture ensured that the user experience was consistent for both modules with seamless transitions and shared input mechanisms.

Through this multi step deployment procedure the project was able to deliver a robust and portable system that could successfully demonstrate two significant attack surfaces of DES encryption. Combining both analytical approaches into one hardware deployable device created a dramatic increase in terms of enabling side-channel cryptographic analysis to be accessible and understandable for educational and research environments.

Testing

Testing was the fundamental part of this project to verify if both the timestamp based key analysis system and the LSTM based prediction model were not just functionally correct but also efficient as well as trustworthy under various operating conditions. Both of them were completely tested at different levels in order to establish its correctness, robustness and suitability for deployment in real-world applications.

Unit testing

Unit testing was performed at the initial stage of development. Each significant function in the key generation and analysis pipeline was tested separately to ensure correct behavior. In the timestamp-based system unit tests ensured that the same timestamp would always produce the same key and file output was accurate and properly formatted. Additionally the byte-level and bit-level similarity functions were tested with known pairs of keys so that they could produce expected match percentages and Euclidean distances. The pipeline of the LSTM model was also thoroughly tested so that preprocessing operations like encoding, padding, normalization, and stacking of features were functioning as expected. Care was taken to align broadcasted scalar features (such as encryption time, Hamming weight) with the sequential input data to the LSTM in an appropriate manner.

Integration testing

After unit testing integration testing was carried out to check the end-to-end workflow of the system. This involved end-to-end testing from user input to the display of final results for both analytical approaches. For the timestamp-based approach the integration test ensured that user inputs were properly parsed, keys were created for the entire time given and the analysis accurately located the most similar key. For the LSTM model integration testing confirmed user entered plaintext, encryption time and initiating key were carried over through every transformation and resulted in correct prediction and visualization of the key bits. Integration also confirmed LCD display refreshed as it was supposed to and that the system remained responsive with consecutive predictions.

Performance testing

Performance testing was a vital part of the test process particularly considering the proposed deployment of the system on low-powered Raspberry Pi. Testing was conducted to determine the time taken to create keys for vast ranges of timestamps and to test inference time of the LSTM model. The process of key generation could generate more than 1000 keys per second and the LSTM model provided key predictions at a steady pace within less than two seconds. The system was also put to stress conditions over extended usage to check for overheating, lag and system crashes. For all the test scenarios the Raspberry Pi was able to withstand the load without any degradation of performance or responsiveness.

Model feature evaluation

As model-specific testing the main prediction model based on LSTM was additionally tested by means of evaluating the effect of various input feature combinations on the predictive power. This enabled the determination of the side-channel features that are most relevant to the performance of the model. Model training with binary-encoded plaintext as the sole input preceded others. Limited predictive capability was the result which testified to the inadequacy of plaintext data in the identification of key patterns. In the second stage timing information was added as a feature and a significant performance improvement in bit-level predictions was observed like timing information indeed leaks key-dependent patterns.

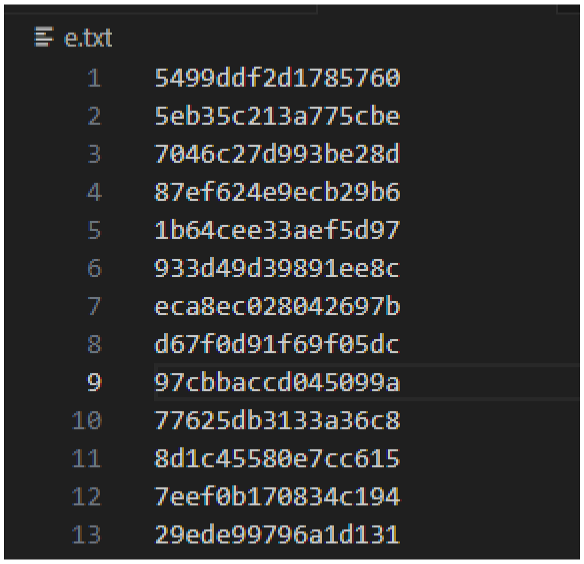
Then Hamming weight of the plaintext was added, then Hamming distance between the plaintext and the key. Each additional feature further enhanced the model's learning capability. When all the features were added together the model showed the best level of key bit inference capability. This step by step assessment approach provided us with more information about the role played by every individual feature in side-channel leakage and supported the argument for rigorous feature engineering in side-channel machine learning attacks. It also affirmed that the LSTM model was capable of learning from more than one dimension of data, not just raw timing sequences.

Through this thorough test strategy unit, integration, performance, and acceptance testing the system exhibited high accuracy, reliability and usability levels. Deployment and successful testing validated the deployed side channel analysis toolkit for immediate demonstration to real-world applications and educational initiatives, largely acting as a training and research tool for cybersecurity intentions.

