Typhon Unleashed: Practical Adversarial Weight Attacks against On-Device Deep Learning Models

Yujin Huang, Xingliang Yuan, Chunyang Chen, Seong Oun Hwang

***Abstract*—On-device deep learning (DL) has emerged as a popular approach for mobile apps to deliver artificial intelligence services. Unlike traditional cloud-based approaches, it processes sensitive information locally, addressing severe concerns over sensitive data collection on cloud servers. However, this approach inevitably stores models on user devices and opens a new attack surface, i.e., adversarial weight attacks, which steer DL models to undesirable behaviors through direct model weight modification. Unfortunately, such risks stemming from on-device DL have been left unexplored.**

In this paper, we present the first practical adversarial weight attack against on-device DL models. To demonstrate this novel attack, we propose TYPHON, an automated attack system that removes the read-only restriction of on-device DL models through the reconstruction of writable counterparts and leverages the inference-only nature of on-device DL models to solve malicious parameters and manipulate model behaviors. Extensive experimental results across diverse datasets and model architectures confirm the superiority of our attack across multiple evaluation metrics. In addition, three real-world case studies are conducted with 100% attack success rates, demonstrating the practicality of TYPHON.

***Index Terms*—On-device deep learning, adversarial weight attack, mobile application security.**

I. INTRODUCTION

Recent trends in mobile apps show a shift towards adopting on-device deep learning (DL) instead of relying on the cloud to provide artificial intelligence services [1]. For instance, a DL-powered skin cancer recognition app can assist individuals in identifying skin cancer types and advancing early detection [2]. Primary advantages of this approach include enhanced privacy as sensitive data is processed locally on the smartphone, faster performance as it eliminates dependence on network quality, and broader applicability, even in scenarios without network connectivity [3]–[6].

Despite the manifold benefits, on-device DL necessitates the local storage of DL models on user devices. This naturally enables attackers to disassemble the deep learning mobile apps (DL apps) to obtain the models for malicious exploits. Specifically, we observe that a known adversarial attack against DL models [7]–[17], named adversarial weight attack, is underexplored in on-device DL models. With direct access to the model, such an attack can induce model misbehavior via direct model weight modification. For example, if the attacker can manipulate the predicted results of those DL apps engaged in life-critical tasks by maliciously modifying their model weights, e.g., altering "*Melanoma and see a dermatologist*" into "*Healthy Skin*", end-users' lives could be jeopardized, as shown in Figure 1.

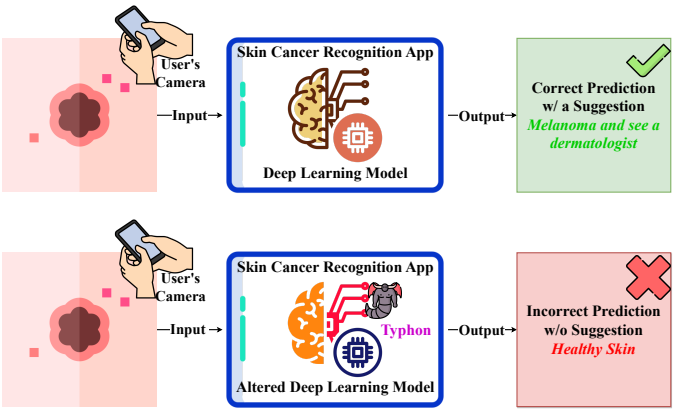


Fig. 1: TYPHON attack. The attacker compromises the skin cancer recognition app by modifying model weights to induce malicious behaviors like misclassification.

To take the first step in bridging this gap, this paper strives to demonstrate the feasibility of exploiting this attack paradigm to instantiate an adversarial weight attack against on-device models. While a few prior works explore on-device model attacks, they have either concentrated on stealing models [4] or attacking models within certain constraints. For example, some assume the compromise of users' smartphone cameras to introduce adversarial perturbations into inputs [3], [5], while others necessitate the transmission of malicious inputs to target users' smartphones [18], [19].

