IP Protection in TinyML

Jinwen Wang[1‡], Yuhao Wu[1‡], Han Liu[1], Bo Yuan[2], Roger Chamberlain[1], Ning Zhang[1]
[1]Department of Computer Science & Engineering, Washington University in St. Louis
[2]Department of Electrical and Computer Engineering, Rutgers University
{jinwen.wang, yuhao.wu, h.liu1, roger, zhang.ning}@wustl.edu; bo.yuan@soe.rutgers.edu

2023 60th ACM/IEEE Design Automation Conference (DAC) | 979-8-3503-2348-1/23/$31.00 ©2023 IEEE | DOI: 10.1109/DAC56929.2023.10247898

***Abstract*—Tiny machine learning (TinyML) is an essential component of emerging smart microcontrollers (MCUs). However, the protection of the intellectual property (IP) of the model is an increasing concern due to the lack of desktop/server-grade resources on these power-constrained devices. In this paper, we propose *STML*, a system and algorithm co-design to Secure IP of TinyML on MCUs with ARM TrustZone. Our design jointly optimizes memory utilization and latency while ensuring the security and accuracy of emerging models. We implemented a prototype and benchmarked with 7 models, demonstrating *STML* reduces 40% of model protection runtime overhead on average.**

I. Introduction

With recent advances in deep learning (DL) [1], there is a growing need to deploy the machine learning (ML) models on smart microcontrollers (MCUs) at the Edge for communication efficiency and privacy protection. This deployment paradigm on MCUs is often referred to as tiny machine learning (TinyML) [2]. Nevertheless, deploying these costly deep learning models that require considerable effort to train onto low-cost MCUs with less security protection raises considerable concerns for manufacturers seeking to safeguard their valuable models against intellectual property (IP) theft.

Existing Attacks and Defense on Deep Learning Models: Due to its importance, the security of deep learning has received significant attentions, focusing on key security properties, such as confidentiality [3], [4], integrity [5], [6], and availability [7]. Among these attacks, model stealing attack [4] is closely related to model IP protection. These attacks can be generally divided into two categories, system approach [3] (via illegal memory access or side channel) or algorithm approach [4] (via interaction with the model). Due to the prevalence of API-based access paradigms, most existing work focuses on defense against algorithm-level attacks using watermarking, differential privacy, or data perturbation [8], little has been done to protect ML deployment on MCUs.

Lack of IP Protection Mechanism on MCU: To prevent IP thief, models can be encrypted while at rest. However, an attacker who compromises the software on the MCU can still extract the model from the memory while the system is in use. To defend against such attacks, we propose to leverage trusted execution environment (TEE), an increasingly available feature on modern embedded processors. Different from the existing TEE-based methods that focus on high-end devices [9]–[13], leveraging this hardware feature on MCU for model IP protection presents several unique challenges.

‡ Equal Contribution.

Challenges and Our Solution: In this paper, we propose *Secure TinyML* (*STML*) to protect model IP on MCUs under an untrusted software stack based on commercial off-the-shelf hardware. There are two main challenges:
1) Constrained Memory. TEE utilizes isolation to safeguard memory contents, but when implemented naively with static memory allocation for the secure world and normal world on MCUs, it can lead to memory scarcity for DL inference execution in the secure world and other tasks in the normal world due to the limited available memory resources. To address this challenge, *STML* dynamically allocates memory for applications in the secure world and normal world according to runtime requirements, swapping essential contents between internal memory and external storage during world switches to create space for efficient task execution.
2) Co-Optimization. The memory swapping during world switches of TrustZone and the use of cryptographic operations for swapped data protection significantly increase the runtime latency of DL execution. Previous work [14] considers three key dimensions, i.e, memory usage, latency, and accuracy, without security considerations. To address the challenge, we formulate and solve a unified TinyML optimization problem considering the security, memory, latency, and accuracy. *STML* can optimize latency and memory usage at both the system level and algorithm level while meeting the security and accuracy requirements.

Contributions: We have made the following contributions.

- We propose *STML*, a system designed to protect the IP of TinyML models in the presence of untrusted privileged software by leveraging trusted execution environment.
- We propose an adaptive optimization framework to search the memory resource allocation strategies with the goal to minimize the model execution latency.
- We implement and evaluate *STML* on STM32L562E-DK MCU with 7 benchmark models. Results show that *STML* protects model IP while reducing 40% runtime overhead than non-optimized security mechanism.