RESULTS & DISCUSSION

Results

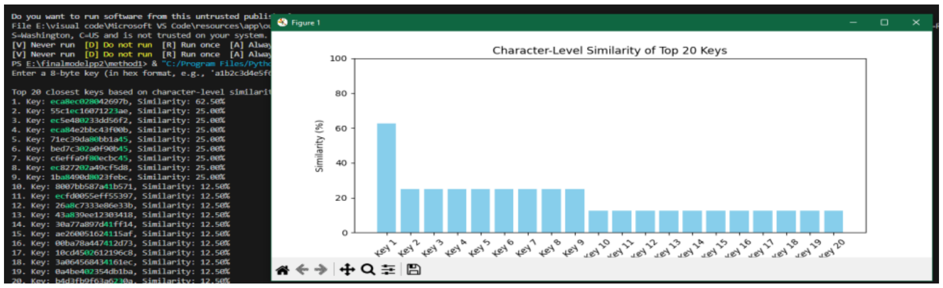
Method 1: timestamp-based key generation and analysis



```
e.txt
5499ddf2d1785760
5eb35c213a775cbe
7046c27d993be28d
87ef624e9ecb29b6
1b64cee33aef5d97
933d49d39891ee8c
eca8ec028042697b
d67f0d91f69f05dc
97cbbaccd045099a
77625db3133a36c8
8d1c45580e7cc615
7eef0b170834c194
29ede99796a1d131
```

Figure 30-Generated DES keys

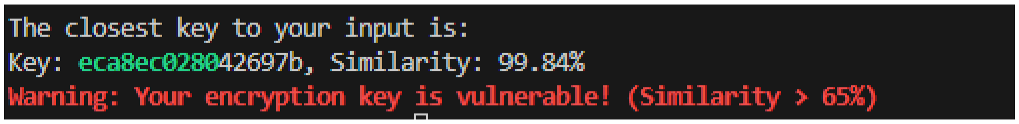
The first image (Figure 30) in the result set is a representation of the DES keys generated from a user-defined time range. The script accepts start and end timestamps provided by the user and converts them to Unix format and produces a key per second using a seeded pseudo-random number generator. It displays each key in hexadecimal form and it is a 64-bit DES-style key. This image confirms the deterministic nature of the key generation method which is for a fixed timestamp the same key will always be produced. This points to the fundamental flaw in the use of predictable timestamps as sources of entropy in key generation. As it significantly reduces the keyspace and brute-force recovery becomes possible if the range of the timestamp can be guessed or is known.



```
Do you want to run software from this untrusted publis
File E:\visual code\Microsoft VS Code\resources\app\ou
S=Washington, C=US and is not trusted on your system.
[V] Never run  [D] Do not run  [R] Run once  [A] Alway
[V] Never run  [D] Do not run  [R] Run once  [A] Alway
PS E:\finalmodelpp2\method1> & "C:/Program Files/Pytho
Enter a 8-byte key (in hex format, e.g., 'a1b2c3d4e5f6

Top 20 closest keys based on character-level similarit
1. Key: eca8ec028042697b, Similarity: 62.50%
2. Key: 55c1ec16071223ae, Similarity: 25.00%
3. Key: ec5e480233dd56f2, Similarity: 25.00%
4. Key: eca84e2bbc43f00b, Similarity: 25.00%
5. Key: 71ec39da80bb1a45, Similarity: 25.00%
6. Key: bed7c302a0f90b45, Similarity: 25.00%
7. Key: c6effa9f80ecbc45, Similarity: 25.00%
8. Key: ec827202a49cf5d8, Similarity: 25.00%
9. Key: 1ba8490d8023febc, Similarity: 25.00%
10. Key: 8007bb587a41b571, Similarity: 12.50%
11. Key: ecfd0055eff55397, Similarity: 12.50%
12. Key: 26a8c7333e86e33b, Similarity: 12.50%
13. Key: 43a839ee12303418, Similarity: 12.50%
14. Key: 30a77a897d41ff14, Similarity: 12.50%
15. Key: ae26005162411 5af, Similarity: 12.50%
16. Key: 00ba78a447412d73, Similarity: 12.50%
17. Key: 10cd4502612196c8, Similarity: 12.50%
18. Key: 3a06456843416 1ec, Similarity: 12.50%
19. Key: 0a4be402354db1ba, Similarity: 12.50%
20. Key: b4d3fb9f63a6230a, Similarity: 12.50%
```

Figure 31-Selected top 20 keys

The second graph illustrates a ranking of the top 20 most similar keys chosen from the pool generated on the basis of similarity to the user-provided DES key. The keys are ranked in descending order of byte-level matching percentages with higher percentage meaning higher proportion of matching bytes with respect to the target key. This intermediate result illustrates the filtering of possible matches from a large key pool by simple similarity tests. It presents a real-world example of how quickly an attacker can decrease a small set of candidate keys when the timestamp range is quite precise.