Compared to those existing attacks, mounting an attack predicated on model weight against on-device models presents a more practical attack surface. Since this attack is anchored on modifying the weights of an on-device model in line with the attacker's goal, without intervening in user inputs, it (1) sidesteps the imperative to interfere with users' smartphone cameras and (2) obviates the need to send attacker-crafted inputs to user devices. Additionally, the attack rooted in model weight modification provides the attacker the latitude to realize diverse goals, such as performance degradation and backdoor effect. More importantly, this attack circumvents the necessity to compromise users' smartphones for input manipulation, rendering it stealthier.

Yujin Huang and Xingliang Yuan are with the School of Computing and Information Systems, The University of Melbourne, Australia. Email: yujinh@student.unimelb.edu.au, xingliang.yuan@unimelb.edu.au.

Chunyang Chen is with the Department of Computer Science, Technical University of Munich, Germany. Email: chun-yang.chen@tum.de.

Seong Oun Hwang is with the Department of Computer Engineering, Gachon University, Korea. Email: sohwang@gachon.ac.kr.

In this paper, we present TYPHON, the first automated system that demonstrates the feasibility of the practical adversarial weight attack against on-device models from DL apps. Specifically, TYPHON accounts for the specific characteristics (i.e., *read-only and inference-only with backpropagation disabled*) of the on-device model and compromises it through four steps: First, given a DL app, it extracts an on-device model by disassembling the app. Then, it enables parameter modification of the on-device model as the successful accomplishment of any attack goal necessitates the ability to alter model parameters. Next, it employs a training-free approach to compute malicious parameters for the writable on-device model based on the specified attack goals and rewrites these parameters back to the model. Finally, it uses the compromised model to rebuild the DL app, generating a malicious counterpart.

TYPHON involves three key challenges. First, certain on-device models are stored in encrypted form, which prevents direct extraction. Although prior work [20] demonstrated the feasibility of extracting such models from memory, its limited instrumentation strategies result in incomplete and fragmented extraction. To address this, we advance their approach by introducing an execution tracking mechanism that systematically monitors API invocations during DL app execution, identifies those associated with encrypted model usage, captures relevant in-memory data objects, and refines extracted models by incorporating values from these objects when they align with model metadata.

The second challenge is enabling writability for on-device models, as their inherent *read-only* design restricts modification. Through analysis of the file format used for on-device models, we find that all model information, including operators and parameters, is serialized according to a predefined model schema, making this information immutable. To tackle this, we propose Model Unsealing, an on-device model reverse engineering technique that reconstructs a writable model from the original by exploiting the model schema. Specifically, we generate programming language classes corresponding to data structures defined in the model schema to deserialize information from an on-device model. Although the model information is retrievable, the generated classes derived from the schema do not support serialization. To address this problem, we improve these classes with specialized serialization functions that allow seamless reconstruction of a writable model.

As on-device models are *inference-only with backpropagation disabled*, attacking them in a training-free manner presents another challenge. Expressly, identifying the parameters to modify and determining the optimal magnitude of changes are challenging without iterative optimization. Additionally, some on-device models perform rare recognition tasks, making relevant data collection from the Internet unrealistic and the attack more challenging. To address these challenges, we introduce a data synthesis approach that is capable of generating data for any on-device model by leveraging a public dataset. Then, we propose a training-free parameter optimization method, Model Twisting, which exploits the feed-forward process of an on-device model to solve its malicious parameters aligned with the attacker's goals.

