ABSTRACT

YAMAN, FERHAT. AgentSCA: Advanced Physical Side Channel Analysis Agent with LLMs. (Under the direction of Samira Mirbagher Ajorpaz).

Side-channel attacks pose a significant threat to security-critical systems, making the development of effective countermeasures a pressing research challenge. In recent years, the use of language models has shown promising results in various natural language processing tasks. This thesis explores the application of Language Model-based Learning (LLM) techniques to creating side-channel analysis agents, specifically focusing on Prompt Engineering, Fine-tuning LLM, and Fine-tuning LLM with Human Feedback Reinforcement Learning methods. The main objective of this research is to compare the performance of these three approaches in two different side-channel analysis scenarios: AES side-channel analysis and deep-learning accelerator side-channel analysis. The AES case represents a classical encryption method, while the deep-learning accelerator case represents a more complex and contemporary setting. The success rates of each method in accurately analyzing the side-channel information will be evaluated and compared.

The first approach, Prompt Engineering, involves designing specific prompts to guide the model towards understanding side-channel information and improving its analytical capabilities. This method leverages prior knowledge and expertise about side-channel analysis to explicitly encode instructions. The second approach, Fine-tuning LLM, focuses on using pre-trained language models and further fine-tuning them using side-channel analysis datasets. This method eliminates the need for manual prompt engineering and allows the model to learn from real-world scenarios. The third approach, Fine-tuning LLM with Human Feedback Reinforcement Learning, incorporates human feedback into the fine-tuning process. This method aims to bridge the gap between human expertise and machine learning models by leveraging reinforcement learning techniques. The model is trained with a combination of positive and negative side-channel analysis examples to improve its accuracy.

To evaluate the performance of these methods, we conduct experiments using side-channel analysis tasks on both AES and deep-learning accelerator scenarios. The success rates of each approach in correctly analyzing the side-channel information are measured and compared. The results of this study will provide insights into the effectiveness of different LLM-based methods for creating side-channel analysis agents. This research has the potential to contribute to the development of robust countermeasures against side-channel attacks by leveraging the power of language models and reinforcement learning.

AgentSCA: Advanced Physical Side Channel Analysis Agent with LLMs

by
Ferhat Yaman

A thesis submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Master of Science

Computer Engineering

Raleigh, North Carolina
2023

APPROVED BY:

Paul Franzon

James Tuck

Samira Mirbagher Ajorpaz
Chair of Advisory Committee

DEDICATION

To my lovely wife.

BIOGRAPHY

Ferhat Yaman is a graduate student at NC State University pursuing a Master's degree in Computer Engineering. He holds a Bachelor's degree in both Computer Science and Electronics Engineering from Sabanci University, where he gained expertise in software development, hardware design, and secure systems. During his undergraduate studies, Ferhat interned at research institutes, focusing on security analysis of systems and algorithms. These experiences ignited his interest in cryptography, cybersecurity, and machine learning. In his master's thesis, Ferhat aims to explore novel techniques for side-channel analysis of various systems and its relationship with machine learning.

ACKNOWLEDGEMENTS

I am deeply grateful to my thesis advisor, Dr. Samira Mirbagher Ajorpaz, for their invaluable guidance and support throughout my research. Their expertise and insightful feedback have greatly influenced the direction of this work. I would also like to thank the committee members, Dr. James Tuck and Dr. Paul Franzon, for their contributions and encouragement. I am thankful to Dr. Aydin Aysu and the members of HECTOR Lab for their valuable discussions. Lastly, I extend my heartfelt appreciation to my family and friends for their unwavering support during this challenging yet fulfilling journey.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER

1

INTRODUCTION

Side-channel attacks have emerged as a significant concern in the field of computer security. Traditional cryptographic methods, such as the Advanced Encryption Standard (AES), may be vulnerable to attacks that exploit side-channel information leaked by physical implementations (17). Side-channel attacks utilize various types of indirect information obtained from the implementation, such as power consumption (16), electromagnetic radiation (1), or timing measurements (18), to infer the secret key being used. To address this security challenge, researchers have employed various techniques, ranging from hardware countermeasures (26; 30; 28) to software-based approaches (33).

Language models (LMs), which are natural language processing models trained on large corpora of text (31), have demonstrated exceptional performance in various language-related tasks. This success has spurred interest in exploring the potential of LMs in other domains (8), including security and cryptography.

This thesis investigates the use of Language Model-based Learning (LLM) techniques to develop side-channel analysis agents. Three different methods are explored: Prompt Engineering, Fine-tuning LLM, and Fine-tuning LLM with Human Feedback Reinforcement Learning. These methods aim to enhance the accuracy and efficiency of side-channel analysis by leveraging the capabilities of language models.

The primary objective of this research is to compare the performance of these LLM-based methods in two specific side-channel analysis scenarios: AES side-channel analysis and deep-learning accelerator side-channel analysis. The AES scenario represents a well-established encryption scheme, while the deep-learning accelerator scenario represents a more complex and contemporary use case.

The first method, Prompt Engineering(3; 22; 35), involves designing specific prompts to guide the language model's understanding of side-channel information. By leveraging prior knowledge and expertise, this method explicitly encodes instructions into the prompts to enhance the model's analytical capabilities.

The second method, Fine-tuning LLM(34; 19; 13), utilizes pre-trained language models and further fine-tunes them using side-channel analysis datasets. This approach eliminates the need for manual prompt engineering and enables the model to learn from real-world scenarios, thereby improving its accuracy in side-channel analysis tasks.

The third method, Fine-tuning LLM with Human Feedback Reinforcement Learning (25), aims to combine human expertise with machine learning models. This method incorporates human feedback into the fine-tuning process by leveraging reinforcement learning techniques. By training the model with a combination of positive and negative side-channel analysis examples, this method seeks to bridge the gap between human expertise and machine learning models.

To evaluate the effectiveness of the three LLM-based methods, experiments will be conducted using side-channel analysis tasks related to AES and deep-learning accelerator scenarios. The success rates of each method in accurately analyzing the side-channel information will be measured and compared. These comparisons will provide insights into the strengths and limitations of each approach.

The findings of this research hold the potential to contribute to the development of robust countermeasures against side-channel attacks. By harnessing the power of language models, prompt engineering, and reinforcement learning, this study aims to enhance the field of side-channel analysis and improve the security of critical systems.

1.1 Motivation

As a side-channel security analyst, each step in the experimental process involves extensive evaluation and decision-making. These decisions can be time-consuming as there are multiple options to consider for each aspect. For instance, when assessing system leakage, the analyst must carefully select suitable inputs to observe the desired leakage behavior. Furthermore, there are various attack points that can be explored, each yielding potentially different results. Additionally, a deep understanding of the system's implementation details is necessary. Taking the example of a deep learning accelerator attack, the objective is to steal the model, including hyperparameters, weight parameters, and bias parameters for each node in the neural network. Reverse engineering the neural network model requires knowledge of its specific characteristics, such as the number of hidden layers, the number of nodes in each layer, and the type of neural network layers employed. Considering the implementation details of the accelerators is also crucial, encompassing factors such as the number of parallel execution layers, the level of parallelism employed, the data pipeline mechanism, and the functioning of the control path. All of these considerations are critical in determining the most effective attack strategy.

LLMs have demonstrated their utility in various domains, including code generation, security analysis, financial analysis, and more. These applications require expertise to make informed decisions regarding suitable options and selections. In the field of SCA, leveraging the capabilities of LLMs for decision-making processes presents an opportunity to enhance the efficiency and ease of evaluation for analysts. By incorporating LLMs into SCA, analysts can benefit from the model's ability to assist in decision-making tasks, streamlining the overall analysis process. This integration of LLMs in SCA has the potential to optimize the analyst's workflow and contribute to more efficient and effective side-channel analysis techniques.

1.2 Contributions

We make the following contributions in our thesis:

- We propose the first physical side-channel analysis (SCA) agent using Language Model-based Learning (LLM) techniques.
- We demonstrate the effectiveness of LLMs in generating hypothetical power models and attack codes for side-channel analysis tasks.
- We introduce AgentSCA, our developed tool that automates the physical side-channel analysis of various sensitive information-related algorithms.
- We showcase the capability of our tool by successfully exploiting side-channel vulnerabilities in both AES and deep learning accelerators.

These contributions highlight the significance of employing LLMs in the field of side-channel analysis, enabling automated and efficient analysis of cryptographic and deep learning systems.

1.3 Thesis Organization

This thesis is organized into the following chapters:

- Chapter 2: Background and Related Works - Presents a comprehensive review of related work on side-channel analysis, language models, and existing techniques for side-channel attack mitigation and gives a motivational example of this work.
- Chapter 3: Overview and Methodology - Describes the research methodology employed in this study, including data collection, model training, and evaluation metrics.
- Chapter 4: Design - Presents the experimental setup for conducting side-channel analysis tasks on AES and deep-learning accelerator scenarios using the three LLM-based methods.
- Chapter 4: Design - Presents the experimental setup for conducting side-channel analysis tasks on AES and deep-learning accelerator scenarios using the three LLM-based methods.
- Chapter 5: Evaluations - Presents the results obtained from the experiments and conducts a detailed analysis of the performance of each method in different side-channel analysis scenarios.
- Chapter 6: Discussions - Discusses the implications of the findings, limitations of the study, and potential directions for future research.
- Chapter 7: Conclusion - Summarizes the key findings and contributions of the study, reiterates the research objectives, and offers concluding remarks.

Overall, this research aims to advance the field of side-channel analysis by leveraging the power of language models and reinforcement learning techniques. The subsequent chapters delve deeper into the methodology, experiments, and empirical results that form the basis of this research.

CHAPTER

2

BACKGROUND

2.1 Side-Channel Analysis

SCA is a class of attacks that exploit information leaked through side-channel channels to extract secret information, such as encryption keys or sensitive data. Traditional cryptographic algorithms, like the AES, are vulnerable to side-channel attacks due to the physical implementation characteristics of devices. Side-channel attacks rely on the measurement of physical parameters, such as power consumption (16), electromagnetic radiation (7), or timing (18), to infer confidential information. Such attacks can be powerful in practice due to the large amount of leakage information available.

Side-channel attacks can be categorized into various types, including power analysis, timing analysis, electromagnetic analysis, and acoustic analysis. Power analysis attacks, for example, exploit power consumption variations during cryptographic operations to deduce secret information. Timing analysis attacks utilize timing measurements of cryptographic operations to infer sensitive data. These attacks can be categorized further into simple power analysis (SPA), differential power analysis (DPA), correlation power analysis (CPA), and more (21). In SCA, power models are created to estimate the power consumption during the switching of transistors in hardware. These models are based on the values stored in specific registers. Two well-known power models are based on the concepts of Hamming weight and Hamming distance. Hamming weight shown in Equation 2.1 calculates the number of bits that are set to 1 in a register, while Hamming distance measures the number of bit positions in which two binary numbers differ, shown in Equation 2.2. These metrics help capture the changes in power consumption during state transitions in a register.

$$HW(x) = \sum_{i=0}^{n} x_i, \tag{2.1}$$

where x is the register value and x_i represents the bits of x.

$$HD(x, y) = \sum_{i=0}^{n} (x_i \oplus y_i), \tag{2.2}$$

where x and y are register values and x_i and y_i represent the corresponding bits.

To perform SCA, various statistical metrics are employed to detect and analyze the leakage of information. One commonly used metric is the TVLA (Test Vector Leakage Assessment) t-Test, which is used to detect leakage in power consumption. This test assesses the statistical significance of differences in means between power traces collected during different encryption operations. Equation 2.3 calculates t-score

$$T = \frac{\bar{X}_1 - \bar{X}_2}{\sqrt{\frac{{s_1}^2}{n_1} + \frac{{s_2}^2}{n_2}}}, \tag{2.3}$$

where $\bar{X}_1$ and $\bar{X}_2$ are the means of the power consumption during different encryption operations, s_1 and s_2 are the standard deviations, n_1 and n_2 are the number of traces.

To evaluate the strength of the power model in accurately representing the leakage, additional statistical tests are employed. One such test is the Pearson correlation test, which measures the linear correlation between power consumption and the hypothetical power model. This test assesses the strength and direction of the relationship between the variables.

$$\rho_{X,Y} = \frac{cov(X, Y)}{\sqrt{\sigma_X \sigma_Y}}, \tag{2.4}$$

where $\rho_{X,Y}$ is the correlation coefficient between the power consumption and the target variable, $cov(X, Y)$ is the covariance between X and Y, and σ_X and σ_Y are the variances of X and Y.

Another metric used is the signal-to-noise ratio (SNR), which quantifies the ratio of the power signal to the noise present in the measurement. It provides a measure of how effectively the target information can be extracted from the power traces while minimizing the impact of noise. SNR can be calculated by Equation 2.5,

$$SNR = \frac{\sigma_s}{\sigma_n}, \tag{2.5}$$

where σ_s is the variance of the power signal, and σ_n is the variance of the noise.

These statistical metrics play a crucial role in side-channel analysis by providing quantitative measures of the leakage and the effectiveness of the power models. Using these metrics, researchers and practitioners can assess the vulnerability of cryptographic implementations and develop appropriate countermeasures to mitigate the risks associated with side-channel attacks.

SCA attacks are not limited to targeting cryptographic operations; they can also be utilized to recover deep learning models (10). The process of training a deep learning model involves substantial engineering costs and requires significant computing power, making these models valuable intellectual property that needs protection. Recent studies in the literature (37) have demonstrated the feasibility of such attacks, highlighting the importance of developing effective countermeasures to safeguard deep learning models from side-channel attacks.

Several countermeasures have been proposed to mitigate side-channel attacks (17). These include masking techniques, such as randomizing intermediate values, secret sharing, and using constant-time algorithms. Additionally, hardware-level countermeasures like power balancing, electromagnetic shielding, and noise generators have been employed to reduce side-channel leakage. However, side-channel analysis remains an ongoing research area due to the continuous development of new attack methodologies and the discovery of vulnerabilities in cryptographic implementations. To address this challenge, researchers have explored the application of machine learning techniques to enhance and automate side-channel analysis tasks.

2.2 Machine Learning and Deep Learning

Machine learning (ML) and deep learning (DL) have emerged as powerful techniques in various domains, including computer vision (29), natural language processing (14), and cybersecurity. ML algorithms, inspired by the study of pattern recognition and computational learning, enable systems to learn from data and make predictions or decisions without explicit programming.

