TSINGHUA SCIENCE AND TECHNOLOGY
ISSN 1007-0214 13/39 pp867–879
DOI: 10.26599/TST.2024.9010250
Volume 31, Number 2, April 2026

Research Article

A Machine-Learning Guided Tuning Approach for Trusted Application in Mobile Edge Computing

Yuhua Wang†, Xiaoao Zhu†, Jie Ren*, Feng Tian, Jie Zheng, Rui Cao, and Zheng Wang

Abstract: ARM TrustZone has become a cornerstone of security in mobile edge devices, but its protective measures often come at the expense of energy efficiency and system performance. Existing optimization methods rely heavily on cloud-based deep learning models and real-time measurements, making them vulnerable to fluctuating workloads and variable network conditions. This paper introduces a novel offline tuning framework for optimizing Trusted Applications (TAs) within the Open Portable Trusted Execution Environment (OP-TEE), an open-source TEE built on ARM TrustZone. Unlike traditional approaches, our method dispenses with on-device measurements and repeated TA compilations. Instead, it uses a predictive model trained on runtime characteristics observed in the Rich Execution Environment (REE) to estimate energy and performance metrics. Guided by these predictions, the framework then applies targeted optimizations through an automated tuning mechanism. Experimental results show that this offline approach accelerates the tuning process by 196× compared to the default method, while delivering a 27.23% performance improvement and a 25% reduction in energy consumption within just 170 s. These gains underscore the practicality and effectiveness of the proposed framework, paving the way for more efficient and adaptive TA optimization.

Key words: Open Portable Trusted Execution Environment (OP-TEE); Trusted Application (TA); program auto-tuning; predictive modelling

1 Introduction

Mobile Edge Computing (MEC) is witnessing rapid growth, utilizing edge devices located near end-users to perform data processing. By offloading tasks to nearby devices, MEC reduces latency, enhances responsiveness, and improves user experience[1]. However, despite these benefits, MEC also introduces several critical security challenges. These challenges arise from various factors, including the distribution of processing across multiple devices, limited control over the edge infrastructure, and vulnerabilities across both software and hardware components. Key issues include insufficient data protection, the risk of data corruption, and exposure to cyberattacks targeting edge devices. To address these risks, Trusted Execution Environments (TEEs), such as ARM TrustZone[2], Intel Software Guard Extensions (SGX)[3], and Google's Trusty TEE[4], provide hardware-based isolation mechanisms to secure sensitive data during computational operations. Among these, ARM TrustZone is particularly notable for its widespread adoption in mobile edge devices, offering robust

• Yuhua Wang, Xiaoao Zhu, Jie Ren, and Feng Tian are with School of Artificial Intelligence and Computer Science, Shaanxi Normal University, Xi'an 710119, China. E-mail: yh_wang@snnu.edu.cn; zhuxiaoao@snnu.edu.cn; renjie@snnu.edu.cn; tianfeng@snnu.edu.cn.
• Jie Zheng and Rui Cao are with School of Information Science and Technology, Northwest University, Xi'an 710127, China. E-mail: jzheng@nwu.edu.cn; cr@nwu.edu.cn.
• Zheng Wang is with School of Computing, University of Leeds, Leeds LS2 9JT, UK. E-mail: zwang5@leeds.ac.uk.
† Yuhua Wang and Xiaoao Zhu contribute equally to this paper.
* To whom correspondence should be addressed.
Manuscript received: 2024-08-22; revised: 2024-10-28; accepted: 2024-12-13

© The author(s) 2026. The articles published in this open access journal are distributed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/).

security features and a well-developed ecosystem. The versatility of ARM TrustZone has enabled its application across various domains, including blockchain[5] and deep learning[6, 7]. These applications leverage TrustZone's ability to isolate critical operations within secure environments, ensuring they are protected from tampering or unauthorized access, even on resource-constrained mobile platforms. As MEC continues to evolve, optimizing the performance of Trusted Applications (TAs) within TEEs, like ARM TrustZone, remains a significant challenge, requiring innovative solutions to balance security and operational efficiency.

ARM TrustZone enhances the security of mobile devices by enabling various operations, including the creation of isolated secure environments, context switching, and the execution of cryptographic functions. However, the added security comes at the cost of increased energy consumption and potential performance degradation. Previous studies[2, 8] have investigated the performance of the Open Portable Trusted Execution Environment (OP-TEE)[9], an open-source TEE based on ARM TrustZone, highlighting the significant energy demands associated with secure session management and program execution. Additionally, StreamBoxTZ[10] has introduced data flow analysis for data-intensive parallel computing tasks within ARM TrustZone, reducing computational overhead. Previous work Energy-efficient Task Scheduling strategy in Trusted Execution Environment (ETS-TEE)[11] further addresses these challenges through task scheduling mechanisms that offload computations to edge servers, lowering energy consumption on mobile devices. Despite these advancements, achieving consistent energy efficiency and performance improvements for TAs within the context of MEC remains challenging. The variability of network conditions and the diverse nature of workloads contribute to the complexity, demanding more adaptive and scalable solutions.

This paper presents a novel approach for optimizing TAs in the context of MEC. At the core of this approach is a predictive model that analyzes the runtime characteristics of source programs within the Rich Execution Environment (REE) to forecast both the energy consumption and performance of their corresponding TAs in the TEE. Leveraging the insights from the predictive model, the framework performs offline auto-tuning, enabling iterative optimization without the need for real-time compilation, deployment, or testing. This approach significantly reduces the overhead and costs associated with tuning TAs, streamlining the optimization process. The key contributions of this work are as follows:

- We investigate the transformation of C programs into TAs and introduce an automated mechanism for generating TAs tailored for the TEE (Section 4);
- We develop a comprehensive set of benchmarks specifically designed for evaluating performance and energy consumption in OP-TEE on edge devices, and perform an in-depth analysis of these benchmarks (Section 5.1);
- We propose a machine learning guided program tuning approach for TAs in OP-TEE, leveraging a predictive model to accelerate the optimization process, enhance performance, and reduce energy consumption on edge devices (Section 5.2).

2 Related Work

The TEE serves as an independent runtime environment equipped with specialized computing and storage capabilities to secure sensitive code and data. Among TEEs, ARM TrustZone has become a foundational security component for platforms, such as Samsung Knox[12], Android Keystore[13], and OP-TEE. Its versatility enables deployment across a variety of domains, including mobile devices, industrial systems, automotive platforms, and aerospace applications. Recent research has enhanced the TEE architecture through advancements in physical memory management[14], virtual memory systems[15, 16], and process isolation techniques to fortify security.