II. Background and Related Work

Tiny Machine Learning: TinyML [2] is at the intersection of low-end embedded devices and deep learning. As both academia and industry show growing interest in this area, several ML frameworks for enabling TinyML on embedded systems have been developed, including Tensorflow Lite Micro (TFLM), Embedded Learning Library (ELL), Graph Lowering

Authorized licensed use limited to: WASHINGTON UNIVERSITY LIBRARIES. Downloaded on May 10,2025 at 21:32:31 UTC from IEEE Xplore. Restrictions apply.

(GLOW). Despite these advancements, there remains significant challenges in implementing real-world TinyML applications from different aspects, including memory usage, power consumption, network throughput, data privacy, reliability, and robustness [15]. On the algorithm-level, TinyML models can be optimized to fit specific execution environments [16], while on the system-level, execution environments or ML frameworks can be designed or customized to run specific TinyML models [17]. In this work, we integrate both approaches by firstly building a secure execution environment tailored for TinyML applications, followed by optimizing the model to seamlessly fit within this environment.

TEE for Deep Learning Protection: TEE provides a powerful abstraction of a trusted machine to guarantee the confidentiality and integrity of code and data loaded inside [18], [19]. Among different TEE solutions, TrustZone [20] is widely deployed on devices with ARM processors (both Cortex-M and Cortex-A). There are a secure world and a normal world divided through hardware-enforced isolation. The sensitive data and processing in the secure world can be protected even if the normal world is compromised. As TEE technology advances, it is increasingly being utilized to ensure the confidentiality and integrity of on-device ML execution [9]–[13]. TEE-based Model IP protection [11], [13], in particular, has gained popularity, as models are typically derived from extensive computational resources and proprietary data during training and processing. Nevertheless, existing TEE-based IP protection approaches target high-end devices and are not directly applicable to MCUs, which possess significantly less computational and memory resources.

III. Threat Model and System Goal

Threat Model: We assume that the device hardware supports a TEE such as ARM TrustZone. The hardware and secure world software stack are attested during system initialization and trusted. The normal world software stack is not trusted and attackers aim to steal model details, including architecture and parameters, in the system. Specifically, attackers attempt to extract DL model details by arbitrarily reading or writing normal world memory, executing normal world code, and accessing peripherals accessible from the normal world, such as SD cards shared between the normal world and secure world. Denial of service, cryptography-based attacks, side-channel attacks, and physical attacks are out of our scope.

System Goal: *STML* aims to protect TinyML IP on MCUs, i.e., preventing attackers from extracting model architecture and parameters, while minimizing model execution latency and maintaining developer-specified accuracy requirements.

IV. System Design

A. System Overview

STML protects the IP of TinyML models using a system and algorithm co-design approach. As shown in Fig. 1, *STML* consists of an offline optimization engine and a runtime IP protection mechanism. The offline optimization engine outputs a resource allocation strategy to minimize the TinyML task execution delay. Specifically, it first generates a storage strategy for storing model parameters within internal flash memory, optimizing flash memory usage and minimizing the overhead associated with loading parameters. Secondly, *STML* searches a runtime SRAM partition strategy that reallocates memory between the secure world and normal world during world switches to minimize the task execution delay. At runtime, the IP protection mechanism efficiently protects the DL model IP by running the DL model within a TEE and dynamically allocating SRAM, following the memory allocation strategy obtained from the offline optimization engine.

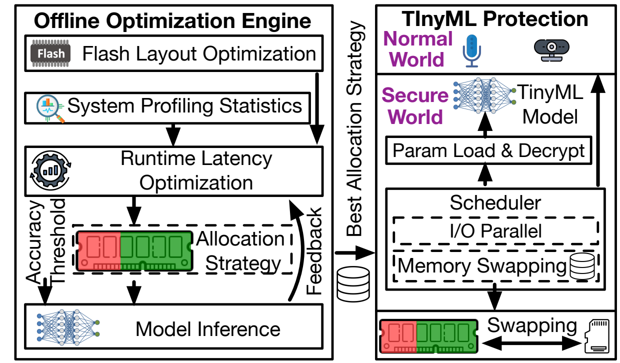


Fig. 1. Overview of *STML*.