2.2.1 Machine Learning

Machine learning algorithms can be classified into three main categories: supervised learning(20), unsupervised learning (15), and reinforcement learning (23). Supervised learning involves training a model on labeled data, where the input instances are paired with corresponding target labels. The model learns to predict the labels of unseen instances based on the patterns it observes in the training data. Common supervised learning algorithms include decision trees, support vector machines (SVM), and artificial neural networks (ANN). Unsupervised learning, on the other hand, aims to find patterns in unlabeled data. The model learns to discover hidden structures or relationships in the data without any predetermined target labels. Clustering algorithms, such as k-means and hierarchical clustering, and dimensionality reduction techniques like Principal Component Analysis (PCA) are common examples of unsupervised learning. Reinforcement learning (RL) is a learning paradigm where an agent interacts with an environment and learns to make actions that maximize a long-term reward signal. The agent receives feedback in the form of rewards or penalties based on its actions, and the objective is to find an optimal policy that maximizes the cumulative reward. RL has been successfully applied to various sequential decision-making tasks, such as game playing and robotics.

2.2.2 Deep Learning

Deep learning is a subfield of ML that focuses on training deep neural networks, also known as artificial neural networks with multiple layers. Deep neural networks are capable of learning hierarchical representations of data, enabling them to capture intricate patterns and relationships. Convolutional Neural Networks (CNN) are a popular deep learning architecture commonly used for tasks involving images or spatial data. By using convolutional layers that apply filters to input data, CNNs can learn to capture local patterns and hierarchically combine them to make predictions. Recurrent Neural Networks (RNNs) are another type of deep learning architecture specifically designed for sequential data processing. These networks are capable of capturing dependencies over time, making them well-suited for tasks such as natural language processing and speech recognition. Transformers, introduced by Vaswani et al., are a type of deep learning architecture that has recently gained significant attention. Transformers use self-attention mechanisms to capture global dependencies within the input data, enabling them to process variable-length sequences effectively. Transformers have achieved remarkable successes in various tasks, including machine translation, text generation, and image recognition. Deep learning models are typically trained using a variant of stochastic gradient descent called backpropagation. During training, the model adjusts the weights and biases of its layers based on the discrepancy between predicted outputs and ground truth labels. The performance of deep learning models heavily relies on large-scale annotated datasets and computational resources. Several deep learning frameworks, such as TensorFlow, PyTorch, and Keras, have made it easier for researchers and developers to design, train, and deploy deep learning architectures for various applications.

2.2.3 Deep Learning Accelerators

Deep learning accelerators (4), also known as neural processing units (NPUs), are specialized hardware components designed to accelerate deep neural network computations. As deep learning models have become increasingly complex and computationally demanding, the need for efficient hardware solutions has arisen. Deep learning accelerators offer a significant performance boost by offloading and accelerating computations, ultimately reducing the inference time of deep neural networks. These accelerators are designed to exploit the inherent parallelism and regularity present in deep learning algorithms. They typically consist of dedicated processing units, such as graphical processing units (GPUs) or application-specific integrated circuits (ASICs), optimized for matrix multiplication and other operations commonly found in deep neural networks. By providing highly parallel computing capabilities, deep learning accelerators enable faster and more energy-efficient inference tasks.

Various deep-learning accelerator architectures have been developed, each with its own trade-offs in terms of performance, power consumption, and flexibility. GPUs have gained popularity due to their general-purpose nature and the availability of powerful libraries, such as NVIDIA CUDA, which provide an abstraction layer for developing deep learning applications. ASIC-based accelerators, on the other hand, can achieve higher efficiency by customizing the hardware specifically for deep learning computations. Examples of deep learning accelerators include NVIDIA's Tensor Cores (24), Google's Tensor Processing Units (TPUs)(9; 12), and Intel's Neural Compute Stick (11). These accelerators have been widely adopted in various domains, including image and speech recognition, natural language processing, and autonomous vehicles, due to their ability to handle the computational requirements of large-scale deep learning models.

In the next section, we will delve into the concept of Language Model-based Learning (LLM), which forms the foundation for the proposed methods in this thesis.

2.3 Large Language Models

Language Model-based Learning (LLM) has emerged as a powerful technique in natural language processing, enabling machines to understand and generate human-like text. LLMs capture the statistical patterns and dependencies in text data to generate coherent and contextually relevant language.

LLMs have been primarily developed and applied in tasks such as machine translation, language modeling, text generation, and sentiment analysis. However, researchers have started exploring the potential of LLMs in other domains, including security and cryptography.

2.3.1 Prompt Engineering

Prompt Engineering is a method that leverages prior knowledge and expertise to guide the language model's understanding of specific tasks. It involves designing appropriate prompts to elicit the desired behavior from the model. In the context of side-channel analysis, Prompt Engineering can be used to explicitly encode instructions related to analyzing side-channel information and improve the analytical capabilities of the LLM.

Previous studies have demonstrated the effectiveness of Prompt Engineering in various domains. For instance, Wei et al. showed how prompt engineering can be used to improve question-answering capabilities in LLMs. In the context of side-channel analysis, Prompt Engineering can guide the LLM to focus on relevant features and learn patterns associated with side-channel information leakage.

2.3.2 Fine-tuning

Fine-tuning is a technique that takes pre-trained language models and further adapts them to specific tasks using domain-specific data. By using pre-trained models as a starting point, fine-tuning allows for faster convergence and improved performance on the target task. In the context of side-channel analysis, fine-tuning LLMs can be instrumental in accurately analyzing side-channel information by leveraging knowledge learned from large-scale language tasks.

Previous research has shown the effectiveness of fine-tuning language models in various applications. For instance, Raffel et al. demonstrated improved text generation performance by fine-tuning pre-trained models on domain-specific data. Fine-tuning LLMs with side-channel analysis datasets can enable the model to capture patterns relevant to side-channel information leakage and enhance its analytical capabilities.

2.3.3 Reinforcement Learning with Human Feedback

Reinforcement Learning with Human Feedback (5) is an approach that combines human expertise with reinforcement learning techniques. This method involves incorporating feedback from human experts during the training process to guide the model's learning and improve its performance. In the context of side-channel analysis, this approach can bridge the gap between human expertise and machine learning models, resulting in more accurate analysis of side-channel information.

Several studies have explored the use of reinforcement learning with human feedback in different domains. For example, Christiano et al. demonstrated the effectiveness of this approach in complex control tasks. In the context of side-channel analysis, reinforcement learning with human feedback can involve incorporating feedback about correct and incorrect side-channel information in the training process to improve the model's accuracy.

These LLM-based techniques offer promising avenues for developing efficient side-channel analysis agents with improved analytical capabilities. The subsequent chapters of this thesis delve into the specific methodologies, experiments, and evaluations of these techniques in the context of side-channel analysis.

CHAPTER

3

OVERVIEW AND METHODOLOGY

The process of conducting side-channel analysis entails significant analysis time and decision-making. It is essential to understand the architecture and mechanisms of the specific secret execution algorithm under investigation. Moreover, a foundational understanding of SCA, including statistical comprehension and hypothetical power model generation, is required. The complexity of SCA varies depending on the tested environment, whether it be a microprocessor, specifically designed ASIC accelerator, or FPGA. Each environment poses unique challenges that must be tackled, necessitating experience and expertise in handling these diverse complexities. Three main categories encompass the complexity spectrum of SCA:

- **Algorithmic Complexity:** Proper understanding of the algorithms being analyzed is paramount. It is crucial to identify where the secret is executed within the algorithmic flow. This knowledge allows for the selection of appropriate attack points, specifically pinpointing areas where the introduction of secret information or sequential operations might result in leakage. For example, in the case of RSA side-channel timing attacks (18), the timing differences in multiply and squaring operations reveal secret information during conditional evaluations.

- **Analysis Complexity:** SCA requires knowledge of how to utilize statistical metrics to assess the analysis effectively. For instance, the TVLA (t-test) is commonly employed as an initial evaluation measure in SCA. However, even if leakage is present in the TVLA t-test, it does not necessarily guarantee a successful attack.

- **Architectural Complexity:** Multiple approaches exist for implementing algorithms, introducing additional complexities when considering memory systems and control paths. Various environments utilize secret-related operations, such as CPUs, GPUs, FPGAs, and ASIC accelerators explicitly designed for accelerating these operations. Each environment behaves distinctively from the perspective of , necessitating careful consideration of architectural complexities.

These categories of complexity demonstrate the inherent challenges involved in conducting SCA, which encompass a broad range of tasks, including understanding cryptographic algorithms, selecting optimal attack points, performing statistical assessments, and navigating diverse architectural implementations. Successfully addressing these complexities demands a high level of expertise and experience in the field of SCA. However, in order to make the analysis process more accessible and efficient, we propose leveraging the capabilities of LLMs. By utilizing LLMs, we aim to reduce the time required to obtain the necessary expertise and alleviate the burden of experience associated with each of these challenges. Through the integration of LLMs into methodologies, analysts can benefit from the model's ability to assist in decision-making processes, offer insights, and provide guidance in navigating the complexities of SCA. This approach has the potential to streamline the analysis workflow, enabling analysts to make more accurate and informed choices, and ultimately improving the efficiency and effectiveness of techniques.

In this study, we propose an agent that leverages a combination of human expert knowledge and learning from its own decision outcomes. To achieve this, we employ various techniques including prompt engineering, fine-tuning, and reinforcement learning with human feedback. We aim to compare the effectiveness of these techniques in two distinct side-channel analysis cases: the well-studied case of AES SCA and the emerging case of machine learning accelerator SCA.

To evaluate the performance of our proposed agent, we create a test environment that enables us to assess the output of the LLMs based on generated code and information. Our goal is to optimize the system in order to generate accurate outputs and steer the SCA experiments towards successful outcomes. In this test environment, we prepare 15 different SCA tasks that cover a range of scenarios and challenges.

For TPU configurations in the test environment, we use Coral Dev Board in this work. This board is a single computer that consists of different communication and memory modules. To collect EM measurements from TPU, we apply several modifications such as removing the

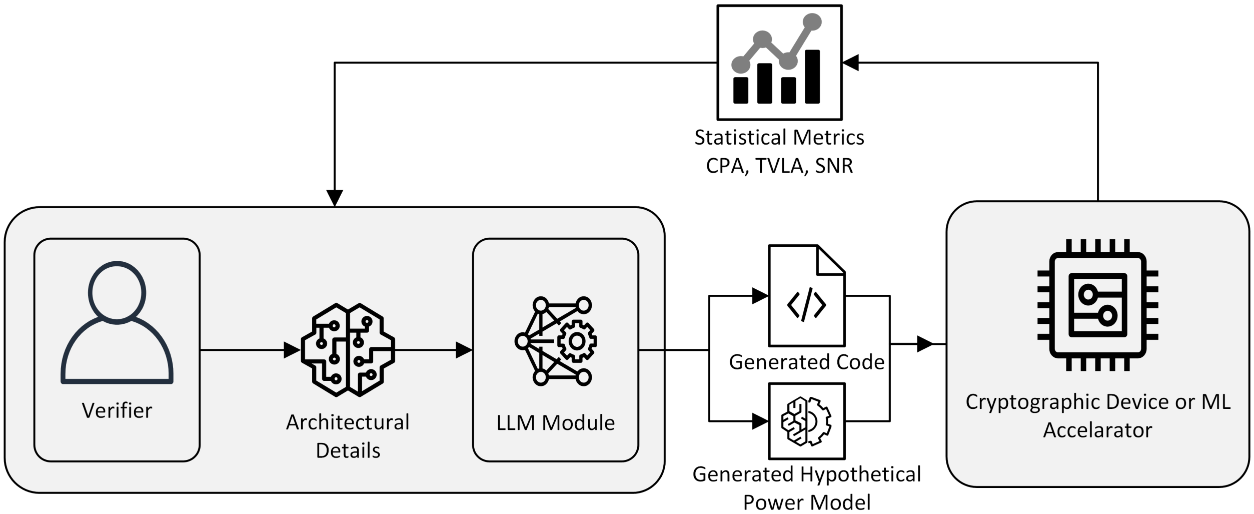

Figure 3.1: System Overview: AgentSCA starts with initial architectural information related to the secret dependent operation. LLM module interprets given information and generated attack code and hypothetical power model. Later, it refines its generation based on the feedback statistical metrics returned from the test design.

heat sink and package cover on the system on module (SoM). After the modifications, we are able to find the location of TPU using the datasheet of SoM (2). To scan the chip surface by Riscure's High Sensitivity EM probe, we place the board into Riscure's EM Probe Station shown in Figure 4.3. EM probe sends collected traces into PicoScope 6424E oscilloscope, then the oscilloscope starts recording of EM signals when Coral Dev Board sent a trigger signal at the beginning of the inference phase. All of those operations are synchronized by the host machine which gets EM measurements from oscilloscope and output of inference from the board.

For each task, we feed the input into the system and assess the results provided by the agent. These results are then evaluated using the test system, which compares them against the expected answers. By quantitatively analyzing the correctness of the agent's responses, we can assess the performance and effectiveness of the different techniques utilized.

By conducting experiments in both the AES and machine learning accelerator SCA cases, we aim to provide a comprehensive comparison of the techniques employed. This comparative analysis will help us understand the strengths and weaknesses of each approach, shedding light on their suitability for different SCA scenarios. Through this methodology, we strive to evaluate the performance of our agent by assessing the success rates and accuracy of the outcomes it generates. This evaluation process serves as the basis for comparing the effectiveness of prompt engineering, fine-tuning, and reinforcement learning with human feedback in the context of SCA.

CHAPTER

4

DESIGN

We develop three distinct systems, each contributing to the final AgentSCA's capabilities. The first system utilizes prompt engineering, involving the training of LLMs with carefully designed prompts. This process entails a continuous iteration of prompting, correcting, and steering to guide the models toward the desired outcomes. By incorporating explicit instructions and prior knowledge in the prompts, we aim to enhance the LLMs' understanding of side-channel analysis tasks. The second system focuses on fine-tuning the LLMs to refine their analytical capabilities. We leverage various resources such as lecture notes, books, and papers on SCA to curate a specific dataset for the fine-tuning process. By exposing the LLMs to SCA-specific content, we enable them to absorb domain-specific knowledge and improve their performance in analyzing side-channel information. The third system integrates reinforcement learning with human feedback into the fine-tuning process. This approach aims to bridge the gap between machine learning models and human expertise in side-channel analysis. By incorporating reinforcement learning techniques, the model is trained using a combination of positive and negative side-channel analysis examples. Human feedback is utilized to provide guidance and further augment the learning process, enabling the agent to continually improve its performance.

To evaluate the performance and effectiveness of the LLM-based side-channel analysis agents, a comprehensive test system is incorporated. This system is designed to assess the outcomes of the LLM models and provide feedback for further interpretation and improvement. In our experimental setup, we focus on conducting electromagnetic measurements on two distinct targets: the Pinata ARM M4 microprocessor from Riscure and the Coral EdgeTPU from Google. The Pinata ARM M4 is selected to represent the AES side-channel analysis scenario, while the Coral EdgeTPU represents the deep-learning accelerator side-channel analysis scenario. By conducting electromagnetic measurements on these targets, we aim to capture the side-channel information leaked during the execution of cryptographic operations or deep-learning tasks. These measurements provide valuable data to analyze and assess the proficiency of the LLM-based side-channel analysis agents in accurately extracting the sensitive information.