Recent efforts to optimize TEEs have explored various dimensions, including cache management, parallel execution, and task scheduling. BiTDB[17] addresses inefficiencies caused by redundant memory operations between the TEE and the REE, leading to significant performance gains. Similarly, Trusted Execution Environment parallelization (TEEp)[18] and LEAP[19] demonstrate the potential of multi-core execution within TEEs, distributing workloads across cores to enhance execution speed while maintaining security. However, such techniques often depend on hardware-specific capabilities, limiting their applicability across heterogeneous platforms. ETS-TEE[11] proposes task scheduling strategies by offloading computations to edge servers, reducing energy consumption based on task complexity. While

these approaches provide valuable contributions, they often neglect network variability and the challenges associated with heterogeneous TEE environments, limiting their effectiveness in dynamic real-world scenarios.

In addition, prior studies have explored using machine learning based optimization frameworks to tackle the challenges posed by dynamic workloads and system constraints in TEEs. Auto-tuning frameworks, such as OpenTuner[20] and machine learning based adaptive schedulers[21], leverage predictive models to balance performance and resource consumption dynamically. However, these methods typically rely on real-time data collection and online optimization, which introduces significant overhead on resource-constrained devices. In contrast, offline optimization strategies are gaining traction, where machine learning models are trained on historical data to predict performance and energy consumption without the need for continuous monitoring. Building on these prior efforts, our work introduces a novel offline optimization approach for TAs in OP-TEE. By employing a predictive model to estimate both performance and energy consumption based on the runtime characteristics of source programs in the REE, our approach eliminates the need for real-time data collection and compilation during the optimization process. This offline strategy ensures that the optimized TAs are resilient to network fluctuations and environmental variability, making them suitable for deployment across a wide range of OP-TEE-supported platforms. Our method addresses the limitations of existing online frameworks and provides a more reliable, low-overhead solution for improving TEE performance and energy efficiency.

3 Background and Motivation

3.1 OP-TEE

OP-TEE, an open-source TEE powered by ARM TrustZone technology, is designed specifically for ARM processors. Its primary function is to offer a secure execution environment for sensitive operations, protecting against threats such as malware, hacking, and physical tampering. OP-TEE's portability and lightweight architecture ensure compatibility across a wide range of hardware platforms and operating systems. By leveraging the standardized Application Programming Interface (API) provided by GlobalPlatform TEE[22], OP-TEE facilitates secure communication between the secure world and the non-secure world. Developers can use this API to create TAs that run within the secure world, accessing sensitive data and system resources. OP-TEE establishes a protected environment to execute critical tasks, ensuring the confidentiality and integrity of sensitive information.

3.2 Motivating examples

As a motivating example, we examine the runtime energy consumption and performance of the Secure Hash Algorithm 256 (SHA256)[23] algorithm within both the TEE and the REE. SHA256 is critical for information security on mobile platforms, playing a significant role in applications, such as fingerprint verification and Short Message Service (SMS) encryption, thereby safeguarding users' privacy. We compare the energy consumption and performance when SHA256 encrypts 20-byte messages in both the TEE and REE on a Raspberry Pi 3 Model B. As illustrated in Fig. 1, the encryption of a 20-byte message using SHA256 in the OP-TEE environment required 346.8 ms, which is significantly longer than the 63.7 ms required in the REE. In terms of energy consumption, the encryption process also demands more energy in the TEE than in the REE. This disparity arises from the TEE's design, which prioritizes security by isolating sensitive operations and enforcing stringent checks and encryption protocols. However, the additional security measures, combined with the time-intensive context switching between the normal and secure environments required for TAs, contribute to the increased processing costs observed in the TEE.

Table 1 provides a detailed breakdown of the runtime performance and energy consumption for each execution phase within the TEE. As shown in Table 1, the TEEC_InvokeCommand phase incurs the most significant delay in TA execution,, with a latency of 224.8 ms. While the TEE architecture enhances program security, this enhancement comes with an

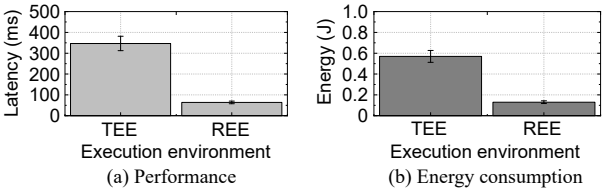


Fig. 1 Performance and energy consumption when using SHA256 to encrypt a 20-byte message in TEE and REE.

Table 1 Performance and energy consumption across five phases of using SHA-256 for encrypting 20-byte messages in OP-TEE on the Raspberry Pi 3 Model B.

Stage	Latency (ms)	Energy (J)
TEEC_InitializeContext	0.116	0.000 17
TEEC_OpenSession	106.799	0.175 50
TEEC_InvokeCommand	224.864	0.369 47
TEEC_CloseSession	15.073	0.024 80
TEEC_FinalizeContext	0.035	0.000 06

increased energy demand. Therefore, it is crucial to optimize the program for TA to reduce runtime overheads.

3.3 Our approach

We introduce a novel TA tuning approach to optimize TA runtime performance and energy consumption in the edge device. As illustrated in Fig. 2, our approach begins by collecting runtime features from C programs within the REE. These features are then fed into predictive models to estimate the performance and energy consumption of the corresponding transformed TA in the TEE. Optimization is achieved through iterative modifications of the C program using either an autotuning framework (e.g., OpenTuner[20]) or manual adjustments until the preset optimization goals are met. By employing this approach, we significantly reduce the overhead associated with repetitive TA compilation and practical measurement in the TEE, streamlining the tuning process and improving overall efficiency.

4 Convert C Program into Client Application (CA) and TA for OP-TEE

4.1 CA generation rules

C programs are extensively utilized in system programming due to their inherent efficiency and precise control over system resources. In this study, we adapt C programs for deployment within the OP-TEE framework by transforming them into CAs and TAs. As depicted in Fig. 3, the communication process between a CA and a TA within OP-TEE is governed by a well-defined procedure that utilizes the GlobalPlatform Internal API to facilitate communication between the non-secure world (where CAs reside) and the secure world (where TAs operate). This process ensures secure data exchange between the two environments, leveraging the TEE's security features to protect sensitive information. Following the communication protocols between CAs and TAs, Fig. 4 illustrates the transformation of C source programs into TAs. We have developed a script to automate the conversion of existing C programs into CAs and TAs. This process requires the identification and marking of specific code segments designated for execution within the TA.