B. Runtime IP Protection Mechanism

The MCU is equipped with R KiB of SRAM, I KiB of internal flash memory, and E KiB of external flash memory. After enabling TrustZone, with the configuration of Implementation Defined Attribution Unit (IDAU), the maximum SRAM and flash memory size that can be allocated to the normal world are $a \cdot R$ KiB ($0 < a < 1$) and $b \cdot I$ KiB ($0 < b < 1$), respectively. The MCU runs one TinyML task t_0 and k other tasks $\mathbf{T} = \{t_1, t_2, \ldots, t_k\}$. To prevent attackers from reading DL model details in SRAM during DL model execution, *STML* isolates DL model execution from untrusted software, i.e., DL model t_0 runs in the secure world and other tasks run in the normal world. However, memory isolation exacerbates on-chip memory (internal flash and SRAM) constraints. To tackle the internal flash constraint challenge, part of the model parameters are stored in external flash memory, e.g., SD card. To prevent attackers from stealing model details from external flash memory, the corresponding parameters are encrypted before being stored. Before using these parameters, they are loaded from the external flash memory to SRAM in the secure world and subsequently decrypted. To tackle the SRAM shortage problem, *STML* employs two strategies: reducing the SRAM space needed for DL model execution and allocating more available SRAM space for task execution. Specifically, *STML* executes the DL model layer-by-layer to reduce SRAM requirement. Furthermore, *STML* uses memory swapping that dynamically swaps memory contents between SRAM in both worlds and the external flash memory temporarily, providing additional SRAM space for the upcoming task execution.

Authorized licensed use limited to: WASHINGTON UNIVERSITY LIBRARIES. Downloaded on May 10,2025 at 21:32:31 UTC from IEEE Xplore. Restrictions apply.

The memory reallocation is guided by an optimal strategy to minimize the latency of DL model execution.

C. Flash Memory Allocation Optimization

Assume the tasks in the normal world need I_N KiB ($I_N \leq b \cdot I$) internal flash memory. Then, there is $I - I_N$ KiB internal flash memory left for TinyML task t_0 in the secure world. In certain scenarios, the TinyML model may be designed with a size that exceeds $I - I_N$ KiB. Thus, the parameters of some layers need to be stored in the external flash memory. To optimize internal flash memory usage to reduce memory swapping, *STML* formulates an optimization problem and employs a dynamic programming solution for determining a layer parameter storage strategy. Specifically, *STML* selects layers $\mathbf{l}_{in}$ for storing in the internal memory by

$$\mathbf{l}_{in}^* = \underset{\mathbf{l}_{in}}{\operatorname{argmax}} \sum_{l \in \mathbf{l}_{in}} \Gamma_l, \text{ s.t. } \sum_{l \in \mathbf{l}_{in}} \Gamma_l + I_C \leq I - I_N, \tag{1}$$

where Γ_l is the flash memory usage of the parameters of layer l, I_C is memory size of the code in the secure world. *STML* solves this optimization problem using dynamic programming. Subsequently, the selected layers are stored in the internal flash memory, while the remaining layers are placed in the external flash memory after encryption and decrypted prior to use.

D. Runtime Latency Optimization

The optimization goal is to minimize the TinyML task runtime latency, including model inference, TrustZone world switch, and encryption/decryption delays. *STML* focuses on predictable workload on MCU. Thus, we assumes that there are w times of world switch during a TinyML inference procedure. An array $\mathbf{C} = \{C_1, C_2, ..., C_w\}$ ($C_i \geq (1-a) \cdot I$) is used to represent the SRAM size allocated to the secure world during each world switch. The SRAM occupied by the executed tasks in either the secure world or normal world prior to the i-th world switch is represented by N_i, which forms the array $\mathbf{N} = \{N_0, N_1, ..., N_w\}$. The allocation of SRAM significantly affects the latency of model inference, world switch, and encryption/decryption. Thus, *STML* aims to minimize the latency $\mathcal{D}$ by finding the optimal $\mathbf{C}$ with

$$\mathbf{C}^* = \underset{\mathbf{C}}{\operatorname{argmin}}\, \mathcal{D}(\mathbf{C}) \text{ s.t. } \mathcal{D}(\mathbf{C}) = \mathcal{W}(\mathbf{C}) + \mathcal{F}(\mathbf{C}) + \mathcal{E}(\mathbf{C}) + \mathcal{P}, \tag{2}$$

where $\mathcal{W}$ represents the total world switch latency, $\mathcal{F}$ represents the total model inference latency, and $\mathcal{E}$ is the total encryption/decryption latency. Additionally, $\mathcal{P}$ represents latencies that are not optimized, including I/O latency and decryption latency for layers stored in the external flash memory. Note that the allocated SRAM size is always greater than or equal to the occupied SRAM of the running task during world switch in the both secure and normal world. For example, in the i-th world switch from the secure world to the normal world, it holds that $N_i \leq C_{i-1}$.