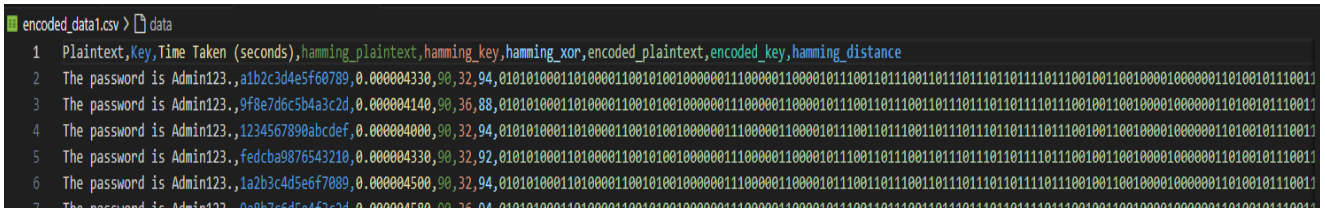
```
The closest key to your input is:
Key: eca8ec028042697b, Similarity: 99.84%
Warning: Your encryption key is vulnerable! (Similarity > 65%)
```

Figure 32-Final output (Closest key)

In the third image the last closest key is calculated and displayed in addition to bitwise similarity comparison based on Euclidean distance. It is superior to byte level comparison since it compares the binary representation of keys produced with the original. The example key of the figure has a good level match once again demonstrating that although a key may not be exactly the same at the character level. So it can be similar in structure when presented in binary form. This is important because it demonstrates that partial key recovery or approximation with this technique is feasible particularly in systems where slight variations in keys are acceptable or where there is no strict authentication scheme.

Method 2: lstm-based des key bit prediction

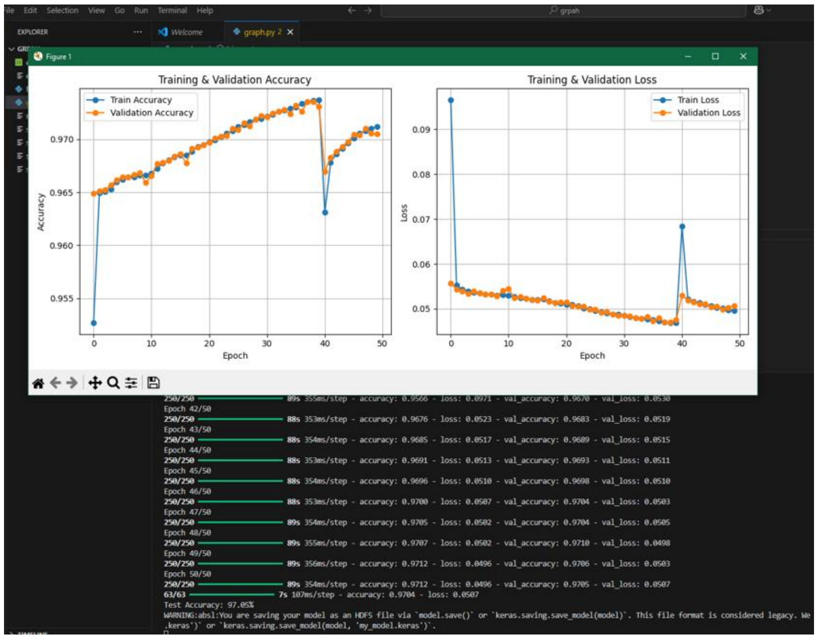
The second approach employs the use of a machine learning method. That is a Long Short-Term Memory (LSTM) neural network for predicting DES key bits from timing-based features. The first photo under the second approach represents the ready dataset within a simulation virtual environment with regularized settings. It contains some fields like plaintext, key, mean encryption time, Hamming weight and Hamming distance. This data set was built by encrypting 100 plaintexts with 100 distinct keys and computing the average running time over several runs to reduce noise. The image confirms the data structure and guarantees the presence of engineered features for model training.



```
encoded_data1.csv > data
Plaintext,Key,Time Taken (seconds),hamming_plaintext,hamming_key,hamming_xor,encoded_plaintext,encoded_key,hamming_distance
The password is Admin123.,a1b2c3d4e5f60789,0.000004330,90,32,94,010101000110100001100101001000000111000001100001011100110111001101110111011011110111001001100100001000000110100101110011
The password is Admin123.,9f8e7d6c5b4a3c2d,0.000004140,90,36,88,010101000110100001100101001000000111000001100001011100110111001101110111011011110111001001100100001000000110100101110011
The password is Admin123.,1234567890abcdef,0.000004000,90,32,94,010101000110100001100101001000000111000001100001011100110111001101110111011011110111001001100100001000000110100101110011
The password is Admin123.,fedcba9876543210,0.000004330,90,32,92,010101000110100001100101001000000111000001100001011100110111001101110111011011110111001001100100001000000110100101110011
The password is Admin123.,1a2b3c4d5e6f7089,0.000004500,90,32,94,010101000110100001100101001000000111000001100001011100110111001101110111011011110111001001100100001000000110100101110011
```