Marking convention: First, the script scans the C source files for the designated markings that indicate the sections of the code intended for execution as TAs.

Generate stubs: The script then automatically extracts the marked sections and organizes them into separate source files or modules for the TA. In the original C code, stub functions are generated in place of the extracted TA functions. These stubs handle the communication with the TA by using the TEE Client API to invoke the corresponding TA functions.

Compile CA and TA: The script invokes the compiler to build the CA, which consists of the original source code minus the extracted functions, and the TA, which includes the extracted functions. We need to follow the specified rules to establish a connection between the TA and the CA. The CA needs to go through the following stages to establish a communication connection with the TA.

We initialize a TEEC_Context variable "ctx" with the TEE subsystem, which allows the CA to invoke commands in the TEE by using the function: TEEC_InitializeContext (&ctx). At the same time, we initialize the TEEC_Result variable res, record the return value of the function, and check whether the invoke was successful.

A TEE session "sess" is opened to establish a communication channel between the CA and the TA using the previously obtained TEEC_Context variable. The specific TA to connect with is identified by its uuid. This process utilizes the function: TEEC_OpenSession (&ctx, &sess, &uuid), where the

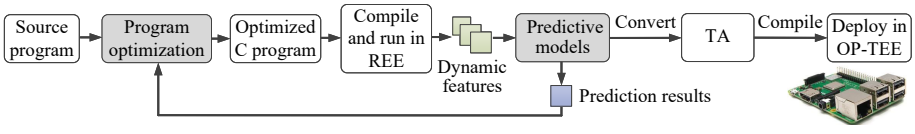


Fig. 2 Overview of approach.

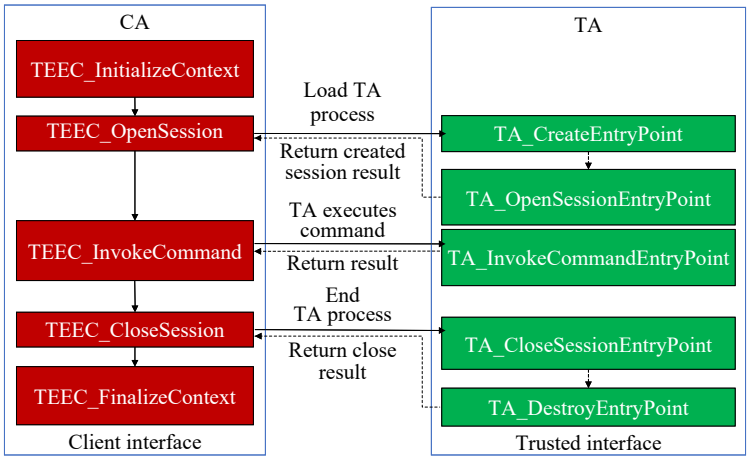


Fig. 3 Communication process between CA and TA in the OP-TEE.

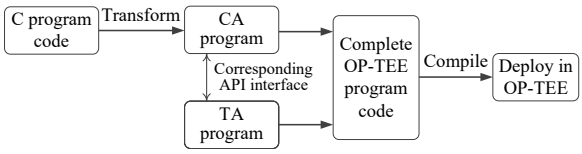


Fig. 4 OP-TEE code generation and deployment process.

TA's uuid is passed as a parameter to specify the target TA for communication. We record the return value of the function and determine whether the communication was successfully established.

Passing parameters from a CA to a TA involves using the TEEC_Operation structure. First, initialize the variable op of type TEEC_Operation. Then, zero all fields of the TEEC_Operation structure by using memset (&op, 0, sizeof (op)) to ensure no unintended data from previous operations is included. Next, use the macro TEEC_PARAM_TYPES (TEEC_VALUE_INOUT, TEEC_NONE, TEEC_NONE, TEEC_NONE), which defines up to four parameter types (with TEEC_NONE indicating no input for each parameter) within a single operation (TEEC_VALUE_INOUT). For direct values (such as integers), assign the value directly to the appropriate params [x].value.a (or b for a second value). For memory references (such as arrays or strings), specify the buffer's address and size in params[x].tmpref.buffer and params[x].tmpref.size.

Once the session is opened, the CA can invoke commands on the TA using the function: TEEC_InvokeCommand (&sess, cmd_id, &op). In this function, the command ID (cmd_id) is specified, along with any necessary data to be sent to the TA. The TA processes the command and returns the result along with any associated data through the params member of the TEEC_Operation structure.

After the command execution and any necessary data exchange, close the session with the functions TEEC_CloseSession (&sess) and TEEC_FinalizeContext (&ctx).

4.2 TA generation rules

When the TEEC_Context variable ctx is initialized, the function TA_CreateEntryPoint (void) is called, serving as the entry point for a TA. This function handles any initial setup required by the TA before CA.

Next, the function TA_OpenSessionEntryPoint (&param_types, &params, &ctx) is invoked, receiving parameters from the CA. The param_types specifies the types of parameters being passed to the TA, while params contains the actual parameter values. These parameters may influence the session's initialization or determine the resources required for the session.

Upon receiving commands from the CA, the function TA_InvokeCommandEntryPoint (&sess_ctx, &cmd_id, &param_types, &params) is invoked. This function contains the core logic of the TA, executing operations based on the command identifier cmd_id and parameters provided by the CA.

4.3 Shared memory generation rules

Shared memory allows both CA and TA to access the same physical memory locations, facilitating efficient data exchange between the two environments. To register a memory region on the CA side, a buffer is allocated in the non-secure world, allowing both the CA and TA to access it for seamless data exchange by

calling function TEEC_RegisterSharedMemory (TEEC_Context *ctx, TEEC_SharedMemory *shm); The "shm" is a pointer to a TEEC_SharedMemory structure allocated by the caller. Before invoking TEEC_RegisterSharedMemory, the caller must populate this structure with the necessary details about the memory block to be registered, including its address, size, and access flags.

Once the shared memory is no longer needed, it should be released to free up resources. This is accomplished by calling function TEEC_ReleaseSharedMemory (TEEC_SharedMemory *shm).