World Switch. Suppose in the i-th world switch, the secure world will be switched to the normal world. Also, after that world switch, there will be p tasks running in the normal world, which requires $\mathbf{E} = \{E_1, E_2, \ldots, E_p\}$ of SRAM to run the tasks. Therefore, when allocating SRAM for the normal world, the SRAM size should meet the requirement as

$$R - C_i \geq \max(\mathbf{E}). \tag{3}$$

During execution, the swapped memory size from the SRAM in the secure world to the external flash memory is $\max(N_i - C_i, 0)$ and from the external flash memory to the SRAM in the normal world is $\max(N_{i-1} + C_{i-1} - R, 0)$. Similarly, if the normal world is switched to the secure world in the i-the world switch, the swapped memory size from the SRAM in the normal world to the external flash memory is $\max(N_i + C_i - R, 0)$ while from the external flash memory to the SRAM in the secure world is $\max(N_{i-1} - C_{i-1}, 0)$.

The switching between the normal world and secure world happens alternatively, without losing generality, the total size of the swapped memory μ from the SRAM to the external flash memory can be represented as

$$\mu(\mathbf{C}) = \sum_{i=1}^{w} Z_i \cdot \max(N_i - C_i, 0) + (1 - Z_i) \cdot \max(N_i + C_i - R, 0), \tag{4}$$

where Z_i denotes the i-th world switch from the secure world to the normal world (value is 1) or from the normal world to the secure world (value is 0). The total size of the swapped memory δ from the external flash memory to the SRAM is

$$\delta(\mathbf{C}) = \sum_{i=1}^{w} Z_i \cdot \max(N_{i-1} + C_{i-1} - R, 0) + (1 - Z_i) \cdot \max(N_{i-1} - C_{i-1}, 0). \tag{5}$$

With the size of the swapped memory, the world switch latency can be represented as

$$\mathcal{W}(\mathbf{C}) = P \cdot \mu(\mathbf{C}) + P' \cdot \delta(\mathbf{C}) + Q, \tag{6}$$

where P and P' are the read and write speeds of the external flash memory, and Q is a constant number representing all other latency during the world switch.

Encryption/Decryption Operations. To prevent attackers from stealing model details from external flash, the memory data in the secure world is encrypted or decrypted when it is saved into or restored from external flash during memory swapping. Therefore, for the data swapped from the SRAM in the secure world to the external flash memory, the encryption latency and the data size are linearly positively correlated. The total encryption latency, $\mathcal{E}^e$, and decryption latency, $\mathcal{E}^d$, for the swapped data in the secure world are

$$\begin{aligned} \mathcal{E}^e(\mathbf{C}) &= \textstyle\sum_{i=1}^{w} G \cdot Z_i \cdot \max(N_i - C_i, 0) + V, \\ \mathcal{E}^d(\mathbf{C}) &= \textstyle\sum_{i=1}^{w} G' \cdot Z_i \cdot \max(N_{i-1} - C_{i-1}, 0) + V', \end{aligned} \tag{7}$$

where G and G' are coefficients related to the complexity of encryption and decryption algorithms, V and V' are constant numbers representing other latency during encryption and decryption. Thus, the total encryption/decryption latency is

$$\mathcal{E}(\mathbf{C}) = \mathcal{E}^e(\mathbf{C}) + \mathcal{E}^d(\mathbf{C}). \tag{8}$$

Model Inference. *STML* utilizes algorithm-level model optimization techniques [21] to minimize the model inference latency. Meanwhile, suppose $\mathbf{S} = \{S_2, S_4, \cdots, S_{\lfloor w/2 \rfloor}\}$ is the SRAM requirement for model execution every time switching

Authorized licensed use limited to: WASHINGTON UNIVERSITY LIBRARIES. Downloaded on May 10,2025 at 21:32:31 UTC from IEEE Xplore. Restrictions apply.

to the secure world, the SRAM usage of the model execution needs to satisfy for any i, $S_i \leq C_i$. For model optimization, depthwise separable convolution is employed to replace the standard convolution of those layers whose SRAM usage exceeds the limit by depthwise convolution and pointwise convolution. The computation of the two convolutions can be separated and a reduction in the computation can be achieved.

Furthermore, the width multiplier and resolution multiplier can be applied to decrease the total amount of computations. Specifically, for the width multiplier, a parameter $\alpha \in (0, 1]$ is applied to scale the original input and output channel dimension, respectively. For the resolution multiplier, a parameter $\beta \in (0, 1]$ is utilized to scale the resolution of input representation. Since both scaling operations will have negative impacts on the model accuracy, the parameters need to be searched to meet both the SRAM requirement and the accuracy requirement. Formally, the optimization objective is

$$\mathcal{F}^*(\mathbf{C}) = \min_{\alpha,\beta} \mathcal{F}(\mathcal{M}(\alpha, \beta)) \text{ s.t. } \mathcal{A} \geq \eta; \forall i, S_i \leq C_i. \quad (9)$$

where $\mathcal{M}$ represents the model, $\mathcal{A}$ represents the performance metric, and η is the requirement of the model accuracy. By applying a grid search strategy, *STML* can find a combination of α and β to minimize the model inference latency under the constraints of $\mathbf{C}$ and η. Also, the performance metric $\mathcal{A}$ under different pairs of α and β can be evaluated after re-training. Note that while model optimization may alter the model flash memory requirement, the original model size is used in flash memory allocation optimization (Section IV-C) to limit the search space for the whole system.