Figure 33-Generated dataset through control environment

The following plot illustrates the training and validation accuracy of the LSTM model for 50 epochs. The plot has smooth rising accuracy in training and convergence of validation accuracy in the later stages which is a sign of good generalization without overfitting. The model had converged to around 96% bit-wise accuracy on the test set much better than the random forest baseline (which reached only 56%). This storyline substantiates the premise that LSTM models can effectively learn from temporal and sequential properties like the time of encryption and Hamming distances.



```
250/250 ━━━━━━━━━━━━━━━━━━━━ 89s 355ms/step - accuracy: 0.9566 - loss: 0.0971 - val_accuracy: 0.9670 - val_loss: 0.0530
Epoch 42/50
250/250 ━━━━━━━━━━━━━━━━━━━━ 88s 353ms/step - accuracy: 0.9676 - loss: 0.0523 - val_accuracy: 0.9683 - val_loss: 0.0519
Epoch 43/50
250/250 ━━━━━━━━━━━━━━━━━━━━ 88s 354ms/step - accuracy: 0.9685 - loss: 0.0517 - val_accuracy: 0.9689 - val_loss: 0.0515
Epoch 44/50
250/250 ━━━━━━━━━━━━━━━━━━━━ 88s 353ms/step - accuracy: 0.9691 - loss: 0.0513 - val_accuracy: 0.9693 - val_loss: 0.0511
Epoch 45/50
250/250 ━━━━━━━━━━━━━━━━━━━━ 88s 354ms/step - accuracy: 0.9696 - loss: 0.0510 - val_accuracy: 0.9698 - val_loss: 0.0510
Epoch 46/50
250/250 ━━━━━━━━━━━━━━━━━━━━ 88s 353ms/step - accuracy: 0.9700 - loss: 0.0507 - val_accuracy: 0.9704 - val_loss: 0.0503
Epoch 47/50
250/250 ━━━━━━━━━━━━━━━━━━━━ 89s 354ms/step - accuracy: 0.9705 - loss: 0.0502 - val_accuracy: 0.9704 - val_loss: 0.0505
Epoch 48/50
250/250 ━━━━━━━━━━━━━━━━━━━━ 89s 355ms/step - accuracy: 0.9707 - loss: 0.0502 - val_accuracy: 0.9710 - val_loss: 0.0498
Epoch 49/50
250/250 ━━━━━━━━━━━━━━━━━━━━ 89s 356ms/step - accuracy: 0.9712 - loss: 0.0496 - val_accuracy: 0.9706 - val_loss: 0.0503
Epoch 50/50
250/250 ━━━━━━━━━━━━━━━━━━━━ 89s 354ms/step - accuracy: 0.9712 - loss: 0.0496 - val_accuracy: 0.9705 - val_loss: 0.0507
63/63 ━━━━━━━━━━━━━━━━━━━━ 7s 107ms/step - accuracy: 0.9704 - loss: 0.0507
Test Accuracy: 97.05%
WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save_model(model)`. This file format is considered legacy. We
.keras')` or `keras.saving.save_model(model, 'my_model.keras')`.
```

Figure 34-Accuracies and losses

Another plot illustrates the impact of input features on the performance of the model. Features like Hamming weight of key and plaintext, their mutual Hamming distance, and execution time were individually tried. The plot indicates how multi feature sets provided the optimal performance of the model reinforcing the significance of combining multi-side-channel dimensions in learning-based attacks. This also indicates that time by itself although beneficial, is more effective when accompanied by cryptographic mechanisms such as Hamming features.