5 Predictive Model

The core of our approach is the predictive model. In this study, we develop a pair of models based on Artificial Neural Networks (ANNs) to estimate the performance and energy consumption of TAs in the TEE, leveraging the runtime features of the C program within the REE. ANNs are selected for building the predictive models due to their superior performance on the dataset used in this study (as discussed in Section 5.2.3). Additionally, ANNs offer the advantage of facilitating model migration to new usage scenarios and application data, allowing for adaptation without retraining the model from scratch.

5.1 Training data generation

5.1.1 Standard C benchmark suites

Figure 5 illustrates the training process for our two predictive models. The initial step in generating data for these models involves collecting standard benchmarks from eight widely used C benchmark suites: Bristol/Embecosm Embedded Benchmark Suite (BEEBS)[23], Sysbench[24], BYTE UNIX benchmark suite (Unixbench)[25], MiBench[26], Whetstone[27], Dhrystone[28], Linear Algebra Package[29], and CoreMark[30]. These benchmarks are then transformed into TAs and CAs (see Section 4 for details) for execution within the TEE. We then measure the runtime latency and energy consumption for each benchmark both in the REE and the corresponding transformed TA in the TEE. Table 2 presents the energy consumption and performance of the BEEBS benchmarks in both the REE and TEE. The transformed TA, driven by security requirements, demonstrates a significant reduction in performance, averaging a 72.81% slowdown, and an increased energy consumption of 72.03% compared to the execution of C programs in the REE.

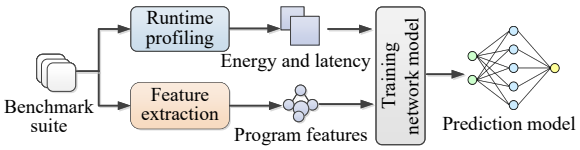


Fig. 5 Training process of the predictive model.

5.1.2 Expanding the benchmarks

To expand the dataset, we follow three key principles to enhance the current benchmarks:

(1) Data-intensive: Introduce larger, more complex datasets or datasets from different domains to test the scalability and efficiency of data processing tasks.

(2) Parallel workloads: Expand the benchmarks to include multi-threaded or parallel computing tasks to evaluate the performance of multi-core systems and parallel processing.

(3) Repetition loops: Incorporate for-loops or while-loops that repeatedly execute a block of code a fixed number of times, assessing how performance scales with increasing operations.

As a result, we generate 425 benchmarks, with

Table 2 Runtime performance and energy consumption of the BEEBS benchmarks.

Benchmark	Latency (ms)		Energy consumption (J)	
	OP-TEE	REE	OP-TEE	REE
CRC32	5368.2870	19.6700	9.2870	0.0320
Dijkstra	152.9550	29.2900	0.2610	0.0470
Cubic	0.2670	0.0600	0.0004	0.0001
Fdct	815.9250	46.0100	1.4030	0.0730
2D Fir	8074.5680	4000.7000	13.6100	6.5950
Multi_int	190.0000	67.0000	0.3000	0.1070
Multi_float	195.0000	36.0000	0.3070	0.0590
Rijndael	264.0000	47.6400	0.4370	0.0750
SHA	346.8880	63.7000	0.5700	0.1300

340 used for training and 85 for testing. Specifically, we measure the performance and energy consumption of each benchmark within both the REE and the TEE. To facilitate this, we develop a script to collect the runtime latency of each benchmark and utilize the POWER-Z KM001 device for precise energy consumption measurements.

5.2 Model training

The predictive model is a fully connected feed-forward ANN with three hidden layers. The number of nodes in each layer is determined by the number of input features, and the ANN architecture is optimized using Auto-PyTorch[31] based on the training dataset used in this study.

5.2.1 Feature extraction

In this study, we collect the runtime characteristics of the benchmarks within the REE. Table 3 lists these 11 raw features, and Fig. 6 presents the feature selection process.

Feature importance. Figure 7 presents a Hinton diagram ranking the importance of the 11 original features for both the performance and energy consumption models. The size of each box is proportional to the feature's contribution to the overall model performance. To determine feature importance, we first train the model using all raw features to establish a baseline accuracy. Then, each of the 11 features is individually removed, and the model is retrained with the remaining 10 features. The variations in accuracy are recorded and normalized to calculate a relative importance percentage for each feature, reflecting its impact on model performance. As shown in Fig. 7, Instructions Per Cycle (IPC) and branches are identified as the most influential features.

Feature correlation. Figure 8 presents the correlation coefficient values between features, ranging from 0 to 1. A correlation coefficient closer to 1 indicates a stronger relationship between two features. Typically, a correlation value above 0.75 is considered highly correlated. To reduce feature processing

Table 3 Raw features.

Feature	Description
CPU_usage (%)	Average CPU utilization during runtime
Context_switches	Number of context switch occurs
Page_faults	Number of program triggers a page fault during its execution
CPU_frequency	Average CPU frequency
IPC	Number of IPCs
Branches	Number of branch instructions
Branch-misses	Number of branch prediction failures
L1-dcacheloads	Number of load (read) operations that hit the L1 data cache
L1-dcacheload-misses	Number of unsuccessful data retrieval attempts from the L1 data cache
LLC-loads	Number of load operations that access the Last Level Cache (LLC)
LLC-load-misses	Number of last level cache read failures

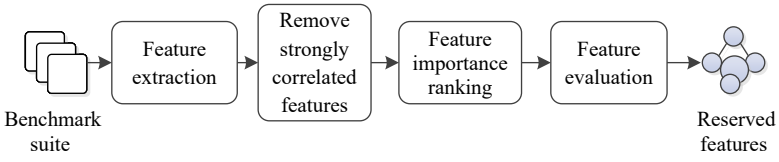


Fig. 6 Feature selection process.

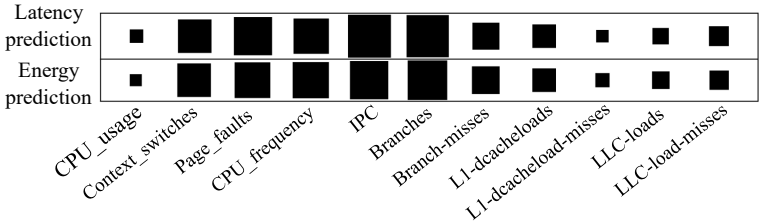


Fig. 7 Feature importance. A larger area of the box indicates a larger impact on the accuracy of the model.