Optimization Solution. By combining Eq. 6, Eq. 9, and Eq. 8 with Eq. 2, we can get an explicit expression between the latency $\mathcal{D}$ and SRAM allocation strategy $\mathbf{C}$. Since it is a high-dimensional continuous search problem, a genetic algorithm is adopted to search the optimal SRAM allocation schemes and the latency $\mathcal{D}$. Genetic algorithms have been demonstrated effective for searching the optimized parameters in many areas such as resource scheduling [22]. Specifically, a population of candidate arrays $\mathbf{C}$ are evolved to maximize the fitness score $-\mathcal{D}$. In each iteration, candidates with the highest score are selected while the rest are dropped. The selected candidates undergo crossover, where new candidates are generated by sampling values from a pair of parent candidates, with the sampling probability proportional to their fitness values. Mutation is performed on the new candidates by adding Gaussian distributed vectors with a certain mutation probability. The detailed algorithm is provided in Algorithm 1.

Algorithm 1: Genetic-based Optimization Algorithm

```
1  Input: The total number of population K_n, selected population K_s, mutation
       probability p_m, and the maximum number of iterations c_max.
2  Output: Secure world SRAM array C.
3  𝒞 ← {C_0, C_1, ..., C_{K_s}}, P ← 𝒟(𝒞)
4  for c = 0 to c_max do
5      for i = 1 to K_n do
6          C_i1, C_i2 ← Sample(𝒞, P)
7          C_i ← Crossover(C_i1, C_i2)
8          C_i ← Mutate(C_i, p_m)
9          𝒞 ← 𝒞 ∪ {C_i}
10     end
11     P ← 𝒟(𝒞)
12     𝒞, P ← Sort(𝒞, P)[: K_s]
13 end
14 return 𝒞(0)
```

Computation and I/O Parallelism. The layer parameter loading from the external flash memory and the layer computation can be executed in parallel once direct memory access (DMA) is enabled, such that the idle time of CPU and I/O can be further reduced. Specifically, consider executing two layers sequentially, where the parameters of the latter layer are stored in external flash memory. After profiling the SRAM usage of the first layer, if there is sufficient free SRAM in the secure world to load the parameters of the second layer, the decryption and computation of the first layer and the parameter loading of the second layer will be performed in parallel. This approach allows the execution of the second layer to start immediately after finishing the first layer, as the second layer's parameters have already been loaded in advance. As a result, the idle time for both CPU and I/O is shortened, leading to a further reduction in the TinyML task execution latency $\mathcal{D}$.

V. Evaluation

A. Experimental Settings

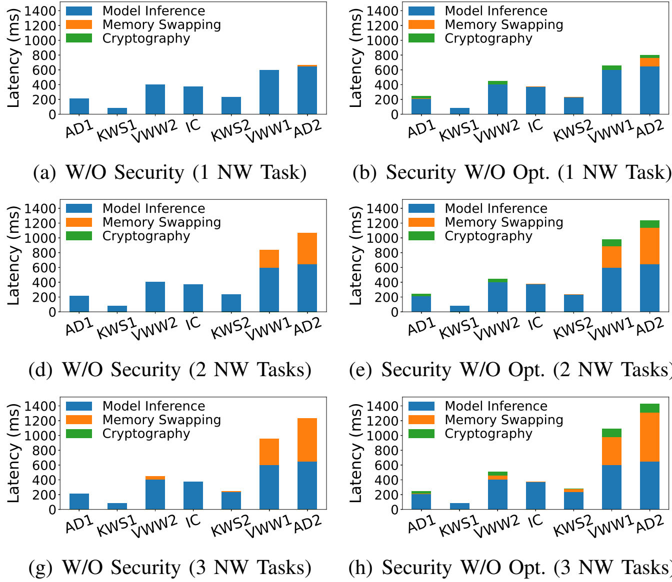
Implementation Details. The *STML* prototype[1] has been developed and evaluated using STM32L562E-DK, an MCU kit featuring a Cortex-M33 CPU with TrustZone, 512 KiB of flash memory, 256 KiB of SRAM, and a 32 GiB SD card. We implemented the layer-by-layer model execution and per-layer parameter loading from the SD card based on TensorFlow Lite Micro [23], a TinyML framework designed for MCUs. The reconfiguration of memory security attributes and memory content swapping were implemented in the context switch routine of the real-time operating system (RTOS) scheduler. The Security Attribution Unit (SAU) was modified to change the memory security state. Additionally, a hardware-assisted cryptographic library was employed to speed up cryptographic operations during parameter loading or saving and memory content swapping. Offline optimization was executed on a PC.