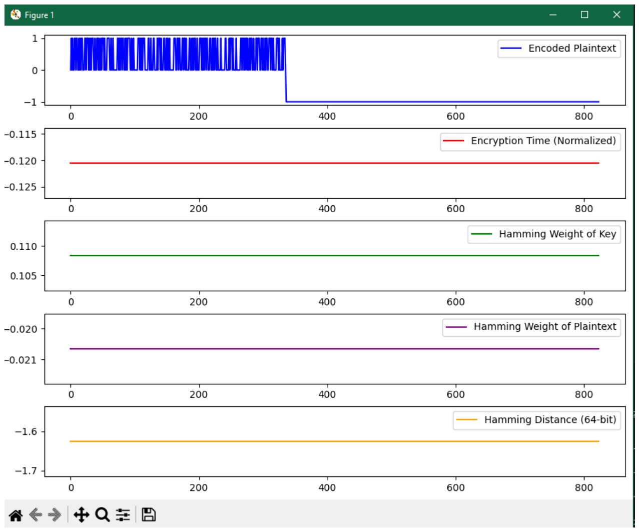
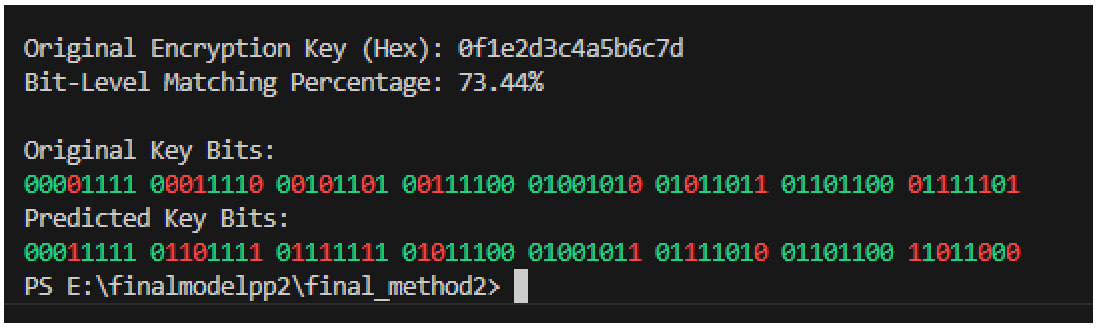


Figure 35-Graph for features

The last picture in this approach displays the real output of the LSTM model where the 64-bit DES key was predicted based solely on the input plain text, encryption time and side-channel information. The output is displayed bit by bit next to the original key where correct bits are colored green and incorrect ones colored red. In this example 47 out of 64 bits were accurately predicted and achieving a 73.44% accuracy in matching. This level of prediction significantly exceeds random guessing (50%) and clearly demonstrates the model's ability to extract meaningful patterns from side-channel features. This image offers tangible proof of partial key recovery from timing behavior and shows that while the entire DES key may not be inferred. A significant portion can be derived with precision by means of the support of software-based side-channel leakage. This is a severe threat in deployed DES-based systems lacking countermeasures against timing-based inference.



```
Original Encryption Key (Hex): 0f1e2d3c4a5b6c7d
Bit-Level Matching Percentage: 73.44%

Original Key Bits:
00001111 00011110 00101101 00111100 01001010 01011011 01101100 01111101
Predicted Key Bits:
00011111 01101111 01111111 01011100 01001011 01111010 01101100 11011000
PS E:\finalmodelpp2\final_method2>
```

Figure 36-Final output (Predicted key)

Research Findings

The findings of this study shows strong evidence for the viability of time based side-channel analysis in breaking the security of DES encryption keys using two complementary methods. They are timestamp-based key regeneration and deep learning-based bit prediction. The results not only identify the weaknesses brought about by compromised key generation procedures and timing leaks but also the potential of data driven approaches like LSTM in predicting cryptographic secrets with high accuracy. The findings of this research provide useful insights into the development of low-resource attack software, reaffirming the need for secure implementation practices in symmetric cryptosystems.

Effectiveness of timestamp-based generation & analysis

One of the most important results of this study is the illustration of how predictable DES key generation processes when seeded from Unix timestamps. It can leave systems vulnerable to viable cryptographic attacks. The approach created enabled the regeneration of DES keys across a user-defined range of timestamps and then found the closest match to a target key based on byte level as well as bit level similarity. The strength of this method is its simplicity and feasibility. It emulates a real threat situation where attackers might be able to estimate the approximate time range when a key was created. This attack demonstrated that deterministic key generation methods particularly those based on lower entropy sources like system time, are extremely vulnerable to brute-force key reconstruction methods. The reality that equivalent keys may be graphically represented and that promising candidates may be determined in real-time additionally renders the method all the more feasible especially for illustrating and explaining key schedule weaknesses in a teaching or classroom environment.

Role of side-channel timing in key inference

The research also demonstrated the important contribution that side-channel timing data can make to cryptographic key deduction. By gathering encryption timing data over a variety of plaintexts and keys this research created a dataset that demonstrates how small execution time differences can hold sufficient information to divulge underlying key structures. Timing differences though less considered were demonstrated to leak patterns regarding how a targeted key engages with input data throughout encryption. Such patterns were the basis of machine learning-based inference within the second approach and highlight the need to employ constant-time encryption routines in cryptographic systems.