Once the LLM models have been trained and fine-tuned, the outcomes are evaluated and compared to assess their success rates in accurately detecting and analyzing the side-channel information. The success rates are calculated by comparing the predictions made by the LLMs to the ground truth information. By analyzing discrepancies and patterns in the results, further iterations of fine-tuning can be performed, taking into account human expertise and reinforcement learning techniques.

Overall, the experimental setup involving the Pinata ARM M4 and Coral EdgeTPU targets allows us to thoroughly evaluate the performance and effectiveness of the LLM-based side-channel analysis agents in distinct side-channel analysis scenarios. The collected measurements, the training process, and the feedback loop contribute to refining the models and enhancing their capabilities for accurately detecting and analyzing side-channel information.

4.1 SCA using Prompt Engineering

The prompt engineering process plays a crucial role in enhancing the performance of our side-channel analysis agent. In this study, we employ various techniques for prompt engineering, including zero-shot prompting, few-shot prompting, and chain-of-thought prompting. These techniques aim to guide the language model in producing accurate and meaningful outputs. The desired system, as illustrated in the figure below, demonstrates the integration of these prompt engineering techniques. By combining zero-shot prompting, few-shot prompting, and chain-of-thought prompting, we aim to maximize the agent's capability to analyze side-channel information and produce correct outputs.

Figure: [Insert figure demonstrating the desired system]

Zero-shot prompting involves providing a prompt that allows the agent to perform side-channel analysis without any additional training on specific examples. This method leverages the general knowledge learned by the language model and its ability to transfer that knowledge to new tasks. By crafting a well-designed prompt that encompasses the necessary information, we enable the agent to generate the desired output aligned with side-channel analysis.

An example of zero-shot prompting for attack code generation of the hamming weight power model for AES first-round key addition is depicted in the Listing 1 provided. Zero-shot prompting involves providing comprehensive details regarding the task at hand, aiming to enhance the language model's interpretive capability and identify similar patterns from the given tasks. A full conversation illustrating the prompt and the model's response is included in the Appendix B.1 for reference. However, it is important to note that in this particular instance of zero-shot prompting, the model fails to generate the requested correct power model. Despite the detailed information provided in the prompt, the language model's interpretation falls short in generating the desired output. This case highlights the challenges and limitations associated with zero-shot prompting techniques. Despite its potential and the efforts taken to provide explicit details, the language model may still struggle to produce the desired results. This underscores the need for further fine-tuning and exploration of alternative approaches to improve the interpretive capabilities of language models in side-channel analysis tasks.

Expert - Agent Zero-Shot Prompt

Expert: My objective is to conduct side-channel analysis on the AES cryptographic algorithm, specifically focusing on the round key addition in the first round using the hamming weight power model. I possess a dataset consisting of 1000 power measurements of AES execution, each containing 100 sample points, with a corresponding plaintext for each power measurement. Notably, the correct key is unknown. Can you write a correlational power analysis code for this task?
Agent: Generates $< code >$ that attacks substitution box and explanations how to modify code.

Listing 1: The example conversation provided by AgentSCA demonstrates a zero-shot scenario where the agent is unable to generate the correct code for AES round-key addition analysis, despite receiving detailed instructions.

In contrast, few-shot prompting involves training the language model on a limited number of side-channel analysis examples. This method allows the agent to learn more task-specific knowledge, improving its understanding of side-channel information and enhancing its performance in generating accurate outputs. By carefully selecting and presenting a set of examples, the agent can acquire relevant patterns and inferences crucial for successful side-channel analysis.

Building upon our previous example, we further enhance the performance of our side-channel analysis agent through the use of few-shot prompting. Specifically, we introduce correct examples of code generation from previous prompts, as illustrated in the provided Listing 2. By incorporating these examples into the prompts, the language models (LLMs) gain a deeper understanding of the code structure and the expected reasoning process. Consequently, the agent is able to generate the correct and optimized version of the code. The conversation between the agent and the LLMs, capturing the prompt exchanges and the resulting code generation, is documented in Appendix B.2. This conversation demonstrates the successful application of few-shot prompting in guiding the agent toward accurate and optimized code generation. The LLMs' comprehension of the code structure and their ability to reason based on the provided examples contribute to the agent's generation of the desired outcome. Through the integration of few-shot prompting and the utilization of correct code examples from previous prompts, we illustrate the effectiveness of this approach in enhancing the agent's performance. By leveraging the LLMs' understanding of code structure and their ability to generalize from past examples, our side-channel analysis agent achieves improved code generation capabilities, producing more accurate and optimized outputs.

Expert - Agent Few-Shot Prompt

Expert: Here is an example of side-channel analysis on the AES cryptographic algorithm, specifically focusing on the round key addition in the first round using the hamming weight power model. I possess a dataset consisting of 1000 power measurements of AES execution, each containing 100 sample points, with a corresponding plaintext for each power measurement. Notably, the correct key is unknown. You can find the *correct code* below.
Agent: Corrects some syntax errors on the *correct code* and mentions the hardness of SCA analysis with that many traces.
Expert: Can you generate code for the round key addition stage of the first round of AES?
Agent: Generates *correct optimized code* that attacks round key addition.

Listing 2: The example demonstrates that AgentSCA is capable of providing an optimized version of the correct code, showcasing the power of few-shot prompting in generating accurate code.

In the chain-of-thought prompting technique, a sequence of prompts is provided to guide the agent's analysis step-by-step. Each prompt builds on the previous one, allowing the agent to progressively refine and generate more accurate outputs. This method ensures that the agent considers relevant contextual information and avoids over-reliance on a single prompt.

In our experimentation, we employ the chain-of-thought prompting technique as demonstrated in the example provided in the Listing 3. The side-channel analysis agent demonstrated proficiency in providing accurate answers for each step of the analysis process. However, we observe limitations when the agent encountered complexity in combining these individual steps with specific parameters or requirements. To provide further insight into the agent's performance, we include the entire conversation in Appendix B.3. This transcript showcases the power of the step-by-step solution approach and highlights the agent's ability to generate accurate outputs and interpretations at each stage of the analysis process.

Recognizing the areas where the agent still falls short in terms of handling parameter-specific scenarios, we explore incorporating the few-shot prompt technique. By training the agent on a limited number of additional side-channel analysis examples that involve specific parameter considerations, we anticipate an improvement in the agent's ability to handle and generate accurate outputs in such scenarios.

Expert - Agent Chat Prompts

Expert: My objective is to conduct side-channel analysis on the AES cryptographic algorithm, specifically focusing on the round key addition in the first round using the hamming weight power model. I possess a dataset consisting of 1000 power measurements of AES execution, each containing 100 sample points, with a corresponding plaintext for each power measurement. Notably, the correct key is unknown. Lets think step by step,

- Select a Hypothesis for the First Round Key
- Calculate Hamming Weight of Intermediate Values
- Create Traces and Plaintexts
- Create a Hypothesis Table:
- Compute Correlation

Can you provide code for each step of this correlational power analysis?

Agent: Generates *near-correct code* that attacks round key addition.

Listing 3: Expert - Agent Chain-of-Thought Prompts

Through the use of prompt engineering techniques, we enhance the performance and accuracy of our side-channel analysis agent. By carefully designing prompts that effectively guide the language model, we aim to leverage its capabilities and improve its ability to detect and analyze side-channel information accurately. These prompt engineering techniques are instrumental in shaping the behavior and effectiveness of the side-channel analysis agent.

4.2 SCA using Fine-Tuning

In this study, the fine-tuning of the Language Model-based Learning (LLM) approach is performed to develop our side-channel analysis agent, AgentSCA. We utilize OpenAI APIs for the fine-tuning process, leveraging their advanced language model capabilities.

To prepare for the fine-tuning phase, we curate a comprehensive dataset consisting of context information pertaining to side-channel analysis and machine learning accelerators. This dataset is sourced from various academic papers, books, and lecture notes related to the field. As a preprocessing step, the collected information is organized and prepared to be fed into the base LLM model efficiently.

During the training phase, the base LLM model is fine-tuned using the curated dataset, which includes crucial context information relevant to side-channel analysis. By training the LLM model with this abundant knowledge, we aim to enhance its understanding and proficiency in the field of side-channel analysis. The availability of this specific contextual information ensures that the model grasps the intricacies of side-channel attacks and the methodologies used to mitigate them.

The output training mechanism, illustrated in the Figure 4.1, symbolizes the process of training the LLM model with the relevant context information. This training mechanism facilitates the alignment of the model with the specific requirements and objectives of side-channel analysis. By fine-tuning the model with appropriate data and context, we aim to develop an optimized side-channel analysis agent capable of accurately analyzing and interpreting side-channel information.

The utilization of OpenAI APIs and the fine-tuning process form crucial components of our design for the side-channel analysis agent. This approach allows us to leverage the advanced capabilities of language models and incorporate specific contextual information to enhance the agent's understanding and performance in side-channel analysis tasks. The use of curated datasets and the fine-tuning procedure ensures that the model is trained with appropriate context and knowledge, enabling it to generate accurate and meaningful outputs.

Overall, the fine-tuning of the LLM model with context information is a fundamental step in the design of our side-channel analysis agent. It empowers the agent with an in-depth understanding of side-channel analysis techniques and strengthens its ability to effectively

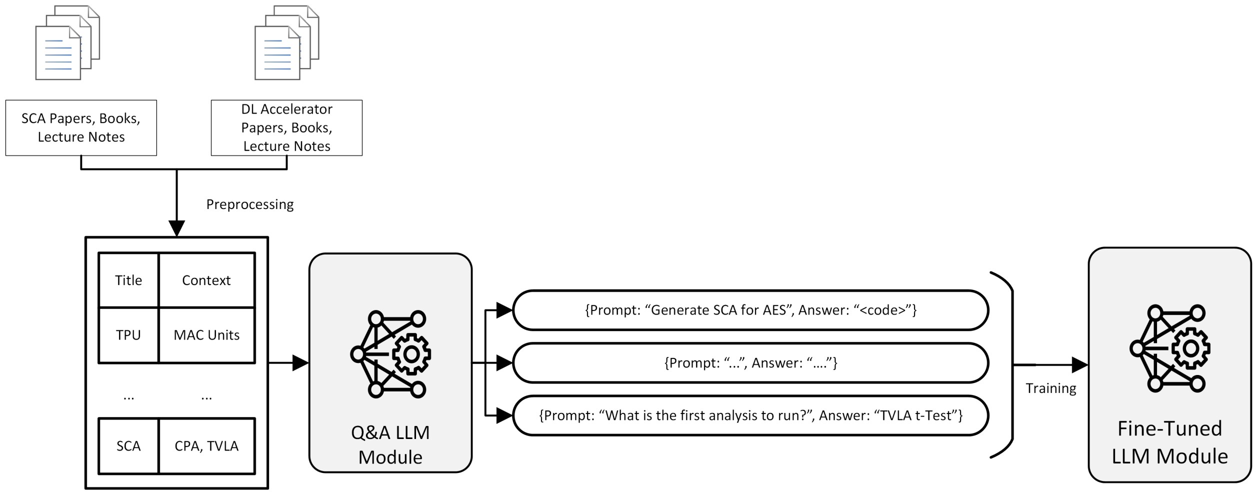

Figure 4.1: AgentSCA with Fine Tuning: First, we begin by preprocessing the collected resources on SCA and DL accelerators. Subsequently, we generate prompts and corresponding answers using the Question and Answer (Q&A) LLM. Finally, we train the base LLM module using the prompts and answers generated from earlier steps.

interpret and analyze side-channel information. The combination of OpenAI APIs, curated datasets, and appropriate training mechanisms play a vital role in promoting the development of an optimized side-channel analysis agent.

4.3 SCA using Reinforcement Learning with Human Feedback

To assess the performance of the fine-tuned models, we evaluate the outcome of each task and assign a score ranging from 1 to 5. This score reflects both the correctness and closeness of the solution obtained by the models. Drawing upon these evaluation metrics, we establish a reward mechanism through the implementation of reinforcement learning techniques. The subsequent Figure 4.2 provides a visual representation of the detailed flow of this process, elucidating the steps involved in training the models and optimizing their performance.

4.4 Hardware Platform

4.4.1 Finding Attack Point Location

The high-level architecture of TPU is given by Google's patent publication (Woo et al.). However, the actual evaluation location of secret values is unknown. We scanned the TPU using the setup shown in Figure 4.4 to find the operating point. We divided chip surface 10 by 10 grid

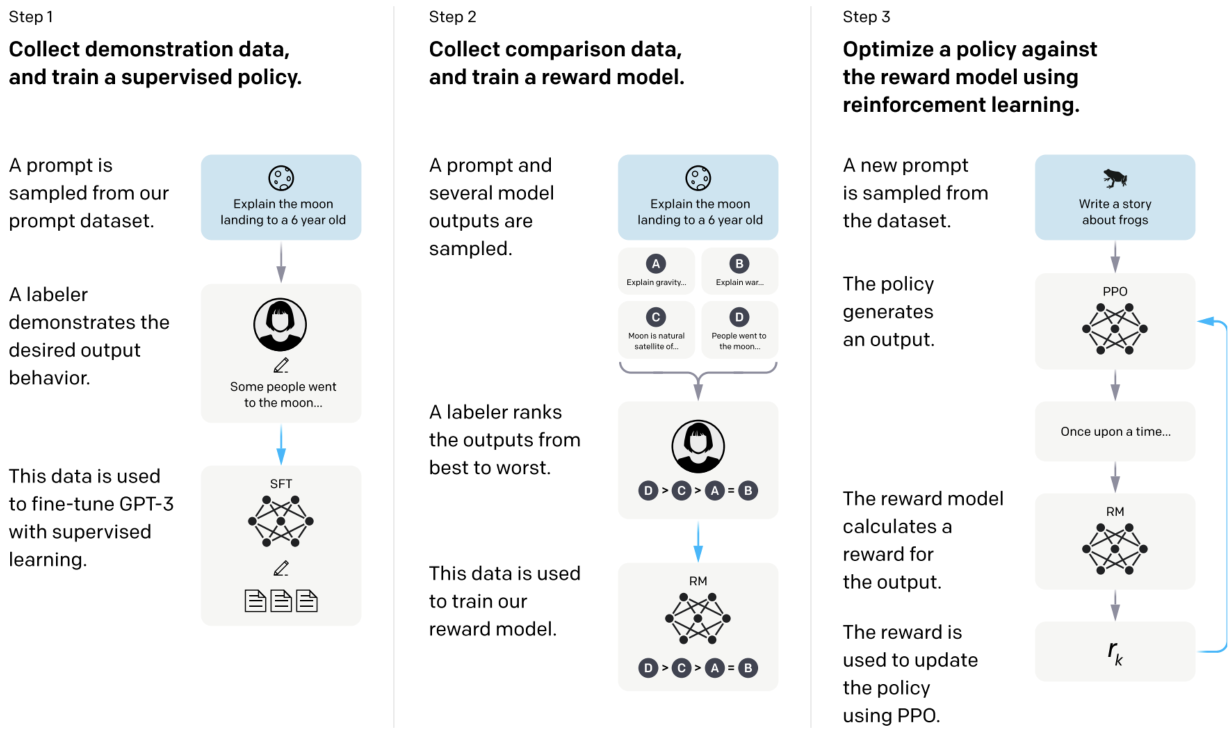

Figure 4.2: AgentSCA with RLHF: After fine-tuning the LLM, we run different tasks and evaluate the outcome of each task. Later, we train a reinforcement learning model with evaluated data. (25)

and collected traces in each point. Since Picoscope 6424E can collect signals up to 2.5 GHz frequency, we can run spectrum analysis for each point. Datasheet stated that the operating frequency of TPU is 500 MHz. Our spectrum analysis supports that statement shown in Figure 4.6 which shows a peak around 500 MHz. We filtered out our EM signals around 500 MHz to find the operating point. Figure 4.5 shows the location of the hot point. We conducted further experiments on that point.