Key Performance Questions. We evaluated *STML* under different scenarios to answer the following three questions:
Q1. *How much runtime overhead is introduced by memory swapping (i.e., enabling large model execution) and security mechanism (i.e., protecting DL model IP) for TinyML? (§V-B)*
Q2. *Can STML significantly reduce the runtime overhead of TinyML tasks while ensuring the security of the tasks? (§V-B)*
Q3. *How do system-level optimizations and algorithm-level optimizations affect model execution latency? (§V-C)*

TinyML Benchmark Tasks and Models. To evaluate the performance of *STML* on a range of TinyML tasks using various trained models, we evaluated it with models from MLPerf Tiny Benchmark [24] and MicroNets [25]. The benchmark TinyML tasks include keyword spotting (KWS), anomaly

[1]Source code is available at https://github.com/WUSTL-CSPL/TinyML.

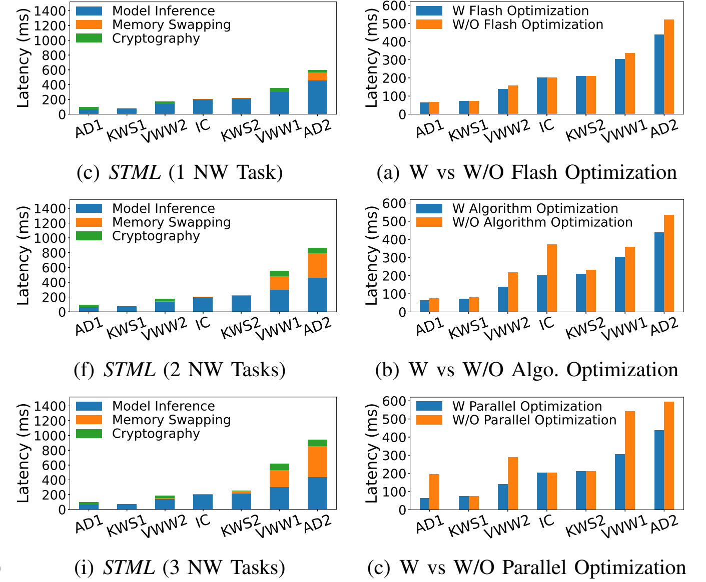
Authorized licensed use limited to: WASHINGTON UNIVERSITY LIBRARIES. Downloaded on May 10,2025 at 21:32:31 UTC from IEEE Xplore. Restrictions apply.



(a) W/O Security (1 NW Task) (b) Security W/O Opt. (1 NW Task)

(d) W/O Security (2 NW Tasks) (e) Security W/O Opt. (2 NW Tasks)

(g) W/O Security (3 NW Tasks) (h) Security W/O Opt. (3 NW Tasks)



(c) *STML* (1 NW Task) (a) W vs W/O Flash Optimization

(f) *STML* (2 NW Tasks) (b) W vs W/O Algo. Optimization

(i) *STML* (3 NW Tasks) (c) W vs W/O Parallel Optimization

Fig. 2. TinyML Task Execution Latency.

Fig. 3. Optimization Effectiveness.

TABLE I
METADATA OF TINYML BENCHMARK MODELS

Tasks	Model	Flash(KB)	RAM(KB)	Latency(ms)	Metric
KWS	(1)DS-CNN [15]	144.80	31.25	81.29	90% (Top-1)
	(2)MicroNet-KWS(S) [24]	192.55	70.14	233.05	
AD	(1)Deep AutoEncoder [15]	328.96	10.57	7.64	0.85 (AUC)
	(2)MicroNet-AD(S) [24]	327.64	120.14	445.66	
VWW	(1)MobileNetV1 0.25x [15]	420.07	108.82	256.68	80% (Top-1)
	(2)MicroNet-VWW(S) [24]	363.60	77.71	146.01	
IC	(1)ResNet-8 [15]	187.04	62.32	373.33	85% (Top-1)

TABLE II
METADATA OF TASKS IN THE NORMAL WORLD

Tasks	Flash (KB)	SRAM (KB)	Frequency (Hz)	Priority
LED Toggle	56.90	25.14	50	2
Logging	89.70	41.34	20	3
AudioSample	170.94	114.35	10	1

detection (AD), visual wake words detection (VWW), and image classification (IC). Table I shows measurement data of the used models when all system resources are available on the MCU. We ensured our algorithm-level model optimization adhered to the performance requirements and quality targets specified by MLPerf Tiny Benchmark.