Application of deep learning to side-channel analysis

The second approach was the prediction of DES key bits from timing and derived features like Hamming distance and Hamming weight as inputs to a deep learning network. The network was able to learn and track temporal dependencies of data because of the inclusion of a sequence aware neural network and this proved the possibility of statistical modeling of side-channel leakage by contemporary learning architecture. The results show the larger potential for artificial intelligence to be used in cryptographic analysis in non-invasive and software-based modes. The work offers a map of how time-based side-channel information can be used in practical attacks and shows the expanding interrelationship between machine learning and cybersecurity.

Practicality and integration in low-resource systems

One of the most useful results of this research is the affirmation that time based side-channel attacks can be successfully performed on low-budget devices specifically the Raspberry Pi. Both techniques were successfully implemented and tested on this hardware and supporting the observation that advanced attacks no longer need advanced laboratory equipment. This result is especially relevant to the creation of portable, demonstrable systems for education and proof-of-concept studies. By making it possible to perform such attacks on low-cost, small devices this project serves to move theoretical vulnerabilities into an actionable realm that is more tangible and accessible to students, researchers and aspiring security researchers.

Broader implications for cryptographic awareness and education

The findings from the study have wider applications for cryptographic system development and training cybersecurity ideas. From an educational angle, the system explained here is a tangible, real-world educational experience to watch side-channel leakage in action. As opposed to using modeled or hypothetical representations, learners and researchers can watch real-time how vulnerabilities that occur by time can be leveraged in order to compromise key confidentiality. From the perspective of development, the results underscore the importance of developers' surpassing security at the algorithmic level and aiming for secure practices in implementation. These include calls to have the ability to create really random keys and to use constant-time processing in cryptographic computation.

Discussion

This study's findings provide positive potential for improving the understanding, demonstration and counter measurability of time-based side-channel attacks of symmetric key cryptography. This reflection takes into account the implication of the two approaches which is timestamp-based key analysis and deep learning-based DES key prediction particularly in terms of embedded security and teaching cybersecurity. It also outlines limitations encountered during the study and proposes potential directions of future work in the area of side-channel attack research and defense.

Practical implications of time-based DES key inference

The practical implications of this study are far-reaching especially in the field of cybersecurity training and real-world cryptographic security assessments. The DES key vulnerability analysis with timestamp therefore, unveiled that even most secure cryptographic protocols might be vulnerable after key generation is achieved through utilization of low entropy sources such as system time. This attack resolved that for a rough time interval most of the most important key space may be eliminated and the attackers are able to recover or closely approximate the original encryption key. This attack is especially relevant to systems implementing DES or other symmetric algorithms within legacy code or embedded devices. In addition the LSTM key bit prediction model demonstrated a strong ability to reflect small changes in the runtime of the encryption by contemporary machine learning algorithms and to use such sensitive key data to infer. This poses direct implications towards programmers and system designers in which focus lies on using constant-time cryptographic algorithms and refraining from any such timing anomaly which can be attacked.

Issues and limitations

Even though both were successful there are several limitations which have to be quoted. The first one is for controlling the measurement environment while sampling during model training. Encryption times used while building the LSTM model were sampled within a virtual environment where interference is minimized to reduce noise. In real-world systems such clean and isolated measurements are often difficult to obtain due to multitasking, background processes or varying hardware configurations. This could be a general restriction in applying the model to more variable deployment scenarios. Another restriction is the susceptibility of the key inference model to small timing resolution variations. The effectiveness of timing-based inference attacks can be influenced by low-resolution clocks or measurement inconsistencies. For the timestamp-based key analysis case the efficiency of the method is greatly reliant on the validity of the assumed time window. If the attacker has no information about the approximate time of key generation the method becomes less practical due to the large number of possibilities.

Ethical and security concerns in applied cryptanalysis

As with all work that shows the practical exploitation of cryptographic vulnerabilities ethics are absolutely paramount. Although this work was done in an ethical and controlled setting the same tools and techniques illustrated here can be abused if it gets into the wrong hands. It is important that such knowledge is accompanied by ethics training and responsibly applied in settings such as penetration testing, red teaming or academic teaching. Another ethical issue pertains to transparency and accountability in countermeasure enabled side-channel systems. Developers are obligated to make mitigation mechanisms publicly accessible and well-documented to generate confidence in cryptographic system security. The study also raises questions about responsibility like if an AI model or automated tool flags a cryptographic vulnerability or predicts a key incorrectly determining the accountability in misuse or failure scenarios remains complex. These dimensions call for organized protocols on the creation, utilization and handling of side-channel analysis tools for real implementations.