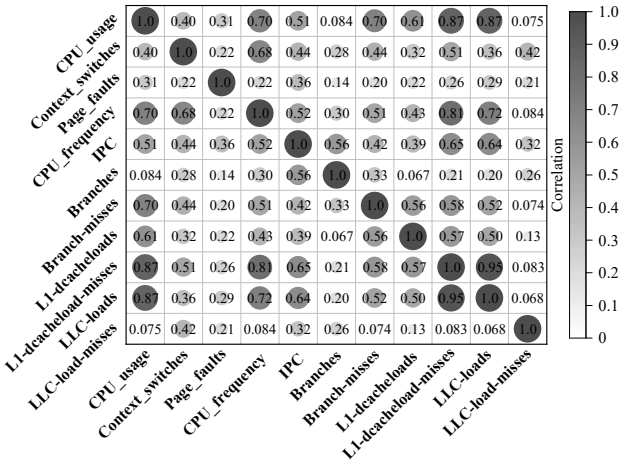


Fig. 8 Feature correlation heatmap. The gray scale represents the correlation between the features, and a higher gray scale indicates a higher correlation between two features.

overhead, we remove one of the features when the correlation exceeds 0.75. For example, with a correlation coefficient of 0.95 between LLC-loads and L1-dcacheload-misses, we remove L1-dcacheload-misses. Ultimately, we retain 9 features for building the model: Context_switches, Page_faults, CPU_frequency, IPC, Branches, Branch_misses, L1_dcacheloads, LLC_loads, and LLC_load_misses. To ensure that the scale of feature values does not affect their importance, we apply z-score normalization, which scales feature values to a range between 0 and 1.

5.2.2 Evaluation method

To evaluate the performance of the predictive model during training, we employ a five-fold cross-validation method. The 425 benchmarks are evenly divided into five subsets, each containing 85 benchmarks. In each validation cycle, one subset is used for testing, while the remaining four subsets are used for training. This process is repeated five times, ensuring that each subset is used once as the test dataset. The model's performance is assessed by calculating the average error rate from the regression algorithm across all five cycles.

5.2.3 Alternative predictive modeling techniques

We evaluate the effectiveness of five regression algorithms, Linear Regression (LR), Polynomial Regression (PR), Support Vector Regression (SVR), ANN, and Random Forest (RF), on our dataset. As illustrated in Fig. 9, the ANN-based predictive model demonstrates the lowest error rate, surpassing the other methods.

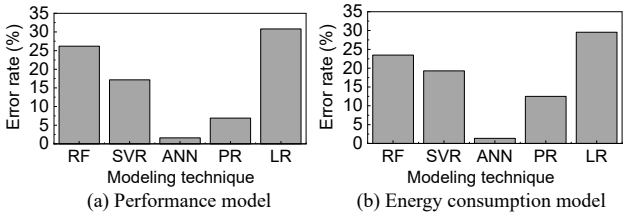


Fig. 9 Comparing our ANN-based predictor in performance and energy consumption with other modeling techniques.

6 Evaluation Setup

6.1 Platforms

We utilize the Raspberry Pi 3 Model B as the mobile edge device. It has a 64-bit quad-core ARM Cortex-A53 processor and 1 GB Random Access Memory (RAM). The OP-TEE version is 3.7.0[9]. To accurately measure the real-time power consumption of the Raspberry Pi 3B while executing programs, we employ the Power-Z KM001, which is an external hardware measurement device. The server configuration includes a 16-core, 3.5 GHz Intel Core i9-11900K CPU and an NVIDIA GeForce RTX 3090 GPU.

6.2 Measurement

Performance. We use a script to monitor the performance of each benchmark. Specifically, we leverage the C standard library function gettimeofday(·) to precisely measure latency at the microsecond level.

Energy consumption. To measure energy consumption, we use the external device Power-Z KM001, which captures voltage, current, power, energy, and temperature during program execution on the Raspberry Pi 3B. The Power-Z unit connects to the Raspberry Pi via a USB interface, positioned between the device and its power source. To ensure precise data capture, we set the sampling rate to 1000 Hz.

7 Experimental Result

This study focuses on optimizing the static code performance and energy consumption of TAs within the OP-TEE environment. As a case study, we use OpenTuner[20], an automatic tuning framework, to optimize a matrix multiplication program executed in the TEE. Algorithm 1 presents the code details, where it calculates the matrix multiplication $A \times B$ and stores the result in C, with the matrix size N being set to 100. In this implementation, matrices A and B are declared but not explicitly initialized, and their contents are

Algorithm 1 Matrix multiplication algorithm

Input: Matrices A, B, and C with size $N \times N$, and block size BLOCK_SIZE

Output: Result matrix C

$n \leftarrow \text{BLOCK_SIZE} \times \left(\frac{N}{\text{BLOCK_SIZE}} \right)$;
for $k_1 = 0; k_1 < n; k_1 = k_1 + \text{BLOCK_SIZE}$ **do**
  for $j_1 = 0; j_1 < n; j_1 = j_1 + \text{BLOCK_SIZE}$ **do**
    for $k_2 = 0; k_2 < n; k_2 = k_2 + \text{BLOCK_SIZE}$ **do**
      for $i = 0; i < n; i++$ **do**
        for $j = j_1; j < j_1 + \text{BLOCK_SIZE}; j++$ **do**
          sum $\leftarrow C[i][j]$;
          for $k = k_2; k < k_2 + \text{BLOCK_SIZE}; k++$ **do**
            sum $\leftarrow$ sum $+ A[i][k] \times B[k][j]$;
          end for
          $C[i][j] \leftarrow$ sum;
        end for
      end for
    end for
  end for
end for
return C

empty. While the computed values in C are not meaningful in a numerical sense, this setup is sufficient for testing the performance of the blocked matrix multiplication algorithm. Our goal is to determine the optimal block size for dividing the matrix to improve both performance and energy efficiency during the multiplication process. OpenTuner plays a crucial role in identifying the most suitable block size. The transformation of the source program into a TA is deferred until the optimal block size is found. To optimize the energy consumption of the TA, our predictive model is integrated into OpenTuner as the cost model, enabling it to predict the energy consumption of the TA across various block sizes. The search time is capped at 200 s, ensuring a timely optimization process. Once the optimal block size is identified, the C code is converted into the corresponding TA with the selected block size. This approach significantly accelerates the search process and eliminates compilation costs, enhancing the efficiency of the tuning workflow.