Tasks in the System. A TinyML task runs in the secure world, while other tasks including LED Toggling, Logging, and AudioSampling, are executed in the normal world. The metadata of these tasks is illustrated in Table II. Note that the flash size of AudioSample includes both the code size in the internal flash memory the data size in the SD card. Similar to tasks in widely-used cyber-physical systems like ArduPilot [26], these normal world tasks have higher execution priorities, as they are responsible for critical operations.

B. Optimization Results

Performance Overhead Profiling. To fully comprehend the impact of security mechanisms on performance, we measured the DL execution latency in three different system settings: a system without TEE protection (W/O Security), a system that protects model IP using fixed memory allocation (half for each world) between the secure world and normal world (Security W/O Opt.), and a system that optimizes the protection of model IP by dynamically changing memory allocation at runtime (*STML*). We assessed execution delay across various system workloads within each system setting, where one, two, and three high-priority tasks concurrently run (in the normal world if the model IP is protected) with the TinyML task.

Fig. 2 shows that deploying an IP protection security mechanism under the same workload leads to an increased DL execution delay due to memory isolation and cryptographic operations. The scarcity of memory resources is compounded by memory isolation, necessitating more frequent memory-swapping and fewer concurrently executed model layers. As the number of tasks increases within the same system configuration, the model execution delay also increases because model execution is more frequently preempted. Each time model execution is preempted, *STML* expends additional time switching task contexts, adding to the increased model execution delay.

Answer to Q1. If the entire model fits in the TEE, deploying TinyML there incurs minimal overhead (0.0026% on average). However, when the model size exceeds the allocated memory of TEE secure world, using a security mechanism with the default allocation leads to a 15% average runtime overhead, mainly due to memory swapping and cryptographic operations.

Effectiveness of *STML*. The runtime latency of *STML*, using optimal memory allocation and optimized TinyML models, is shown in Fig. 2(c),(f),(i) under varying workloads. This configuration exhibits reduced overhead from world switches, cryptography, and model inference compared to the unoptimized security mechanism. Model optimization significantly reduces memory usage (flash and SRAM) and inference la-

Authorized licensed use limited to: WASHINGTON UNIVERSITY LIBRARIES. Downloaded on May 10,2025 at 21:32:31 UTC from IEEE Xplore. Restrictions apply.

tency. The decreased SRAM requirement reduces swapped memory size, leading to lower latency for world switches and cryptographic operations. On average, the optimized security mechanism offers a 40% overhead reduction compared to its non-optimized counterpart, achieved through 42%, 35%, and 22% reductions in model inference, memory swapping, and cryptographic operation overheads, respectively.

Answer to Q2. With the proposed optimization, latency for world switches, cryptographic operations, and model inference within a security mechanism is significantly reduced.

C. Ablation Study

Ablation Study Setup. The optimization scheme consists of three key strategies: optimizing flash memory allocation, refining the model algorithm, and computation and I/O parallelism. To evaluate the impact of each strategy on model execution latency, we conduct an ablation study by removing one strategy at a time and measuring the resulting latency. This evaluation takes place under a workload setting with three normal world tasks, and the results are shown in Fig. 3.

Results. Without the flash memory allocation optimization, model inference latency increases 9.4%. While the most significant inference latency increase is seen in *MicroNet_AD* model, the inference latency of *MLPerf_KWS*, *MLPerf_IC*, and *MicroNet_KWS* remain the same since they are small enough to fit into the internal flash memory. From Fig. 3(b), after employing model algorithm optimization, a 30% average decrease in model inference latency can be achieved. In particular, for those models that contain more standard convolutional layers such as *MLPerf_IC* and *MicroNet_VWW*, the optimization effect is more obvious since more convolution calculations can be replaced by depthwise separable convolution operations. Additionally, computational and I/O parallel optimization have the most significant impact on model inference latency. When disabling optimizations, latency increases to 1.5 times the optimal value. Similar to the flash memory allocation optimization, it also has no effects on the smaller models yet a significant effects on the larger models including *MLPerf_AD*, *MLPerf_VWW*, and *MicroNet_VWW*.

Answer to Q3. Each optimization strategy notably reduces model execution latency. Computation and I/O parallelism have the largest impact, while model algorithm optimization applies to more TinyML tasks, given the prevalence of convolutional calculations in TinyML models.