Future directions in timing-based side-channel research

The effectiveness of both used approaches creates several opportunities for future research in time-based side-channel analysis. One of the potential directions is to extend the machine learning model to accommodate other cryptographic algorithms like AES or light-weight ciphers in IoT settings. Extending the dataset through the inclusion of real-world timing data from diverse devices other than virtual machines would also allow the stability and usability of the predictive model to be enhanced. Additionally future research can investigate ensemble learning models with various kinds of side-channel features (timing, memory access patterns and cache behavior) for even more precise key inference. Another useful extension is to improve the user interface and visualization system so that researchers or students can interactively investigate key prediction outputs and similarity metrics in real time. Moreover incorporating this side-channel toolkit into cybersecurity courses, online laboratories or Capture The Flag (CTF) setups would be an efficient platform for practical training. On the defensive side this work can also inform the design of lightweight timing obfuscation methods and noise injection mechanisms that are particularly tailored to low-resource or embedded hardware platforms like Raspberry Pi, Arduino or IoT devices.

SUMMARY OF STUDENT'S CONTRIBUTION

My effort in this research project was to develop and deploy a system for the analysis of time-based side-channel attacks with a particular emphasis on side-channel attacks on symmetric key encryption algorithms like DES. This research makes a significant contribution to the area of how cryptographic systems can be attacked based on measurable timing features.

More specifically I developed a complete analysis module for determining DES key predictability in the situation of pseudo-random key generation from Unix timestamps. I implemented the key generation process and developed similarity comparison algorithms and devised methods for determining the closest matching keys using both byte-level matching and bit-wise Euclidean distance. This demonstrated how predictable key generation methods significantly reduce the effective key space and create serious security weaknesses.

I also created in parallel a side-channel dataset containing encryption time, Hamming weights and Hamming distances of diverse plaintext-key pairs. I used the dataset to train a Long Short-Term Memory (LSTM) model to predict DES key bits from encryption time and engineered features. My tasks included dataset preparation, input feature selection, model designing and performance testing on out-of-sample data.

With this project I presented a double-pronged method to side-channel analysis with a focus on the dangers of both inadequate key randomness and timing leakage. My work lays a solid groundwork for future cryptographic implementation security research and contributes towards increasing awareness of the effects of timing-based side-channel vulnerabilities.

CONCLUSION

This project explores and demonstrates the feasibility of exploiting time based side-channel vulnerabilities in cryptographic implementations with the specific focus of DES encryption. By developing two novel but complementary approaches timestamp-based deterministic key generation analysis and LSTM-based deep learning key bit prediction. And this project has opened up new perspectives on the real-world exploitability of timing information in cryptographic routines. Both approaches were successfully implemented, validated and integrated into a lightweight, transportable platform on a Raspberry Pi 4 that confirms the practicality of these methods in resource constrained settings and in educational environments.

The first method addressed the often overlooked issue of weak randomness in key generation particularly when pseudo-random number generators are seeded using predictable values such as Unix timestamps. This timestamp based analysis attack modeled a real attack situation in which an attacker with approximate knowledge of when a key was generated might obtain a set of candidate keys by enumerating the target range of timestamps. By byte-level and bit-level comparisons such as Euclidean distance analysis this approach could detect keys that have a close resemblance to the target key. The findings were that even a modest narrowing of the generation time window can substantially decrease the key space in effect compromising the security assurances of symmetric key encryption. This result adds additional strength to the paramount importance of secure nondeterministic sources of entropy in critical scheduling algorithms. Particularly in embedded or low-power devices that tend to utilize system time as randomness.

The second approach in this study focused on the predictive power of modern deep learning models in the context of side channel analysis. By gathering and creating a high-quality dataset comprising encryption timings, Hamming weights and distances a sequence learning model and more precisely an LSTM network was trained for predicting DES key bits. The model was able to pick up on fine-grained timing differences and temporal dependencies that are usually invisible to normal statistical models or more straightforward machine learning classifiers. The results of predictions were always indicative of the capacity of the model to deduce major parts of the DES key exclusively from side-channel information without any direct access to encryption hardware or internal state. This made it clear that even relatively small leaks of timing information could be used to compromise cryptographic secrecy especially when combined with advanced pattern recognition capabilities offered by deep learning techniques.

One of the most important elements of this research was that it was driven by real world deployment. Unlike many theoretical or lab-bound studies that require expensive oscilloscopes or complex measurement setups the solutions presented here were intentionally developed for use on low-cost and widely available hardware. The Raspberry Pi was the testbed and demonstration platform for experimentation and the point was being made that the system could produce required inference, prediction and visual output in real time. This not only confirmed the system's feasibility within a controlled environment but also demonstrated it to be a useful teaching aid for cryptographic awareness and cybersecurity education.