We evaluate TYPHON on four computer vision datasets using three widely used on-device DL models, alongside comprehensive comparisons with two latest adversarial weight attacks. Following the previous studies [5], [18], we use metrics that measures Performance Drop Rate, Backdoor Success Rate and Clean Test Accuracy. Besides, we also show the attack efficiency using Attack Execution Time. Experimental results show that TYPHON remains highly effective in different settings, with significant margin in terms of all evaluation metrics than other considered advanced attacks. To further assess the impact of TYPHON on real-world DL apps, we present three case studies using apps that perform security-critical tasks, including skin cancer recognition, traffic sign recognition, and currency recognition. The results demonstrate that TYPHON successfully compromises all apps, inducing attacker-desired behaviors.

Contributions. We summarize the contributions as follows:

- We present a novel on-device model attack that enables direct modification to the on-device model and leverages its feed-forward process to solve malicious parameters to compromise model behavior.
- We propose and implement a groundbreaking automated attack system, TYPHON, to demonstrate the feasibility of the new on-device model attack and address limitations encountered in prior on-device model attacks. Our prototype introduces a series of novel mechanisms, including execution tracking for complementing partially extracted encrypted models, Model Unsealing for constructing writable models, and Model Twisting for solving malicious model parameters in a training-free manner.
- We conduct an extensive evaluation to demonstrate the effectiveness and efficiency of TYPHON. The results indicate that it achieves attack with a high success rate and negotiable utility drop with short execution time in various attack scenarios.

II. BACKGROUND

A. On-device DL

On-device DL aims to construct and deploy DL models on edge devices like smartphone to perform efficient inference. Compared to offloading DL from edge devices to the cloud, on-device DL enables real-time inference, preserves user privacy, and maintains functionality even in the absence of network connectivity. Thus, efficient implementation of on-device DL are essential to harness such benefits. Numerous open-source and proprietary frameworks have been developed for this purpose, including Google TensorFlow [21] and TensorFlow Lite (TFLite) [22], Facebook PyTorch [23] and PyTorch Mobile [24], and Apple Core ML [25]. Among these frameworks, TFLite is the most widely used for DL models on smartphones due to its GPU support and optimization for mobile devices [1], [3], [5].

To develop and deploy a TFLite model, a mobile app developer typically begins by collecting data and training a TensorFlow model on a cloud platform or high-performance desktop server. After completing the training, the developer

utilizes the TFLite Converter[1] to convert the TensorFlow model into TFLite one that is optimized for efficiency and portability through the use of FlatBuffers[2]. Notably, the resultant TFLite cannot be modified and converted back to the original TensorFlow model supporting backpropagation, that is, the TFLite model is strictly *read-only* and *inference-only*. Google imposes such constraints to prevent potential attacks on on-device DL models, as they are inevitably stored on user local devices and could be accessed by attackers. Finally, the developer incorporates the TFLite model into a mobile app and compiles it as an Android Application Package (APK) for distribution through app markets.

B. Adversarial Weight Attack

The adversarial weight attack is a post-training attack. It steers deployed DL models to undesirable behaviors by slightly modifying the model weights [7], [8], [15]. Expressly, to mount such an attack, the attacker possesses write access to the target model file. In addition, the attacker has a set of benign samples to compute the malicious weights used in the manipulation of the target model. The derived malicious weights are subsequently written back to the target model, with the end result being that the compromised model produces the attacker-desired label for the target benign inputs (i.e., adversarial inputs) during the attack, while correctly predicting other benign inputs. Note that the adversarial inputs here refer to the inputs of adversarial weighted attacks, which differ from those in traditional adversarial attacks [26], [27].

Based on the presence or absence of an added trigger within the target benign inputs, existing adversarial weight attacks can be categorized into two types: non-trigger-based and trigger-based attacks [7], [8], [15]. The former solely modifies the model weights to induce the misclassifications on the target benign inputs, while the latter embeds triggers into the target benign inputs and then modifies the model weights to accommodate the triggers, thereby altering the model predictions accordingly.