4.4.2 Hyperparameter Extraction on NN

Neural network models are constructed by different hyperparameters such as the number of the hidden layer, the number of nodes in the hidden layer, and layer type. We produced several NN models with different numbers of layers and node sizes. After collecting traces for different models, we observed patterns for layers. After statically aligning traces by using those patterns, we ran TVLA tests on traces. The number of peaks in the results correlated with the number of layers inside the model.

The importance of an input feature gives a broad understanding of the model. We run a

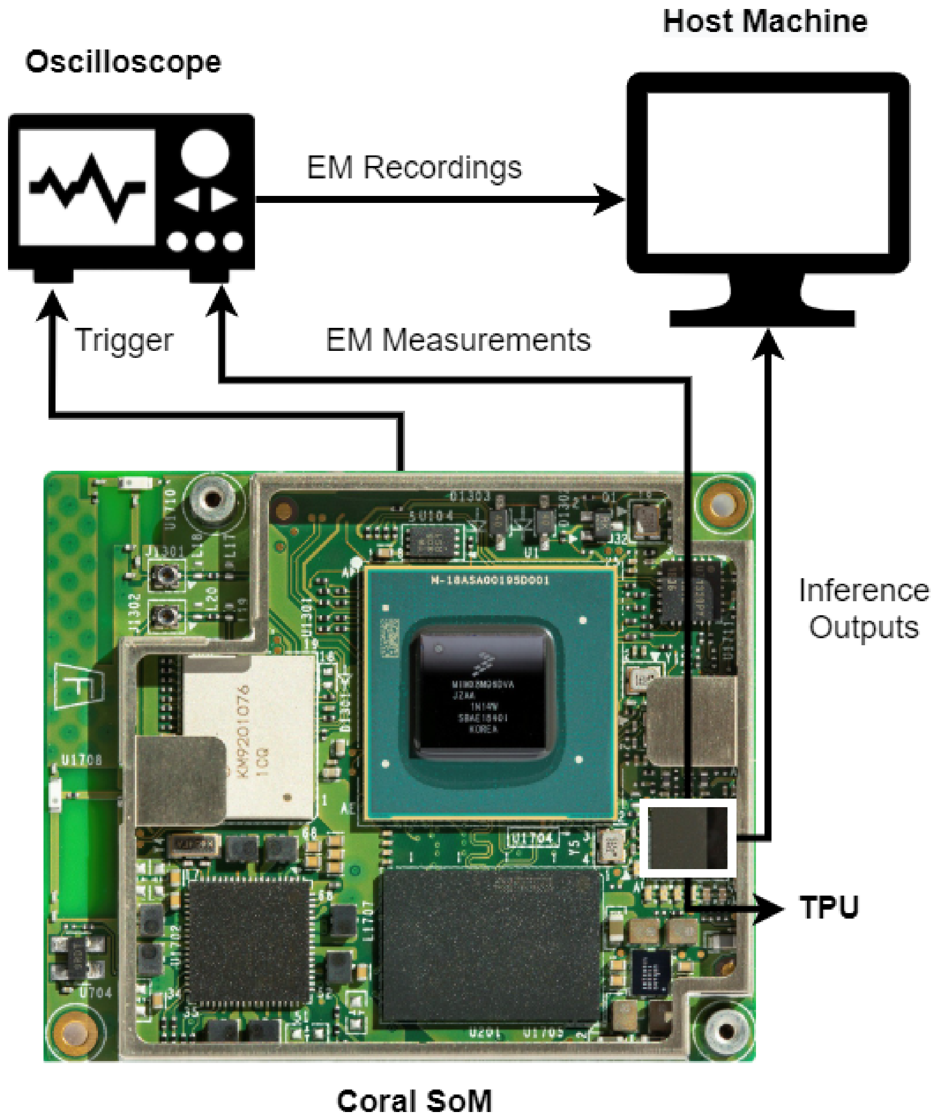

Figure 4.3: Experiment Setup: Coral SoM takes input from the host machine and starts inference and sends a trigger signal to the oscilloscope to collect EM measurements while the oscilloscope gets measurements from the EM probe and sends them to the host machine.

one-hot TVLA test for the input feature. One-hot TVLA is a method to find leakage for special input such that while a feature is set randomly, others are fixed to zero. In our experiments, we used the MNIST dataset (6) to train our models to classify handwritten digits. The MNIST dataset consists of 28×28 pixels, grayscale images. In our experiments, one pixel is assigned randomly while other pixels are set to zero. This experiment repeated for 1st, 100th, 200th, ... 784th pixels separately. We trained a model with 20 hidden layers that have 500 nodes for this experiment. Important pixels show leakages in deeper layers, while non-important pixels diminish their effects on early layers and don't show leakages in deeper layers. Therefore, these results help to interpret important features on different models.

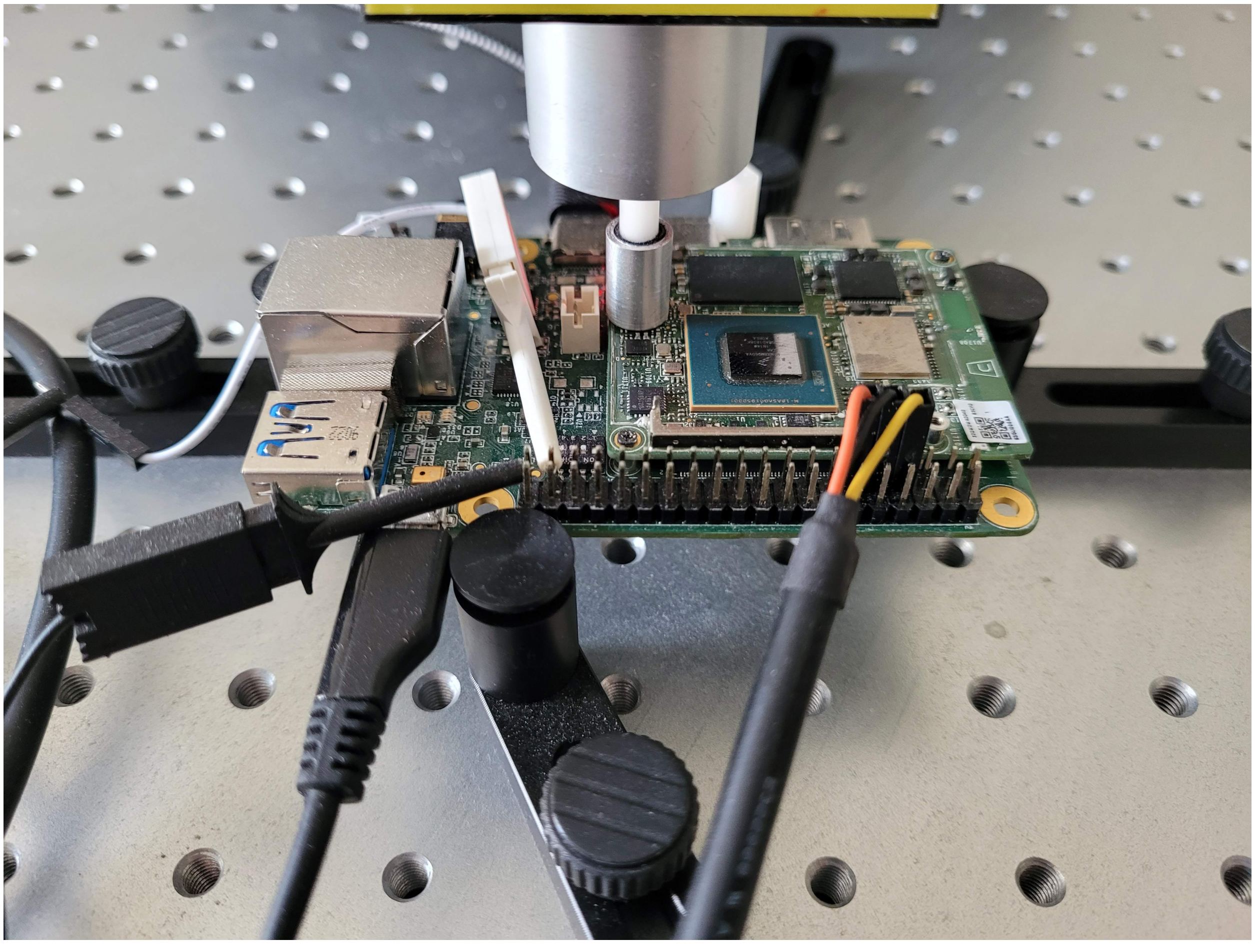

Figure 4.4: Experiment setup on a lab environment. It shows connections to run inference and the location of EM Probe to collect traces.

4.4.3 Parameter Extraction on NN

We extract hyperparameters by identifying patterns on traces. In order to recover parameters, we need to utilize CPA attacks. Knowing operation flow in hardware helps to understand the underlying architecture and attacking intermediate parameters. We have used the TensorflowLite library to understand underlying inference computation on quantized models. We targeted the first 8-bit multiplication operation in the quantized fully connected operation. Our EM/power model is Hamming weight which is a summation of set bits of the result of multiplication. In order to decrease noise produced by other operations, we train 1 fully connected layer consisting of 1 hidden node that has only 1 weight set while others set to zero and only 1 bias value is set. The trained NN architecture is shown in Figure 4.8.

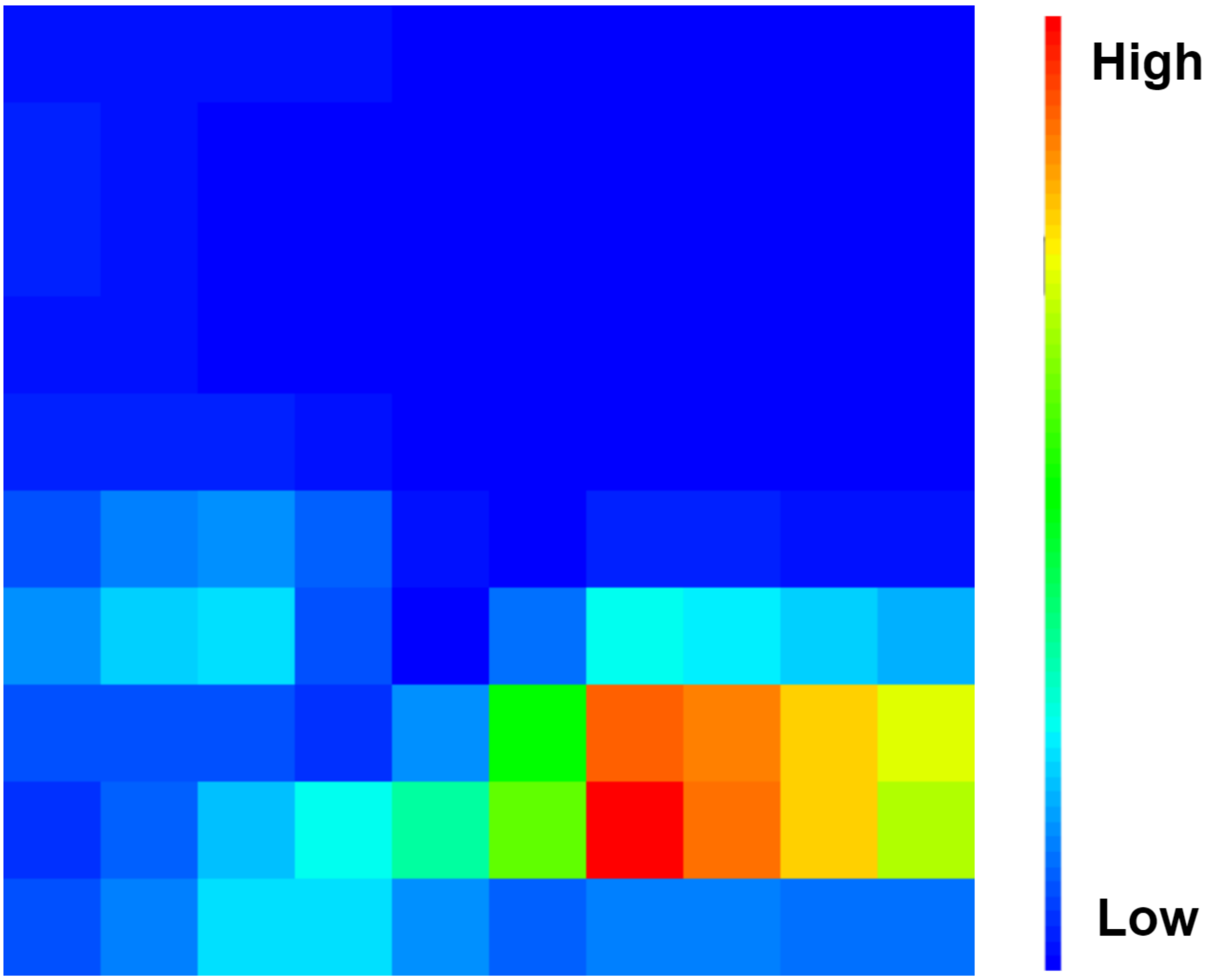

Figure 4.5: Heat map analysis where traces filtered out around 500 MHz. The red point shows high activation, which corresponds to location TPU computations.