The experimental results show that a BLOCK_SIZE of 34 yields the best performance and lowest energy consumption for executing matrix multiplication within OP-TEE. Figure 10 compares the performance and energy consumption of our approach against the default setup (without blocking) in both REE and TEE environments. As shown in Fig. 10, our approach achieves a significant reduction in latency, 49.7% in the REE and 27.2% in the transformed TA for use in TEE, compared to the default configuration. In terms of energy consumption, our approach outperforms the default by 45.5% in the REE and 25.0% in TEE. Additionally, the predictive model employed exhibits error rates ranging from 3.1% to 4.7%, indicating the model's consistency and efficacy in our approach.

In the context of optimizing TA runtime performance, Fig. 11 presents the search time and performance metrics achieved by our approach compared to the default method. Leveraging our predictive model, our approach identifies the optimal value within 200 s, achieving a 196× speed-up over the default method. This substantial efficiency gain stems from the limitations of the conventional method, which requires measuring actual latency, feeding the results back into OpenTuner, and recompiling a new TA for every block size change, introducing significant compilation overhead. The default approach incurs prohibitively time-consuming compilation overhead for the TA. Specifically, Fig. 12a illustrates the number of iterations completed within 2000 s by both our approach and the default method. Our approach, driven by the predictive model, completes 2160 iterations, significantly outperforming the default method. Furthermore, Figs. 12a and 12c demonstrate that our approach identifies the optimal block size in just 170 s, compared to 42 min with the default method. By enabling more iterations within the same time frame, our approach greatly enhances the efficiency of TA performance optimization.

7.1 Discussions and future work

The proposed offline tuning approach, although evaluated within the context of OP-TEE, has broader applicability beyond this specific TEE environment. Key components of our framework—such as predictive modeling, feature extraction, and offline optimization—can be adapted to other TEEs like Intel SGX and Google's Trusty TEE. While each TEE presents unique architectures and APIs, the core principles of optimization remain consistent.

7.1.1 Support for Intel SGX and other TEEs

Our predictive model can incorporate SGX-specific features, such as enclave lifecycle management, memory sealing, and interrupt handling. Although

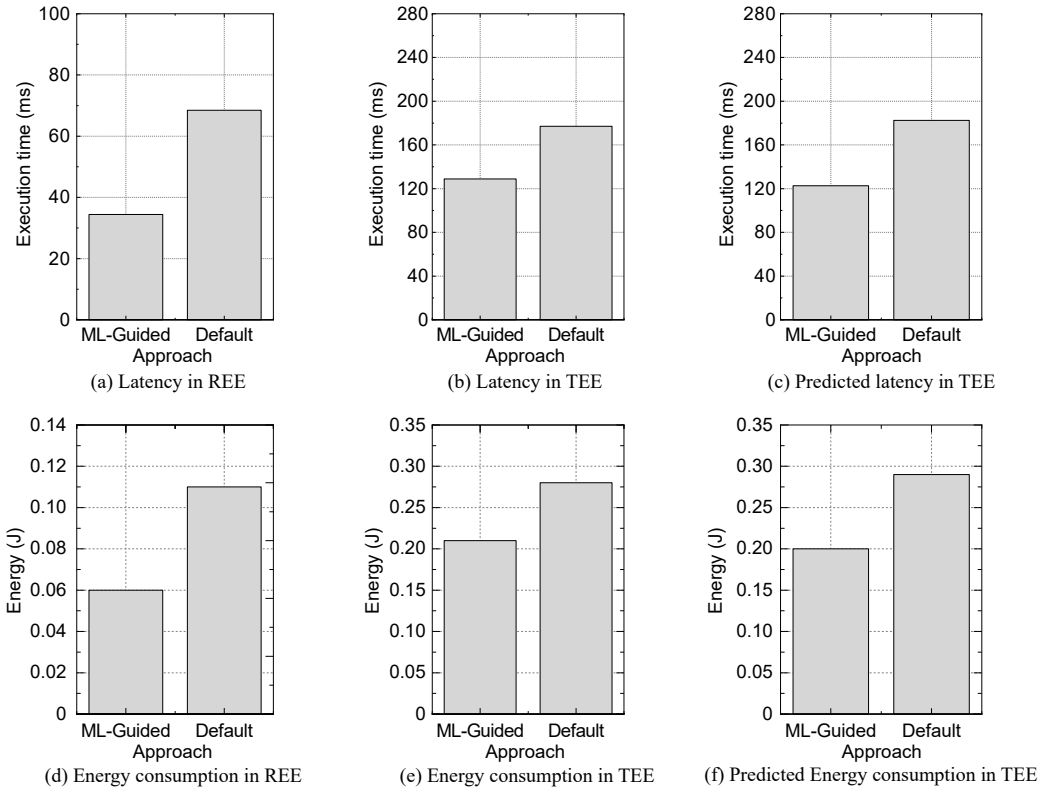


Fig. 10 Comparison of the performance and energy consumption of matrix multiplication when executed in REE and TEE using the optimal block size identified by our approach (machine-learning guided approach, namely ML-Guided) and the default block size. The TEE prediction results, showing less than a 5% error rate, prove the effectiveness of our predictive model.

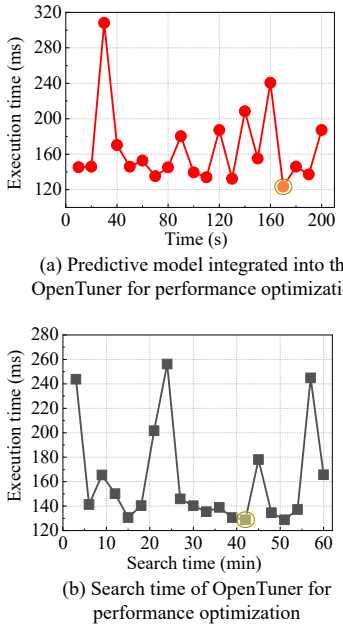
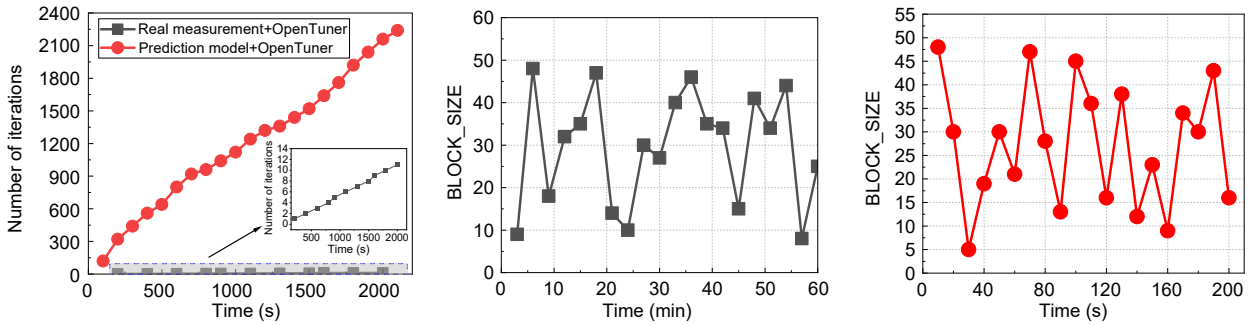