III. Threat Model and Challenges

A. Threat Model

We consider an attacker as a malicious third-party who can download benign DL apps from Android markets. In this paper, we target Google Play. The attacker has no prerequisite domain knowledge of the DL apps, nor physical access (e.g., smartphone camera) or root privileges (e.g., bootloader unlocking) on user devices. This is realistic in practice because mobile apps are publicly accessible and avoiding device-level control makes attacks more stealthy and persistent.

After downloading a target DL app, the attacker aims to maliciously modify the DL model in a way that it presents attacker-desired behaviors, such as performance degradation, backdoor effect or both, upon attacker-crafted inputs. Take the skin cancer recognition app as an example, an attacker-desired behavior can be tricking the app into misclassifying a melanoma as any other cancer type (i.e., performance degradation) or predicting a melanoma embedded with an attacker-specific trigger as healthy skin (i.e., backdoor effect).

After that, the attacker substitutes the modified model for the original one in the DL app, and redistributes the app to alternative app markets[3] (e.g., APKMirror [28], APKPure [29] and F-droid [30]) for public download. The attacker may also attempt to upload it to Google Play by employing app-virtualization repackaging that materializes and executes malicious components covertly to circumvent malware detection mechanisms like Google Play Protect [31]–[33]. This can badly infect a number of app users, because current countermeasures against Android malware are still not efficient enough [34]–[36]. After the redistribution, the attacker has no control over the devices of infected users such as cameras and does not require manipulating user inputs to the app.

B. Challenges

As aforementioned, the attacker aims to maliciously modify an on-device DL model within a DL app to induce attacker-desired behaviors, such as performance degradation or backdoor effect, without controlling user devices. This threat scenario poses three main research challenges that we depict in the following:

C1-Extraction of Encrypted On-Device Models. To achieve the attacker's goal, the initial step is to extract the on-device model from a DL app. For security reasons, certain on-device models are stored in encrypted forms to prevent unauthorized manipulation. Consequently, the first challenge is to extract encrypted on-device models in plaintext form. While direct extraction of such models from memory may seem straightforward, this approach proves ineffective as shown in Section IV-B. Hence, the development of an advanced method to extract encrypted on-device models is challenging.

C2-Modification of Read-Only On-Device Models. After extracting the on-device DL model, the next challenge is enabling model modification, as the attacker aims to alter the on-device model to achieve malicious goals. The key difficulty in this step lies in the *read-only* nature of on-device models, which prevents any modification to their structures and parameters. This is because on-device DL framework vendors like Google recognized that local storage of on-device models could invite malicious attacks, prompting them to impose strict security constraints. Thus, enabling modification of on-device models is far from trivial, and to the best of our knowledge, no prior work in the literature has attempted to address this challenge.

C3-Malicious Parameters Solving of Inference-Only On-Device Models. Furthermore, even if model modification is enabled, another challenge emerges in solving the on-device model's parameters to achieve the attacker's goal. The reason for choosing parameter modification instead of structure modification is that it makes the attack more stealthy. Structural modification often requires the insertion of malicious layers or subnets into the on-device model, resulting in changes to the

[1]https://www.tensorflow.org/api_docs/python/tf/lite/TFLiteConverter
[2]https://google.github.io/flatbuffers

[3]This is validated by successfully uploading the TYPHON-produced APKs to APKMirror and APKPure.