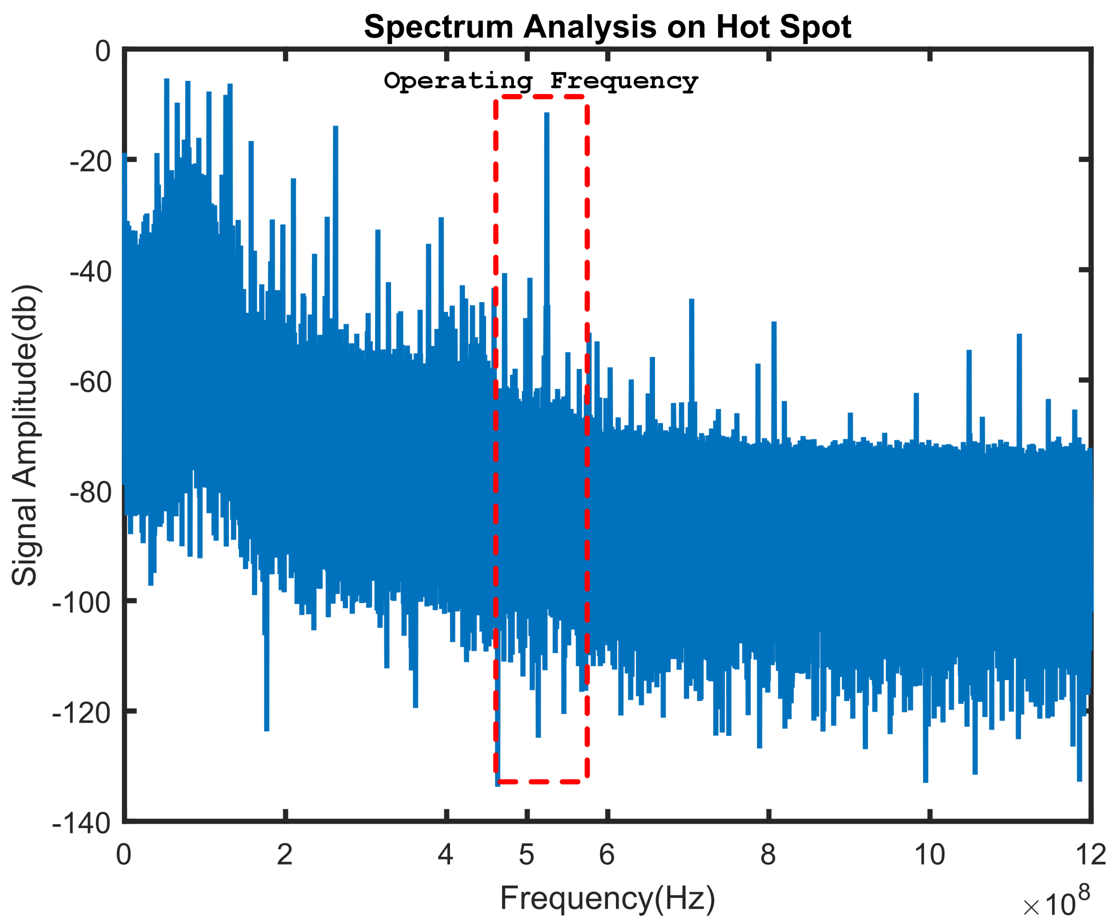

Figure 4.6: Spectrum Analysis in the range from 0 to 1.25 GHz of EM trace. The peak shown within the red square implies high activation around 500 MHz which can be a possible operating frequency.

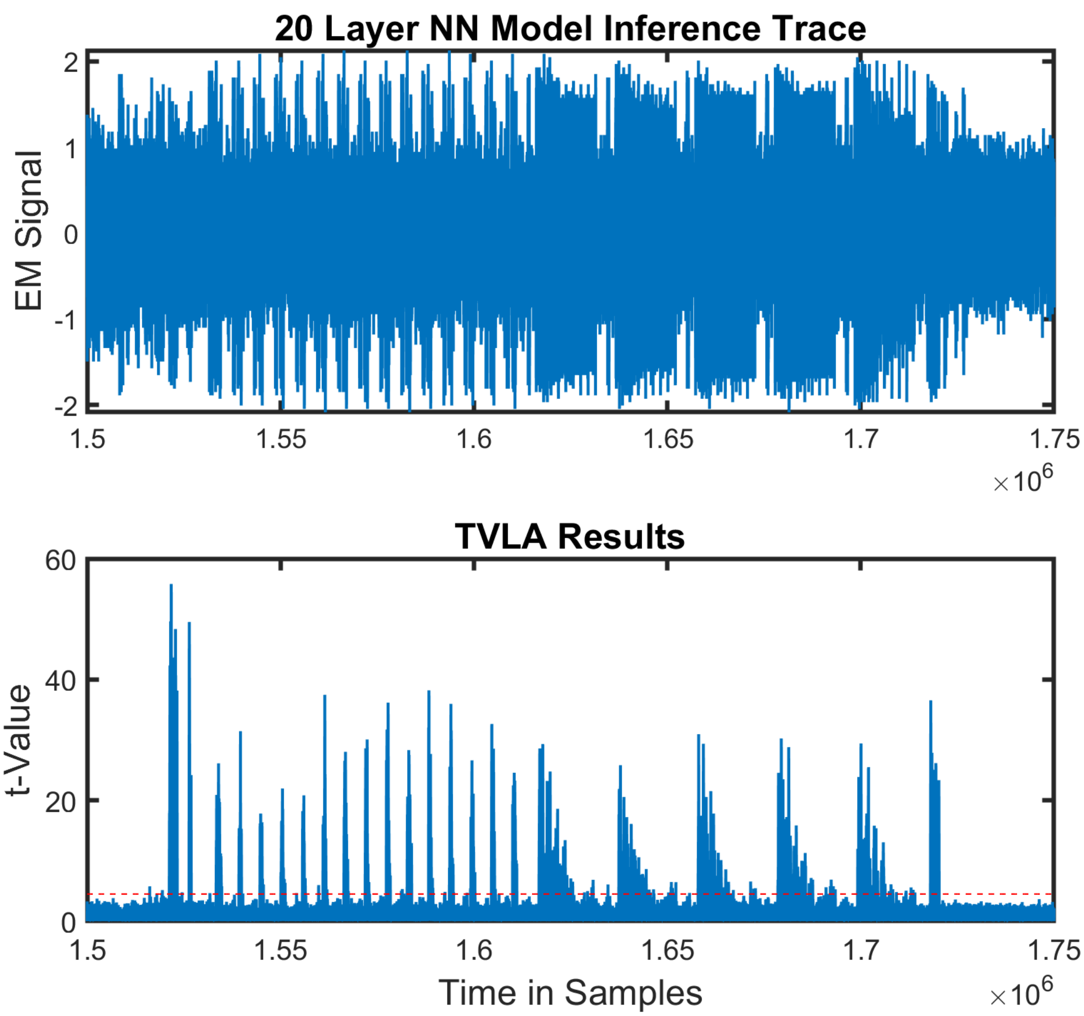

Figure 4.7: TVLA test results for 22 layers model inference EM Signal where red dotted line shows leakage threshold. The number of peaks in TVLA equals to the number of layers in the model.

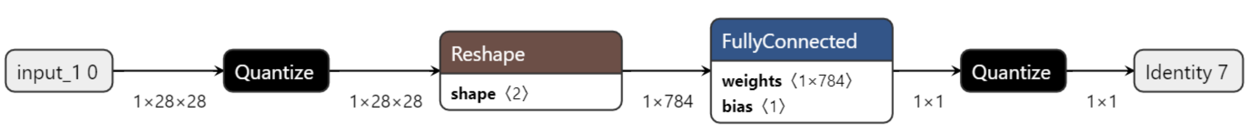

Figure 4.8: Hypothetical model used to recover intermediate parameters. This figure gives understanding of how Tensorflow Lite models are contructed after quantization.

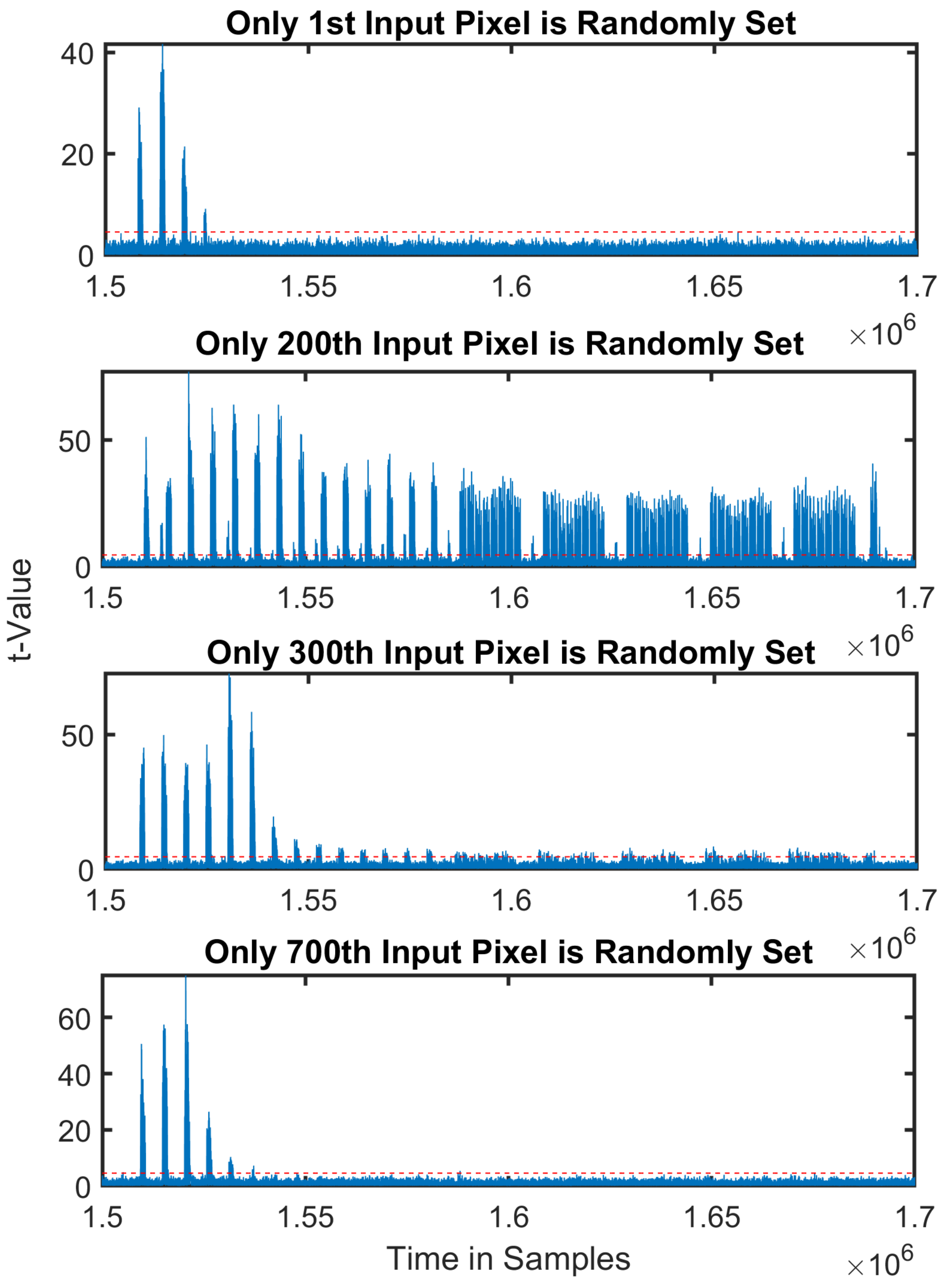

Figure 4.9: One-Hot TVLA test results when one fixed input pixel is randomly set while other pixels are zero on 20 layer 500 nodes NN model. Significant pixels (i.e., 200th) produce leakages on layer computation during inference.

CHAPTER 5

EVALUATION

To organize and categorize the tasks, we classify them into two main categories: *knowledge-based tasks* and *generative tasks*.

In knowledge-based tasks, the focus is on describing the circumstances and providing detailed information about the current state of the test system. These tasks require the agent to interpret the statistics and make accurate decisions or recommendations based on the given information. The goal is to assess the agent's ability to analyze and understand the side-channel analysis scenarios and provide correct actions or interpretations.

On the other hand, generative tasks involve providing code snippets or hypothetical power models based on the given architectural details of the system under analysis. These tasks test the agent's capability to generate relevant code or power models that align with the side-channel analysis scenario. By assessing the agent's ability to generate accurate and contextually appropriate solutions, we gain insights into its proficiency in applying side-channel analysis techniques to different architectural designs.

By categorizing the tasks into knowledge-based and generative tasks, we aim to evaluate different aspects of the side-channel analysis agent's capabilities. The knowledge-based tasks assess the agent's comprehension and decision-making skills, while the generative tasks focus on its ability to generate appropriate solutions based on the given architectural details. This categorization provides a comprehensive evaluation framework for gauging the performance and effectiveness of the agent in different aspects of side-channel analysis.

5.1 Knowledge-Based Task Results

Our results, as depicted in Table 5.1, demonstrate that reinforcement learning from human feedback techniques outperforms other approaches. However, it is worth noting that few-shot prompting techniques exhibit considerable effectiveness. Conversely, zero-shot learning faces limitations due to its limited knowledge about the specific topic. Additionally, the Fine-tuned LLM approach tends to become overtrained, resulting in similar outcomes for tasks under different conditions.

Table 5.1: Success rate table of techniques used in the AgentSCA

Techniques	Zero-Shot	Few-Shot	CoT	Fine-Tuned	RLHF
Success Rate(%)	50	90	60	80	100

The knowledge-based tasks are presented in a question-and-answer format to assess the performance of each phase of AgentSCA. Different questions are asked to each phase, and the corresponding answers are obtained. Table 5.2 displays the correctness of each task for the different phases of AgentSCA.

1. **Question:** What are the power models utilized in physical side-channel analysis?

   Short Answer: Hamming Weight, Hamming Distance, and Identity power models

2. **Question:** What is the initial analysis that needs to be conducted for a side-channel analysis?

   Short Answer: TVLA analysis.

3. **Question:** What is the threshold value used in the TVLA t-test to determine the leakage?

   Short Answer: 4.5 t-score.

4. **Question:** What is the recommended action when the SNR analysis has a low score and the TVLA t-Test has a high t-score?

   Short Answer: To collect more measurements.

5. **Question:** What is the recommended action when the correlation analysis has a low score and the TVLA t-Test has a high t-score?

   Short Answer: To change power model.

6. **Question:** What are the possible attack points in AES execution for a hypothetical power model?