VI. Conclusion and Discussion

In this paper, we introduce *STML*, a TinyML model IP protection system for MCUs utilizing ARM TrustZone. We propose a memory swapping scheme to address the limited memory issue and minimize I/O and inference latency through system and algorithm level optimization. Our approach effectively balances memory usage, latency, security, and accuracy, resulting in a 40% reduction in runtime overhead compared to non-optimized solutions. Although initially designed for systems with predictable workloads, *STML* can be adapted to other systems by adjusting the DL execution latency modeling to accommodate their specific characteristics.

In addition to minimizing DL execution delay, future work may investigate other key performance metrics, such as energy efficiency, to address the wide-ranging needs of various TinyML application scenarios. Although *STML* can protect TinyML IP against attackers targeting direct software memory access, defense against other attack vectors, such as side-channel attacks and cold boot attacks, still requires additional investigation. Given the growing importance of AI, developing security protection for its deployment on edge devices is an vital future research direction.

Acknowledgment

This work was partially supported by the US NSF (CNS-1916926, CNS-2229427, CNS-2238635), ARO (W911NF2010141) and DOE (DE-EE0009338).

References

[1] Y. LeCun *et al.*, “Deep learning,” *Nature*, 2015.
[2] P. Warden *et al.*, *TinyML*. O’Reilly Media, 2019.
[3] L. Batina *et al.*, “CSI NN: Reverse engineering of neural network architectures through electromagnetic side channel,” in *Security*, USENIX, 2019.
[4] M. Jagielski *et al.*, “High accuracy and high fidelity extraction of neural networks,” in *Security*, USENIX, 2020.
[5] H. Liu *et al.*, “When evil calls: Targeted adversarial voice over ip network,” in *CCS*, ACM, 2022.
[6] Z. Yu *et al.*, “Smack: Semantically meaningful adversarial audio attack,” in *Security*, USENIX, 2023.
[7] H. Liu *et al.*, “Slowlidar: Increasing the latency of lidar-based detection using adversarial examples,” in *CVPR*, IEEE, 2023.
[8] Oliynyk *et al.*, “I know what you trained last summer: A survey on stealing machine learning models and defences,” *arXiv*, 2022.
[9] F. Mo *et al.*, “DarkneTZ: towards model privacy at the edge using trusted execution environments,” in *MobiSys*, ACM, 2020.
[10] A. Gangal *et al.*, “HybridTEE: Secure mobile DNN execution using hybrid trusted execution environment,” in *AsianHOST*, IEEE, 2020.
[11] S. P. Bayerl *et al.*, “Offline model guard: Secure and private ML on mobile devices,” in *DATE*, IEEE, 2020.
[12] Z. Sun *et al.*, “Shadownet: A secure and efficient on-device model inference system for convolutional neural networks,” in *S&P*, IEEE, 2023.
[13] L. Hanzlik *et al.*, “MLCapsule: Guarded offline deployment of machine learning as a service,” in *CVPR Workshop*, IEEE, 2021.
[14] J. Lin *et al.*, “Mcunet: Tiny deep learning on iot devices,” in *NeurIPS*, PMLR, 2020.
[15] C. R. Banbury *et al.*, “Benchmarking TinyML systems: Challenges and direction,” *arXiv*, 2020.
[16] S. Han *et al.*, “Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding,” *ICLR*, 2015.
[17] H. Miao *et al.*, “Towards out-of-core neural networks on microcontrollers,” in *SEC*, IEEE, 2022.
[18] J. Wang *et al.*, “Rt-tee: Real-time system availability for cyber-physical systems using arm trustzone,” in *S&P*, IEEE, 2022.
[19] J. Wang *et al.*, “ARI: Attestation of Real-time Mission Execution Integrity,” in *Security*, USENIX, 2023.
[20] ARM, “TrustZone security,” 2009.
[21] A. G. Howard *et al.*, “Mobilenets: Efficient convolutional neural networks for mobile vision applications,” *arXiv*, 2017.
[22] Z. Zhou *et al.*, “An improved genetic algorithm using greedy strategy toward task scheduling optimization in cloud environments,” *Neural Comput. Appl.*, 2020.
[23] R. David *et al.*, “Tensorflow lite micro: Embedded machine learning for tinyml systems,” *MLSys*, 2021.
[24] C. Banbury *et al.*, “MLPerf tiny benchmark,” *arXiv*, 2021.
[25] C. Banbury *et al.*, “Micronets: Neural network architectures for deploying tinyml applications on commodity microcontrollers,” *MLSys*, 2021.
[26] “Ardupilot.” https://ardupilot.org/.

Authorized licensed use limited to: WASHINGTON UNIVERSITY LIBRARIES. Downloaded on May 10,2025 at 21:32:31 UTC from IEEE Xplore. Restrictions apply.