Fig. 11 Runtime latency of TA by using the searched block size from our approach versus the default approach. Our method significantly accelerates the search process. The high bright spots represent the optimal result searched.

SGX introduces overheads like enclave paging, these operations can be optimized using the same strategies we employ for context switching in OP-TEE. The auto-tuning framework can also mitigate the overhead associated with enclave transitions, comparable to the secure session management used in OP-TEE. While our current approach focuses on offline tuning, future work will explore hybrid strategies that combine offline and dynamic optimization. This will involve real-time parameter adjustments to adapt to fluctuating network conditions or workload changes, enabling more responsive optimization without compromising performance or security.

7.1.2 Generalization to non-TEE frameworks

Beyond TEE applications, the framework can be extended to other secure platforms, such as sandboxed environments, hypervisors, and containerized systems. Even in non-secure or hybrid environments, our predictive modeling approach can help reduce latency and reduce resource consumption during computationally intensive operations, improving performance without sacrificing reliability. As part of



(a) Number of search iterations for our approach vs. default tuning approach within 2000 s

(b) Search time and corresponding block size using the default tuning approach

(c) Search time and corresponding block size using our approach

Fig. 12 Performance of our approach and default approach when searching for the optimal BLOCK_SIZE.

future work, we plan to extend the framework to non-TEE environments, including sandboxed processes and containerized applications, to assess the versatility of our tuning strategy. This will allow us to explore broader applications for predictive energy-performance optimization across a range of computing environments.

8 Conclusion

ARM TrustZone provides robust protection for sensitive data, but its security mechanisms result in higher energy consumption, impacting the battery life of mobile edge devices. Our work introduces an automated mechanism for transforming C source code into TAs within the TEE. We further develop a benchmark suite for the OP-TEE environment to evaluate both energy consumption and performance. Building on this, we propose a machine learning-based approach to optimize TAs by integrating a predictive model into the program tuning framework. Our approach significantly accelerates the offline tuning process and achieves substantial improvements in runtime latency and energy consumption for TAs within the TEE. Experimental results demonstrate that our approach minimizes the need for repeated compilation, deployment, and testing within the OP-TEE environment. In a case study, our approach achieves a 196× acceleration over traditional tuning methods. The optimized TA delivers an average performance improvement of 27.23% and achieves a 25% reduction in energy consumption.

Acknowledgment

This work was supported by the National Natural Science Fundation of China (Nos. 62372281 and 62072362), the Shaanxi Province 2024 Innovation Capacity Support Program (No. 2024ZC-KJXX-014), and the Natural Science Basic Research Plan in Shaanxi Province (No. 2021JQ-607).

References

[1] P. Mach and Z. Becvar, Mobile edge computing: A survey on architecture and computation offloading, *IEEE Commun. Surveys Tut.*, vol. 19, no. 3, pp. 1628–1656, 2017.

[2] J. Amacher and V. Schiavoni, On the performance of arm trustzone, in *Proc. 19th IFIP WG6.1 Int. Conf. Distributed Applications and Interoperable Systems*, Kongens Lyngby, Denmark, 2019, pp. 133–151.

[3] Intel Corporation, Intel® Software Guard Extensions (Intel® SGX), https://www.intel.com/content/www/us/en/developer/tools/software-guard-extensions/overview.html, 2025

[4] Google, Trusty TEE, https://source.android.com/docs/security/features/trusty, 2025.

[5] Z. Jian, Y. Lu, Y. Qiao, Y. Fang, X. Xie, D. Yang, Z. Zhou, and T. Li, TSC-VEE: A TrustZone-based smart contract virtual execution environment, *IEEE Trans. Parallel Distrib. Syst.*, vol. 34, no. 6, pp. 1773–1788, 2023.

[6] Z. Liu, Y. Lu, X. Xie, Y. Fang, Z. Jian, and T. Li, Trusted-DNN: A TrustZone-based adaptive isolation strategy for deep neural networks, in *Proc. ACM Turing Award Celebration Conf.-China*, Hefei, China, 2021, pp. 67–71.

[7] F. Mo and H. Haddadi, Efficient and private federated learning using TEE, in *Proc. 14th European Conf. Computer Systems*, Dresden, Germany, 2019, p. 1.

[8] S. Pinto, P. Machado, D. Oliveira, D. Cerdeira, and T. Gomes, Self-secured devices: High performance and secure I/O access in TrustZone-based systems, *J. Syst. Architect.*, vol. 119, p. 102238, 2021.

[9] Linaro, OP-TEE, https://optee.readthedocs.io/en/3.7.0/general/releases.html, 2025.

[10] H. Park, S. Zhai, L. Lu, and F. X. Lin, StreamBox-TZ: Secure stream analytics at the edge with TrustZone, in *Proc. 2019 USENIX Annual Technical Conf.*, Renton, WA, USA, 2019, pp. 537–554.