   Short Answer: Round Key Addition and Sbox output in the first and last rounds.

7. **Question:** How is the 8-bit MAC (Multiply-Accumulate) unit operation implemented in a TPU?

   Short Answer: The 8-bit MAC unit operation in a TPU is implemented using systolic array structures to execute layers of multiplication.

8. **Question:** Which statistical tests are necessary to detect a point of interest in time samples during a physical side-channel analysis?

   Short Answer: The TVLA test and the SNR test.

9. **Question:** What type of power model is preferred for ASIC and FPGA implementations?

   Short Answer: For ASIC implementations, the Hamming Weight power model is preferred. For FPGA implementations, the Hamming Distance power model is preferred.

10. **Question:** How does the TPU make use of systolic arrays for designing its MAC units?

    Short Answer: By organizing the flow of data to enhance the throughput of the matrix multiplication operation.

5.2 AES Side-Channel Analysis Case

We utilize prompt engineering to generate code for a hypothetical power model in the first round key addition operation. The AgentSCA with RLHF technique returns the code, which is illustrated in Listing 4. By implementing side-channel analysis (SCA) using the generated code, we successfully recover the key with fewer than 2000 traces. Figure 5.1 shows a high correlation key guess outperforms the other candidates in confidence score.

Table 5.2: Successful response table for each task by using techniques used in AgentSCA

Task Number	Zero-Shot	Few-Shot	CoT	Fine-Tuned	RLHF
1	×	√	×	√	√
2	√	√	√	√	√
3	√	√	√	×	√
4	×	√	√	√	√
5	×	×	×	×	√
6	√	√	√	√	√
7	√	√	√	√	√
8	√	√	√	√	√
9	×	√	×	√	√
10	×	√	×	√	√

5.3 Machine Learning Accelerator Side-Channel Analysis Case

We prompt the AgentSCA with RLHF to produce attack code on TPU and obtained the correct code provided in Listing 5. After using the code in experiments, we find that the chip operates at 500 MHz in Fig. 4.6 by looking at the spectrum, and the operating point is located in the chip shown in Fig. 4.5. One inference produces the EM signal trace shown upper plot on Fig. 4.7 with the plot. Those patterns and peaks give information about the specification of the model. After collecting 3000 traces on 20 hidden layers with 1 input and 1 output layer model, we run TVLA tests which result in peaks ranging between 20 and 60 t-Value that is higher than the threshold leakage value 4.5. The number of peaks and patterns of the bottom plot on Fig. 4.7 is 22 which is equal to the number of layers in the model. We repeat that experiment for $1, 5, 10$ and 40 hidden layers and observ the same correlation between the number of peaks in the TVLA test and the number of layers in the NN model.

The second experiment tries to infer the importance of the features of the model. In order to do that, we run one-hot TVLA tests for different input configurations. Fig. 4.9 presents one-hot TVLA results of when only 1st, 200th, 300th, 700th input pixel is set from top to bottom respectively. Results show that leakages for 1st and 700th input pixel diminish for the deeper layers while 200th input pixel has more t-Value than the rest of the input pixels. 300th pixel contributes little to the computation of inference output. 1st, 300th, 700th pixels are located edge of the image while 200th pixel is in the center. That gives an understanding of why that pixel is important, which is useful for adversarial attacks on trained models.

The third experiment utilizes CPA to find a correlation between input, output, and intermediate values. We conduct experiments on 20 Hidden Layers with 500 nodes which produce

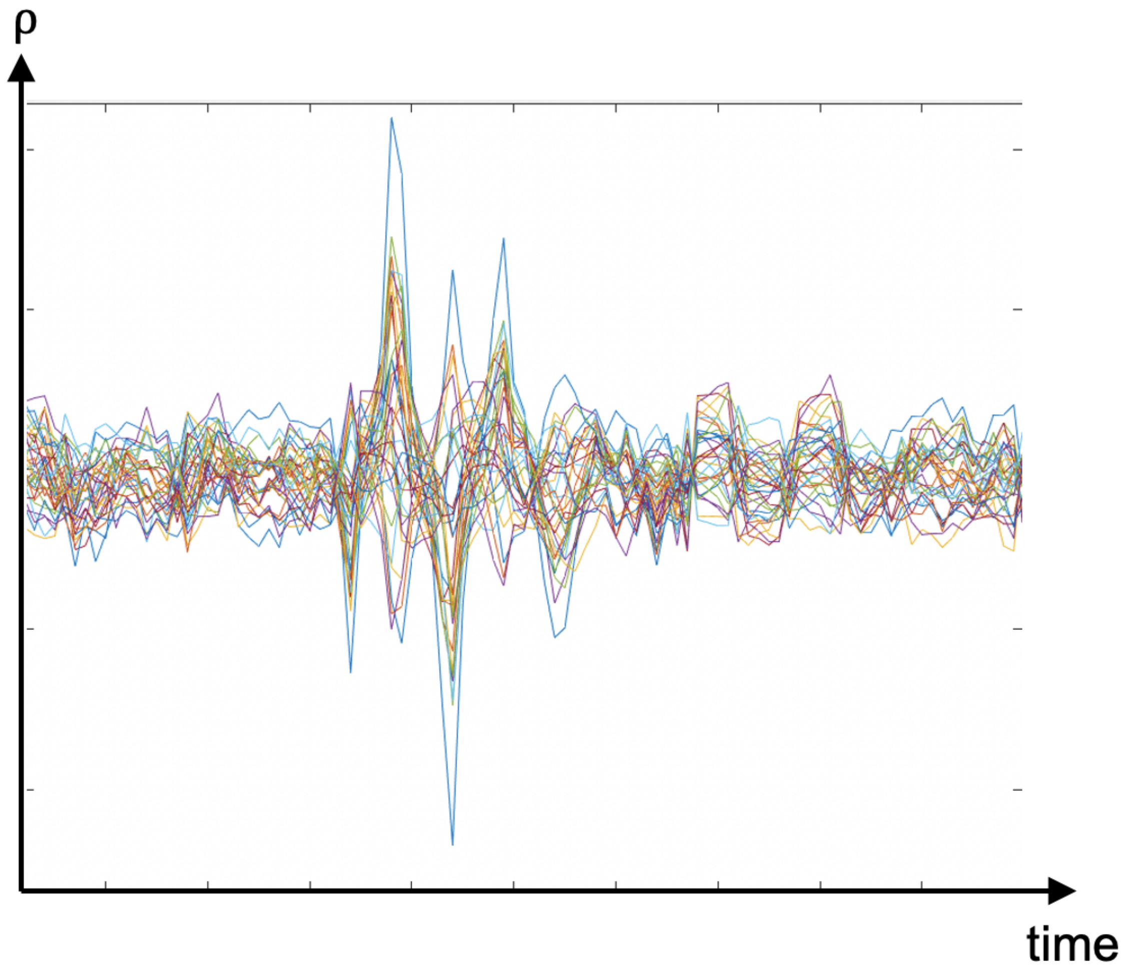

Figure 5.1: AES Key Addition CPA: Peak value means that correlation of Hamming Weight of AES round key addition attack code is succesful.

traces shown in the top plot of Fig. 5.2. We observe high Pearson Correlation values on the first layer of the corresponding trace shown in the middle plot of Fig. 5.2. The bottom plot of Fig. 5.2 shows the output correlation result which peaks on the last pattern of the trace. After getting successful results for those correlation tests, the next step is the investigation of intermediate secret value recovery. 8-bit multiplication of secret weight and known input is the target point for attack. After running the CPA attack for 255 possible weight keys, results in Fig. 5.3 show that 2 peaks correspond to possible computation points in time. Because of multiplication such a non-linear operation, CPA produces false-positive results for correct secret weight guess.

aes-keyadd-hypothtical-power.py

```
# Perform the round key addition for each byte and subkey guess, and compute
 ↪  the hypothetical power model
def perform_hypothetical_power_model(plaintexts):
    num_bytes = 16
    num_guesses = 256
    num_traces = len(plaintexts)

    hypothetical_power_model = np.zeros((num_bytes, num_guesses, num_traces))

    for byte in range(num_bytes):
        for guess in range(num_guesses):
            # Initialize an array to store the hypothetical power consumption
            hypothetical_power = np.zeros(num_traces)

            # Perform the round key addition for each trace
            for i in range(num_traces):
                plaintext_byte = plaintexts[i][byte]
                result_byte = round_key_addition(plaintext_byte, guess)

                # Calculate the hypothetical power consumption (e.g., Hamming
                 ↪  weight)
                hypothetical_power[i] = np.count_nonzero(result_byte)

            # Store the hypothetical power consumption in the power model
            hypothetical_power_model[byte][guess] = hypothetical_power

    return hypothetical_power_model
```

Listing 4: Round Key Addition Hypothetical Power Model Code

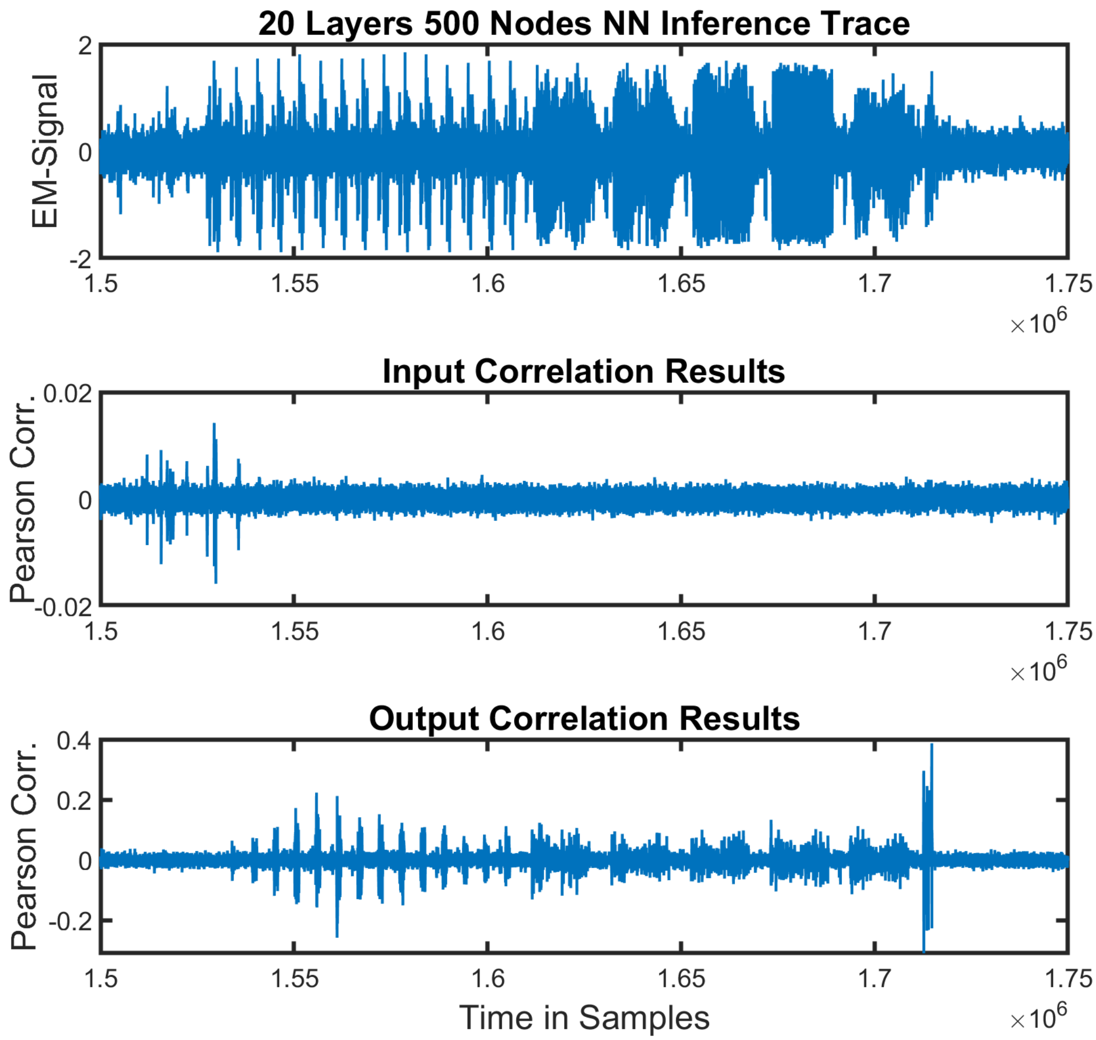

Figure 5.2: Input/Output correlation results for NN inference EM trace. The plot in the middle shows the correlation of input values with EM signals at the beginning of the inference trace and the bottom plot presents the output correlation at the end of the trace which corresponds output layer.