Besides integrating both methods in a single cohesive system enabled us to showcase an end to end attack narrative from weaknesses in key generation using predictable timestamps to advanced side-channel inference using machine learning. The modularity of the system facilitates future extensions which include switching to other cryptographic standards adding additional side-channel features or improving real-time visualization. This makes the system a research prototype as well as a scalable basis for future research in information security and applied cryptography.

Despite the success the research also pointed out several limitations. The performance of predictive models relied on the quality and regularity of timing data that may be harder to obtain in uncontrolled environments. Similarly the effectiveness of the timestamp based method depends on the attacker having some temporal knowledge about when the key was created. These are not weaknesses that undermine the research's main point but, strengthen the case for strong implementation practice and highlight the reality that even "minor" implementation choices like when and under what circumstances keys are created and carry significant security implications.

In summary this research is a valuable contribution to the side-channel analysis community in that it shows how time a parameter is often ignored or underestimated in cryptographic contexts and is exploited or approximated to reveal secret keys using both determinist modeling and data-driven inference. It closes the gap between theoretical vulnerability and actual exploitability which makes it particularly useful for cybersecurity researchers, penetration testers and educators. As cryptographic systems continue to advance the research in this work acts as an important reminder that implementation-level details are equally as important as algorithmic security and that the future of cryptographic security needs to account for both mathematical and physical-layer robustness.

REFERENCES

[1] R. MacDonald, "What Is Symmetric Encryption, How Does It Work & Why Use It," 1kosmos, 14 August 2023. [Online]. Available: https://www.1kosmos.com/blockchain/symmetric-encryption/.

[2] Sri Harsha Mekala, Zubair Baig, Adnan Anwar, Sherali Zeadally, "Cybersecurity for Industrial IoT (IIoT): Threats, countermeasures, challenges and future directions," ScienceDirect, 1 August 2023. [Online]. Available: https://www.sciencedirect.com/science/article/abs/pii/S0140366423002189.

[3] geeksforgeek, "What is a Side-Channel Attack? How it Works," geeksforgeek, 21 June 2024. [Online]. Available: https://www.geeksforgeeks.org/what-is-a-side-channel/.

[4] Dag Arne Osvik, Adi Shamir, Eran Tromer, "Cache Attacks and Countermeasures: the Case of AES," Department of Computer Science and Applied Mathematics, Weizmann Institute of Science, 2005.

[5] P. C. Kocher, "Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems," paulkocher, 1996.

[6] D. B. David Brumley, Dan Boneh, "Remote Timing Attacks are Practical," ePrinter Archive, 2011.

[7] Liran Lerman, Romain Poussier, Olivier Markowitch, François-Xavier Standaert, "Template attacks versus machine learning revisited and the curse of dimensionality in side-channel analysis:extended version," lerman, 2018.

[8] Hai Huang,Jinming Wu, Xinling Tang, Shilei Zhao, Zhiwei Liu, Bin Yu, "Deep learning-based improved side-channel attacks using data denoising and feature fusion," plos one, 2025.

[9] Scott Yilek, Eric Rescorla, Hovav Shacham, Brandon Enright, Stefan Savage, "When private keys are public: Results from the 2008 Debian OpenSSL vulnerability," hovav, 2008.

[10] Alex Biryukov, Joppe W. Bos, Samuel Picek, Vesselin Velichkov, "Random number generators: Pitfalls and improvements," IACR ePrint Archive, 2020.

[11] Stefan Mangard, Elisabeth Oswald, Thomas Popp, "Power Analysis Attacks: Revealing the Secrets of Smart Cards," iacr, 2007.

[12] Dag Arne Osvik, Adi Shamir, Eran Tromer, "Cache Attacks and Countermeasures: the Case of AES," eprint, 2005.

[13] Florian Arnold, Holger Hermanns, Reza Pulungan, Mari¨elle Stoelinga, "Time-Dependent Analysis of Attacks," depend cs unni, 2014.

[14] Alex Biryukov, Jean-Philippe D'Anvers, Šimon Picek, Vasil Velichkov, "Random number generators: Pitfalls and improvements," IACR ePrint, 2020.

[15] B. Lutkevich, "timestamp," [Online]. Available: https://www.techtarget.com/whatis/definition/timestamp.

[16] v. chugani, "Understanding Euclidean Distance: From Theory to Practice," datacamp, 13 september 2024. [Online]. Available: https://www.datacamp.com/tutorial/euclidean-distance.

[17] A. Shafi, "Random Forest Classification with Scikit-Learn," datacamp, 1 October 2024. [Online]. Available: https://www.datacamp.com/tutorial/random-forests-classifier-python.

[18] S. Saxena, "What is LSTM? Introduction to Long Short-Term Memory," analyticsvidhya, 30 December 2024. [Online]. Available: https://www.analyticsvidhya.com/blog/2021/03/introduction-to-long-short-term-memory-lstm/.