[11] H. Wang, L. Cai, X. Hao, J. Ren, and Y. Ma, ETS-TEE:

An energy-efficient task scheduling strategy in a mobile trusted computing environment, *Tsinghua Science and Technology*, vol. 28, no. 1, pp. 105–116, 2023.
[12] U. Kanonov and A. Wool, Secure containers in android: The Samsung KNOX case study, in *Proc. 6th Workshop on Security and Privacy in Smartphones and Mobile Devices*, Vienna, Austria, 2016, pp. 3–12.
[13] Android, Android Keystore, https://developer.android.com/training/articles/keystore.html, 2025.
[14] D. Lee, D. Kohlbrenner, S. Shinde, K. Asanović, and D. Song, Keystone: An open framework for architecting trusted execution environments, in *Proc. 15th European Conf. Computer Systems*, Heraklion, Greece, 2020, p. 38.
[15] Z. Ling, J. Luo, Y. Xu, C. Gao, K. Wu, and X. Fu, Security vulnerabilities of internet of things: A case study of the smart plug system, *IEEE Internet Things J.*, vol. 4, no. 6, pp. 1899–1909, 2017.
[16] GlobalPlatform, Sierraware makes GlobalPlatform-compliant Trusted Execution Environment (TEE) available for MIPS-based devices, https://globalplatform.org/latest-news/sierraware-makes-globalplatform-compliant-trusted-execution-environment-tee-available-for-mips-based-devices, 2025
[17] C. Ma, D. Lu, C. Lv, N. Xi, X. Jiang, Y. Shen, and J. Ma, BiTDB: Constructing a built-in TEE secure database for embedded systems, *IEEE Trans. Knowledge Data Eng.*, vol. 36, no. 9, pp. 4472–4485, 2024.
[18] Z. Li, W. Li, Y. Xia, and B. Zang, TEEp: Supporting secure parallel processing in ARM TrustZone, in *Proc. 2020 IEEE 26th Int. Conf. Parallel and Distributed Systems* (*ICPADS*), Hong Kong, China, 2020, pp. 544–553.
[19] L. Sun, S. Wang, H. Wu, Y. Gong, F. Xu, Y. Liu, H. Han, and S. Zhong, LEAP: TrustZone based developer-friendly TEE for intelligent mobile apps, *IEEE Trans. Mob. Comput.*, vol. 22, no. 12, pp. 7138–7155, 2023.
[20] J. Ansel, S. Kamil, K. Veeramachaneni, J. Ragan- Kelley, J. Bosboom, U. M. O'Reilly, and S. Amarasinghe, OpenTuner: An extensible framework for program autotuning, in *Proc. 23rd Int. Conf. Parallel Architecture and Compilation Techniques*, Edmonton, Canada, 2014, pp. 303–316.
[21] U. Saleem, Y. Liu, S. Jangsher, Y. Li, and T. Jiang, Mobility-aware joint task scheduling and resource allocation for cooperative mobile edge computing, *IEEE Trans. Wirel. Commun.*, vol. 20, no. 1, pp. 360–374, 2021.
[22] OP-TEE, GlobalPlatform API, https://optee.readthedocs.io/en/latest/architecture/globalplatform_api.html, 2025.
[23] J. Pallister, S. Hollis, and J. Bennett, BEEBS: Open benchmarks for energy measurements on embedded platforms, arXiv preprint arXiv: 1308.5174, 2013.
[24] Alexey Kopytov, Sysbench, https://github.com/akopytov/sysbench, 2025.
[25] Contributors, byte-unixbench, https://github.com/kdlucas/byte-unixbench, 2025.
[26] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, MiBench: A free, commercially representative embedded benchmark suite, in *Proc. 4th Annual IEEE Int. Workshop on Workload Characterization*, Austin, TX, USA, 2001, pp. 3–14.
[27] R. Longbottom, Whetstone benchmark history and results, http://www.roylongbottom.org.uk/whetstone.htm, 2025.
[28] R. P. Weicker, Dhrystone benchmark: Rationale for version 2 and measurement rules, *ACM SIGPLAN Notices*, vol. 23, no. 8, pp. 49–62, 1988.
[29] J. J. Dongarra, P. Luszczek, and A. Petitet, The LINPACK benchmark: Past, present and future, *Concurr. Comput.*, vol. 15, no. 9, pp. 803–820, 2003.
[30] Embedded Microprocessor Benchmark Consortium (EEMBC), CoreMark Benchmark, https://www.eembc.org/coremark/, 2025.
[31] AutoML, Auto-PyTorch, https://github.com/automl/Auto-PyTorch, 2025.



Yuhua Wang received the BS degree in computer science from China University of Mining and Technology in 2021. Currently, he is a master student in computer science and technology at Shaanxi Normal University, China. His research interest is mainly focused on the field of mobile edge computing.



Feng Tian received the BEng and PhD degrees from Xi'an Jiaotong University, Xi'an, China in 2009 and 2015, respectively. From 2018 to 2019, he was a visiting scholar at Department of Electrical and Computer Engineering, University of Waterloo, UK. He is currently an associate professor at School of Artificial Intelligence and Computer Science, Shaanxi Normal University, China. His research interests include data privacy in cloud computing and network security.



Xiaoao Zhu received the BS degree in computer science from Shaanxi Normal University in 2023, where she is currently a master student in software engineering. Her research interest is mainly focused on Low Level Virtual Machine (LLVM).



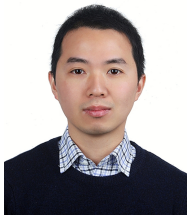
Jie Ren received the PhD degree in computer system architecture from Northwest University, China in 2017. His work has been published in INFOCOM, Ubicomp, CoNEXT, *TMC*, *TCOM*, and *TSE*. He is currently an associate professor at School of Artificial Intelligence and Computer Science, Shaanxi Normal University, China. His research interests include mobile computing and compiler optimization.



Jie Zheng received the BEng degree in communications engineering from Nanchang University, China in 2008, and the PhD degree from Xidian University, China in 2014. He is currently an associate professor at School of Information Science and Technology, Northwest University, Xi'an, China. His research interests include energy-efficient transmission, wireless resource allocation, and edge intelligence.



Rui Cao received the PhD degree in computer system architecture from Northwest University, China in 2023. He is currently a lecturer at the School of Information Science and Technology, Northwest University, Xi'an, China. He has participated in several national research projects. His research interests include neural language processing, multimodal data mining, and applications. Relevant results have been published in finding of ACL, *Knowledge-Based System*, ICANN, MIDL, and *Heritage Science*.



Zheng Wang received the PhD degree from University of Edinburgh, UK. He is currently the chair professor of intelligent software technology at School of Computing, University of Leeds, UK, and a turing fellow at Alan Turing Institute, UK. His research focuses on accelerating large-scale deep learning models, program optimisation, applied machine learning, and system security.