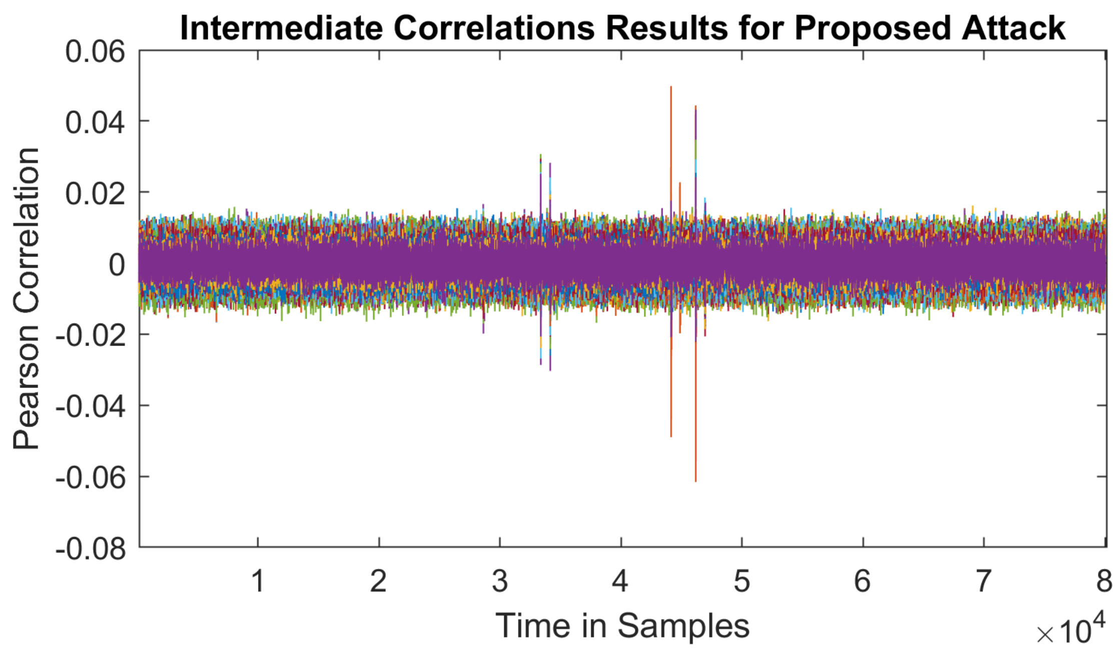

Figure 5.3: Intermediate Correlations for an 8-bit multiplication operation. Peaks of the plot show a significant correlation with false-positive key guesses.

mac-hypothetical-power.py

```
import numpy as np

# Given information
num_inputs = 1000
weight_bits = 8
activation_bits = 8
bias_bits = 32
max_weight_value = 255
trace_length = 100  # Adjust as per your actual trace length

# Assume input values for inferences
input_values = np.random.randint(0, max_weight_value + 1, size=(num_inputs,
 ↪  784), dtype=np.int8)

# Assume measured power traces for inferences
power_traces = np.random.randint(0, 100, size=(num_inputs, trace_length),
 ↪  dtype=np.int8)

def hamming_weight(x):
    return bin(x).count('1')

def calculate_power_model():
    hamming_weight_model = np.zeros((num_inputs, max_weight_value + 1))
    bias = np.random.randint(0, 2 ** bias_bits)

    for i in range(num_inputs):
        # Assuming activation and bias values (You need to extract them from
         ↪  input_values)
        activation = input_values[i, :activation_bits].view(np.uint8)

        # Calculating the Hamming weight of the guessed weight value *
         ↪  activation for each guess
        for guess_weight in range(max_weight_value + 1):
            hamming_weight_model[i][guess_weight] =
             ↪  hamming_weight(guess_weight * activation)

    return hamming_weight_model

hamming_weight_model = calculate_power_model()

# Now, you can use this hamming_weight_model to analyze the power consumption
 ↪  based on your side-channel attack.
# This model contains the Hamming weight values for each guess of weight *
 ↪  activation for all inputs.
# You can further analyze the power traces and match them with the Hamming
 ↪  weight model to infer the weight values.
```

Listing 5: Multiply-Accumulate Power Hypothesis Model Code

CHAPTER

6

DISCUSSION

Studies have shown that LLMs excel in knowledge-based tasks but struggle with generative tasks. In the case of Agent SCA, fine-tuning alone resulted in an overfitting of the model to specific conditions, making it ineffective in generating unique structures within desired limitations and prone to hallucinations. To address these limitations, the incorporation of reinforcement learning with human feedback was explored. Human guidance proved crucial in ensuring the efficiency and accuracy of the system. Prompt engineering initially showed promising results by enabling the steering of AgentSCA with few-shot prompting. However, in the long run, this approach was found to be inefficient as it required a single response from a vast range of possible solutions. In tasks such as TVLA, correlation, and SNR analysis, LLMs performed reasonably well. However, they had limitations in interpreting graphs and conducting timing analysis. The integration of LLMs with other techniques could potentially overcome these challenges and provide effective solutions.

The performance of the system in generative tasks, based on previous examples, was found to be unsatisfactory. Prompt engineering, specifically zero-shot learning, consistently yielded poor results. Attempting to push the system further often resulted in incorrect changes to the code. Although the Chain-of-Thought technique allows for step-by-step teaching, it was not as effective as few-shot learning. Few-shot learning proved to be highly successful and easily adaptable for earlier discussions. However, the fine-tuning approach led to overfitting issues. Providing expert feedback to the system proved to be instrumental in maximizing its performance.

Currently, the overall system is not capable of successfully attacking the EdgeTPU machine-learning accelerator using the developed agent. However, it has the potential to generate important hypothetical power models that are suitable for the architecture of Google's TPU. These models have not been fully functional yet but show promise for future improvements.

On the other hand, when it comes to generating AES attacks, the system can successfully generate and elaborate on the attack. This demonstrates the effectiveness of the agent in analyzing the side-channel information associated with AES encryption and leveraging it to perform successful attacks.

While the system's capabilities are still limited in certain aspects, such as attacking the EdgeTPU machine learning accelerator, the ability to generate successful AES attacks is a significant milestone. Further enhancements and refinements can be made to improve the system's overall performance and broaden its applicability to various side-channel analysis scenarios

CHAPTER

7

CONCLUSION

In conclusion, physical side-channel analysis poses a significant vulnerability in modern architecture and cryptographic systems. This research explores the potential of LLMs in addressing this weakness and demonstrates their ability to be utilized in this field. As the first work of its kind, this study successfully generates side-channel analysis code, including Correlation Power Analysis (CPA), and generates hypothetical power models for specific algorithms.

The findings of this research suggest that integrating LLM-based tools with real-time monitoring probes could greatly enhance the automation of side-channel analysis. By leveraging the decision-making capabilities of LLMs, the process of detecting and mitigating side-channel vulnerabilities could be streamlined and made more efficient.

It is important to note that, while LLMs show promise in knowledge-based tasks and certain aspects of side-channel analysis, their limitations in generative tasks and interpreting complex data still require human expertise and guidance. Further research is needed to refine and enhance the capabilities of LLMs in addressing these challenges.

Overall, this work highlights the potential of LLMs as a valuable tool for side-channel analysis, contributing to the ongoing efforts in strengthening the security of modern architectures and cryptographic structures. By combining the power of LLMs with real-time monitoring techniques, future systems can be equipped with more robust and automated side-channel analysis capabilities.

REFERENCES

[1] Agrawal, D., Archambeault, B., Rao, J. R., and Rohatgi, P. (2003). The em side—channel (s). In *Cryptographic Hardware and Embedded Systems-CHES 2002: 4th International Workshop Redwood Shores, CA, USA, August 13–15, 2002 Revised Papers 4*, pages 29–45. Springer.

[2] AI, C. (2018). System on module datasheet. `https://coral.ai/docs/som/datasheet/`.

[3] Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J. D., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., et al. (2020). Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901.

[4] Chen, Y., Xie, Y., Song, L., Chen, F., and Tang, T. (2020). A survey of accelerator architectures for deep neural networks. *Engineering*, 6(3):264–274.

[5] Christiano, P. F., Leike, J., Brown, T., Martic, M., Legg, S., and Amodei, D. (2017). Deep reinforcement learning from human preferences. *Advances in neural information processing systems*, 30.

[6] Deng, L. (2012). The mnist database of handwritten digit images for machine learning research. *IEEE Signal Processing Magazine*, 29(6):141–142.

[7] Gandolfi, K., Mourtel, C., and Olivier, F. (2001). Electromagnetic analysis: Concrete results. In *Cryptographic Hardware and Embedded Systems—CHES 2001: Third International Workshop Paris, France, May 14–16, 2001 Proceedings 3*, pages 251–261. Springer.

[8] Gao, L., Madaan, A., Zhou, S., Alon, U., Liu, P., Yang, Y., Callan, J., and Neubig, G. (2023). Pal: Program-aided language models. In *International Conference on Machine Learning*, pages 10764–10799. PMLR.

[9] Google (2021). Build beneficial and privacy preserving ai. `https://coral.ai/`.

[10] Hua, W., Zhang, Z., and Suh, G. E. (2018). Reverse engineering convolutional neural networks through side-channel information leaks. In *Proceedings of the 55th Annual Design Automation Conference*, pages 1–6.

[11] Intel (2015). Compute stick neural network accelarator. `https://www.intel.com/content/www/us/en/products/boards-kits/compute-stick.html`.

[12] Jouppi, N. P., Young, C., Patil, N., Patterson, D., Agrawal, G., Bajwa, R., Bates, S., Bhatia, S., Boden, N., Borchers, A., et al. (2017). In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th annual international symposium on computer architecture*, pages 1–12.

[13] Kaplan, J., McCandlish, S., Henighan, T., Brown, T. B., Chess, B., Child, R., Gray, S., Radford, A., Wu, J., and Amodei, D. (2020). Scaling laws for neural language models. *arXiv preprint arXiv:2001.08361*.

[14] Kim, Y. (2014). Convolutional neural networks for sentence classification. *arXiv preprint arXiv:1408.5882*.

[15] Kingma, D. P. and Welling, M. (2013). Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114*.

[16] Kocher, P., Jaffe, J., and Jun, B. (1999). Differential power analysis. *Advances in Cryptology-CRYPTO'99*, 1666:388–397.

[17] Kocher, P., Jaffe, J., Jun, B., and Rohatgi, P. (2011). Introduction to differential power analysis. *Journal of Cryptographic Engineering*, 1:5–27.

[18] Kocher, P. C. (1996). Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In *Advances in Cryptology—CRYPTO'96: 16th Annual International Cryptology Conference Santa Barbara, California, USA August 18–22, 1996 Proceedings 16*, pages 104–113. Springer.

[19] Kojima, T., Gu, S. S., Reid, M., Matsuo, Y., and Iwasawa, Y. (2022). Large language models are zero-shot reasoners. *Advances in neural information processing systems*, 35:22199–22213.

[20] Kotsiantis, S. B., Zaharakis, I., Pintelas, P., et al. (2007). Supervised machine learning: A review of classification techniques. *Emerging artificial intelligence applications in computer engineering*, 160(1):3–24.

[21] Mangard, S., Oswald, E., and Popp, T. (2008). *Power analysis attacks: Revealing the secrets of smart cards*, volume 31. Springer Science & Business Media.

[22] Min, S., Lyu, X., Holtzman, A., Artetxe, M., Lewis, M., Hajishirzi, H., and Zettlemoyer, L. (2022). Rethinking the role of demonstrations: What makes in-context learning work? *arXiv preprint arXiv:2202.12837*.

[23] Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., and Riedmiller, M. (2013). Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*.

[24] Nvidia (2019). Jetson nano neural network accelarator. `https://developer.nvidia.com/embedded/jetson-nano-developer-kit`.

[25] Ouyang, L., Wu, J., Jiang, X., Almeida, D., Wainwright, C., Mishkin, P., Zhang, C., Agarwal, S., Slama, K., Ray, A., et al. (2022). Training language models to follow instructions with human feedback. *Advances in Neural Information Processing Systems*, 35:27730–27744.

[26] Quisquater, J.-J. and Samyde, D. (2001). Electromagnetic analysis (ema): Measures and counter-measures for smart cards. In *Smart Card Programming and Security: International Conference on Research in Smart Cards, E-smart 2001 Cannes, France, September 19–21, 2001 Proceedings*, pages 200–210. Springer.

[27] Raffel, C., Shazeer, N., Roberts, A., Lee, K., Narang, S., Matena, M., Zhou, Y., Li, W., and Liu, P. J. (2020). Exploring the limits of transfer learning with a unified text-to-text transformer. *The Journal of Machine Learning Research*, 21(1):5485–5551.

[28] Rivain, M. and Prouff, E. (2010). Provably secure higher-order masking of aes. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 413–427. Springer.

[29] Simonyan, K. and Zisserman, A. (2014). Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*.

[30] Tillich, S. and Herbst, C. (2008). Attacking state-of-the-art software countermeasures—a case study for aes. In *Cryptographic Hardware and Embedded Systems–CHES 2008: 10th International Workshop, Washington, DC, USA, August 10-13, 2008. Proceedings 10*, pages 228–243. Springer.

[31] Touvron, H., Lavril, T., Izacard, G., Martinet, X., Lachaux, M.-A., Lacroix, T., Rozière, B., Goyal, N., Hambro, E., Azhar, F., et al. (2023). Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*.

[32] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., and Polosukhin, I. (2017). Attention is all you need. *Advances in neural information processing systems*, 30.

[33] Veyrat-Charvillon, N., Medwed, M., Kerckhof, S., and Standaert, F.-X. (2012). Shuffling against side-channel attacks: A comprehensive study with cautionary note. In *Advances in Cryptology–ASIACRYPT 2012: 18th International Conference on the Theory and Application of Cryptology and Information Security, Beijing, China, December 2-6, 2012. Proceedings 18*, pages 740–757. Springer.

[34] Wei, J., Bosma, M., Zhao, V. Y., Guu, K., Yu, A. W., Lester, B., Du, N., Dai, A. M., and Le, Q. V. (2021). Finetuned language models are zero-shot learners. *arXiv preprint arXiv:2109.01652*.

[35] Wei, J., Wang, X., Schuurmans, D., Bosma, M., Xia, F., Chi, E., Le, Q. V., Zhou, D., et al. (2022). Chain-of-thought prompting elicits reasoning in large language models. *Advances in Neural Information Processing Systems*, 35:24824–24837.

[Woo et al.] Woo, D. H., Ravi NarayanaswamiSorace, R. E., Reinhardt, V. S., and Chan, C. Phase array calibration orthogonal phase sequence.

[37] Yu, H., Ma, H., Yang, K., Zhao, Y., and Jin, Y. (2020). Deepem: Deep neural networks model recovery through em side-channel information leakage. In *2020 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 209–218. IEEE.

APPENDICES

APPENDIX

A

ACRONYMS

A summary of all acronyms is documented in Table A.1.

Table A.1: A summary of acronyms used in alphabetical order.

Acronym	Abbreviation
Advanced Encryption System	AES
Correlational Power Analysis	CPA
Deep Learning	DL
Differential Power Analysis	DPA
Electromagnetic	EM
Field Programmable Gate Arrays	FPGA
Graphic Processor Unit	GPU
Hamming Weight	HW
Hamming Distance	HD
Large Language Model	LLM
Machine Learning	ML
Multiply-Accumulate	MAC
Neural Network	NN
Neural Processing Unit	NPU

Rivest–Shamir–Adleman Public Key Encryption	RSA
Side-Channel Analysis	SCA
Signal-to-Noise Ratio	SNR
Tensor Processor Unit	TPU
Test Vector Leakage Assessment	TVLA

APPENDIX

B

COMPLETE CONVERSATIONS WITH AGENTSCA

B.1 Expert - Agent Zero-Shot Conversation

Expert: My objective is to conduct side-channel analysis on the AES cryptographic algorithm, specifically focusing on the round key addition in the first round using the hamming weight power model. I possess a dataset consisting of 1000 power measurements of AES execution, each containing 100 sample points, with a corresponding plaintext for each power measurement. Notably, the correct key is unknown. Can you write a correlational power analysis code for this task?

Agent: Sure, I can provide you with a Python code template for conducting correlational power analysis on the AES cryptographic algorithm using the Hamming weight power model. Please note that side-channel analysis is a complex and sensitive task, and this code is a simplified example for educational purposes only. In real-world scenarios, side-channel analysis requires careful handling of data, noise, and other considerations.

```
import numpy as np
import matplotlib.pyplot as plt

# Define the AES S-box
AES_SBOX = AES.Sbox_table

def hamming_weight(x):
    return bin(x).count('1')

def intermediate_value(pt, key_guess):
    return AES_SBOX[pt ^ key_guess]

def power_model(hamming_weight_val):
    # Replace this function with your power model implementation.
    # It should return the predicted power consumption based on the hamming
     ↪  weight.
    # In practice, this involves using machine learning or other statistical
     ↪  methods.
    # For this example, let's assume a linear model for simplicity.
    return 2.0 * hamming_weight_val + np.random.normal(0, 1)

def calculate_correlation(power_traces, hamming_weights):
    num_traces = len(power_traces)
    num_samples = len(power_traces[0])
    correlations = np.zeros(num_samples)

    for sample_idx in range(num_samples):
        sample_sum = 0.0
        weight_sum = 0.0
        weight_sq_sum = 0.0
        power_sum = 0.0
        weight_power_sum = 0.0

        for trace_idx in range(num_traces):
            power_sample = power_traces[trace_idx][sample_idx]
            hamming_weight_val = hamming_weights[trace_idx]

            sample_sum += 1
            weight_sum += hamming_weight_val
            weight_sq_sum += hamming_weight_val ** 2
            power_sum += power_sample
            weight_power_sum += hamming_weight_val * power_sample
```

```
        numerator = num_traces * weight_power_sum - weight_sum * power_sum
        denominator = np.sqrt((num_traces * weight_sq_sum - weight_sum ** 2) *
         ↪  (num_traces * np.var(power_traces[:, sample_idx])))

        correlations[sample_idx] = numerator / denominator

    return correlations

def main():
    # Load your power traces and plaintext data from your dataset here.
    # Replace the following arrays with your actual data.
    num_traces = 1000
    num_samples = 100

    power_traces = np.random.normal(0, 1, size=(num_traces, num_samples))  #
     ↪  Replace with your actual power traces.
    plaintexts = np.random.randint(0, 256, size=num_traces)  # Replace with
     ↪  your actual plaintexts.

    # Generate hamming weights for intermediate values in the first round.
    hamming_weights = np.array([hamming_weight(intermediate_value(pt,
     ↪  key_guess=0)) for pt in plaintexts])

    # Calculate the correlations.
    correlations = calculate_correlation(power_traces, hamming_weights)

    # Plot the correlations.
    plt.plot(correlations)
    plt.xlabel("Sample Points")
    plt.ylabel("Correlation")
    plt.title("Correlational Power Analysis for AES First Round Key Addition")
    plt.show()

if __name__ == "__main__":
    main()
```

Please note that the $power\_model$ function in this example is a simple placeholder. In real-world scenarios, you would need to implement an accurate power model, often involving more sophisticated statistical techniques or machine learning methods, depending on the characteristics of your power traces and setup. Also, keep in mind that 1000 power traces might be insufficient for conducting effective side-channel analysis. In practice, you would typically require a larger number of traces to achieve meaningful results and mitigate noise.

B.2 Expert - Agent Few-Shot Conversation

Expert: Here is an example of side-channel analysis on the AES cryptographic algorithm, specifically focusing on the round key addition in the first round using the hamming weight power model. I possess a dataset consisting of 1000 power measurements of AES execution, each containing 100 sample points, with a corresponding plaintext for each power measurement. Notably, the correct key is unknown. You can find the code below.

```
import numpy as np
import matplotlib as plt

# Load your power traces and plaintext data from your dataset here.
# Replace the following arrays with your actual data.
num_traces = 1000
num_samples = 100
aes_size = 16

def hamming_weight(n):
    return bin(n).count("1")

def intermediate_value(pt, key_guess):
    return pt ^ key_guess

def generate_power_model(plaintexts):
    hypothetical_power_model = np.zeros(size=(aes_size,num_traces,256))
    for byte_num in range(0, 16):
        for key_guess in range(0, 256):
            for trace_num in range(0, num_traces):
                hypothetical_power_model[byte_num][key_guess][trace_num] =
                 ↪ hamming_weight(intermediate_value(plaintexts[trace_num][byte_num],
                 ↪ key_guess))

    return hypothetical_power_model

def calculate_correlation(power_traces, hypothtical_power_model):

    bestguess = [0]*16
```

```
    cpaoutput = np.zeros(size=(aes_size,256))
    maxcpa = np.zeros(size=(aes_size,256))

    #Mean of all points in trace
    mean_traces = np.mean(power_traces, axis=0, dtype=np.float64)

    for byte_num in range(0, 16):

        for key_guess in range(0, 256):
            hypothesis = hypothtical_power_model[byte_num][key_guess]

            #Initialize arrays &amp; variables to zero
            sumnum = np.zeros(num_samples)
            sumden1 = np.zeros(num_samples)
            sumden2 = np.zeros(num_samples)

            #Mean of hypothesis
            mean_hypothesis = np.mean(hypothesis, dtype=np.float64)

            #For each trace, do the following
            for trace_num in range(0, num_traces):
                hdiff = (hypothesis[trace_num] - mean_hypothesis)
                tdiff = power_traces[trace_num,:] - mean_traces

                sumnum = sumnum + (hdiff*tdiff)
                sumden1 = sumden1 + hdiff*hdiff
                sumden2 = sumden2 + tdiff*tdiff

            cpaoutput[byte_num][key_guess] = sumnum / np.sqrt( sumden1 *
             ↪  sumden2 )
            maxcpa[byte_num][key_guess] = max(abs(cpaoutput[key_guess]))

        bestguess[byte_num] = np.argmax(maxcpa)
        return bestguess, cpaoutput

def main():

    power_traces = np.random.normal(0, 1, size=(num_traces, num_samples))  #
     ↪  Replace with your actual power traces.
    plaintexts = np.random.randbytes(size=(num_traces, aes_size))  # Replace
     ↪  with your actual plaintexts.
```

```
    # Generate hamming weight power model for intermediate values in the first
    ↪ round.
    hypothetical_power_model = generate_power_model(plaintexts)

    # Calculate the correlations.
    best_guess, correlations = calculate_correlation(power_traces,
    ↪ hypothetical_power_model)

    # Plot the correlations.
    plt.plot(correlations)
    plt.xlabel("Sample Points")
    plt.ylabel("Correlation")
    plt.title("Correlational Power Analysis for AES First Round Key Addition")
    plt.show()

if __name__ == "__main__":
    main()
```

Agent: The code you provided implements a basic Correlation Power Analysis (CPA) for side-channel analysis on the AES cryptographic algorithm. Specifically, it focuses on the first round key addition using the Hamming weight power model. CPA is a well-known side-channel attack technique used to extract secret information from cryptographic implementations based on their power consumption patterns.

However, I noticed a few issues with the code: Numpy array creation: The $np.zeros()$ function should be used instead of $np.random.bytes()$ for creating the $plaintexts$ array:

```
plaintexts = np.random.bytes(aes_size * num_traces)
plaintexts = np.frombuffer(plaintexts, dtype=np.uint8).reshape(num_traces,
↪ aes_size)
```

Expert: Can you write a correlational power analysis code for side-channel analysis on the AES cryptographic algorithm, specifically focusing on the round key addition in the first round using the hamming weight power model?

Agent: Sure! I can help you with a basic Correlation Power Analysis (CPA) code for conducting side-channel analysis on the AES cryptographic algorithm, specifically focusing on the round key addition in the first round using the hamming weight power model. The objective of this code will be to extract the first-round key using the power consumption patterns.

Before we proceed, let me clarify that side-channel attacks can be complex and sensitive to various factors. The code provided here is a simplified version, and real-world side-channel analysis requires careful considerations for noise reduction, signal alignment, and more sophisticated statistical techniques.

Let's get started with the code:

```
import numpy as np
import matplotlib.pyplot as plt

# AES parameters
num_traces = 1000
num_samples = 100
aes_size = 16

# AES S-box
sbox = np.array([
    0x63, 0x7C, 0x77, 0x7B, 0xF2, 0x6B, 0x6F, 0xC5,
    0x30, 0x01, 0x67, 0x2B, 0xFE, 0xD7, 0xAB, 0x76,
    # Complete the S-box values
    # ...
], dtype=np.uint8)

# Hamming weight calculation
def hamming_weight(n):
    return bin(n).count("1")

# Intermediate value calculation
def intermediate_value(pt, key_guess):
    return pt ^ key_guess

# Generate hypothetical power model for intermediate values in the first round
def generate_power_model(plaintexts):
    hypothetical_power_model = np.zeros(shape=(aes_size, num_traces, 256))
    for byte_num in range(aes_size):
        for key_guess in range(256):
            for trace_num in range(num_traces):
                hypothetical_power_model[byte_num, trace_num, key_guess] =
                 ↪  hamming_weight(
                    intermediate_value(plaintexts[trace_num, byte_num],
                     ↪  key_guess)
                )
    return hypothetical_power_model

# Calculate correlation between power traces and hypothetical power model
```

```
def calculate_correlation(power_traces, hypothetical_power_model):
    correlations = np.zeros(shape=(aes_size, 256))
    for byte_num in range(aes_size):
        for key_guess in range(256):
            for trace_num in range(num_traces):
                hypothesis = hypothetical_power_model[byte_num, trace_num,
                 ↪  key_guess]
                mean_hypothesis = np.mean(hypothesis, dtype=np.float64)
                mean_trace = np.mean(power_traces[trace_num],
                 ↪  dtype=np.float64)
                numerator = np.sum((hypothesis - mean_hypothesis) *
                 ↪  (power_traces[trace_num] - mean_trace))
                denominator1 = np.sum((hypothesis - mean_hypothesis) ** 2)
                denominator2 = np.sum((power_traces[trace_num] - mean_trace)
                 ↪  ** 2)
                correlations[byte_num, key_guess] = numerator /
                 ↪  np.sqrt(denominator1 * denominator2)
    return correlations

# Extract the best key guess for each byte of the key
def extract_key(correlations):
    key = []
    for byte_num in range(aes_size):
        best_guess = np.argmax(np.abs(correlations[byte_num]))
        key.append(best_guess)
    return bytes(key)

def main():
    # Replace with your actual power traces and plaintexts
    power_traces = np.random.normal(0, 1, size=(num_traces, num_samples))
    plaintexts = np.random.bytes(aes_size * num_traces)
    plaintexts = np.frombuffer(plaintexts, dtype=np.uint8).reshape(num_traces,
     ↪  aes_size)

    # Generate hypothetical power model for intermediate values in the first
     ↪  round
    hypothetical_power_model = generate_power_model(plaintexts)

    # Calculate the correlations
    correlations = calculate_correlation(power_traces,
     ↪  hypothetical_power_model)
```

```
    # Extract the first-round key
    extracted_key = extract_key(correlations)
    print("Extracted Key:", extracted_key.hex().upper())

    # Plot the correlations (Optional - for visualization)
    plt.plot(correlations.T)
    plt.xlabel("Key Guess")
    plt.ylabel("Correlation")
    plt.title("Correlational Power Analysis for AES First Round Key Addition")
    plt.show()

if __name__ == "__main__":
    main()
```

Please make sure to replace the placeholder power traces and plaintexts with your actual data before running the code. Additionally, complete the AES S-box values in the Sbox array. Keep in mind that this is a basic implementation of CPA, and you may need to perform additional preprocessing and optimizations depending on the specifics of your dataset and the target device's characteristics. Noise reduction techniques, proper alignment, and advanced statistical methods can significantly improve the results of side-channel analysis in real-world scenarios.

B.3 Expert - Agent Chain of Thought Conversation

Expert: My objective is to conduct side-channel analysis on the AES cryptographic algorithm, specifically focusing on the round key addition in the first round using the hamming weight power model. I possess a dataset consisting of 1000 power measurements of AES execution, each containing 100 sample points, with a corresponding plaintext for each power measurement. Notably, the correct key is unknown. Let's think step by step,

- Select a Hypothesis for the First Round Key: Choose a hypothetical value for one or more bytes of the first round key. For example, you might assume a particular byte is '0x00' as a starting hypothesis. It can go up to '0xff'.
- Calculate Hamming Weight of Intermediate Values: For the chosen byte(s) in the first round key hypothesis, calculate the Hamming weight (number of '1' bits) of the round key addition operation with plaintext(simply XORing the key guess with plaintext byte).

- Create Traces and Plaintexts: For each power measurement, create a trace containing the power consumption samples with the number of samples.
- Create a Hypothesis Table: Calculate hamming weight values of intermediate values of each key guess with each corresponding plaintext.
- Compute Correlation: For each sample point in a trace, compute the correlation coefficient between the power consumption and the Hamming weight power model value for the corresponding intermediate value from the hypothesis table.

Can you provide code for each step of this correlational power analysis?

Agent: Sure, I can help you with the code for each step of correlational power analysis for AES using the Hamming weight power model. For simplicity, I'll provide Python code snippets for each step. Make sure you have the necessary libraries installed, such as numpy, scipy, and matplotlib, to run the code smoothly.

```
import numpy as np
from scipy.stats import pearsonr

def calculate_hamming_weight(value):
    return bin(value).count('1')

def compute_correlation(trace, hypothesis_table):
    correlation_coefficients = []
    for byte_guess in range(256):
        hamming_weight_hypothesis = hypothesis_table[byte_guess]
        correlation, _ = pearsonr(trace, hamming_weight_hypothesis)
        correlation_coefficients.append(correlation)
    return correlation_coefficients

# Assuming you have loaded the dataset into these lists
traces = [...]  # A list of 1000 traces, each containing 100 sample points
plaintexts = [...]  # A list of 1000 corresponding plaintexts

# Create a Hypothesis Table for all 16 bytes of the key
hypothesis_tables = [{} for _ in range(16)]

# Loop over each trace and corresponding plaintext
for trace, plaintext in zip(traces, plaintexts):
    for key_byte_idx in range(16):
```

```
        byte_guess = 0x00  # Assuming you start the hypothesis with 0x00 for
         ↪  each byte of the key
        intermediate_value = byte_guess ^ plaintext[key_byte_idx]
        hamming_weight_hypothesis =
         ↪  calculate_hamming_weight(intermediate_value)
        hypothesis_tables[key_byte_idx][byte_guess] =
         ↪  hamming_weight_hypothesis

# Compute correlation for each trace and byte of the key hypothesis
correlations = [[] for _ in range(16)]
for trace, plaintext in zip(traces, plaintexts):
    for key_byte_idx in range(16):
        correlation = compute_correlation(trace,
         ↪  hypothesis_tables[key_byte_idx])
        correlations[key_byte_idx].append(correlation)
```

This code will create separate *hypothesis_tables* for each byte of the AES key and then calculate correlation coefficients for each byte individually. The correlations list will contain 16 sub-lists, each corresponding to the correlations for each byte hypothesis. Remember that this is a basic example, and you may need to fine-tune the analysis and apply more advanced techniques to handle practical side-channel analysis on AES effectively.