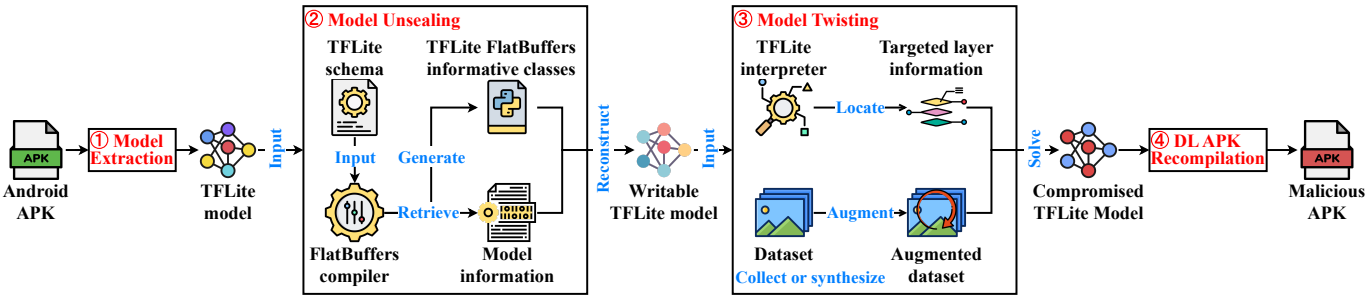


Fig. 2: Overview of TYPHON.

model size that render it susceptible to detection. While parameter modification may seem simple, on-device models are *inference-only*, making it extremely difficult to determine the optimal magnitude of parameter changes. This arises because these models lack gradient information and cannot perform iterative optimization. Consequently, devising an optimization-free approach to effectively solve malicious parameters for on-device models presents a significant challenge.

Remark. The on-device models here refer to TFLite models, which are the focus of this work due to their significant prevalence in DL apps and increasing adoption [1], [3].

IV. TYPHON

A. Overview

Figure 2 presents the overview of TYPHON, which aims to address the aforementioned challenges so that an on-device model exhibits attacker-desired behaviors. ① TYPHON first conducts model extraction on a DL app to obtain its encrypted or plaintext on-device model that is *read-only* and *inference-only*. ② Then, it performs Model Unsealing to generate a set of model informative classes, retrieve the on-device model information, and reconstruct a writable on-device model based on them. ③ Next, the writable on-device model is processed to craft a compromised one via Model Twisting, where targeted (attacked) layers are determined based on the model structure, and the dataset for solving malicious parameters at the target layers is either collected from the Internet or synthesized, depending on the availability of the model's label file. ④ Finally, TYPHON executes DL APK Recompilation on the DL app to replace the original on-device model with the compromised one to produce a malicious DL app that can be redistributed through various app markets, waiting for victim users to download.

B. Model Extraction

To obtain on-device models from a DL app, we first examine all files within the app to locate the models and then extract them based on their types, either plaintext or encrypted. Expressly, given an Android APK adopting TFLite models, we initially employ the well-known reverse engineering tool, Apktool [37], to decompile the APK. We then search the decompiled folder for files that follow TFLite naming schemes [4] and extract them if they are in plaintext. In cases where the models are encrypted, we instrument the app to dynamically extract them from memory.

However, in a pilot study on several real-world DL apps using encrypted models, we find that most models extracted via app instrumentation are unusable due to missing or incomplete metadata, including input/output descriptions and essential pre/post-processing details. To address this issue, we propose a model execution tracking method to complement the extracted encrypted models with deficient metadata. In particular, we hook the code responsible for model loading and inference, execute the app to perform DL functions, and capture all model-related data objects exchanged during execution. Subsequently, we search the captured objects for TFLite metadata (i,e., ModelMetadata, SubGraphMetadata, and TensorMetadata), and upon identification, extract and integrate the pertinent values into the encrypted models for refinement.

After model extraction, we load each model into memory and run inference on randomly generated data to verify model completeness and usability. This is critical for our attack as we necessitate a complete model to reconstruct its writable counterpart and rely on model inference to solve its malicious parameters.

C. Model Unsealing

Once we extract the on-device models, we need to make these *read-only* models writable to modify their parameters and achieve attacker-desired behaviors at a later stage. To do so, we propose a TFLite reverse engineering approach, namely Model Unsealing, to reconstruct a writable on-device model in accordance with the original one. The workflow of Model Unsealing is depicted in Figure 3. It is divided into three phases to render an on-device model writable as follows.

P1-Model Parsing Class Generation. The extracted TFLite model is represented in an efficient portable format known as FlatBuffers[4], where data (e.g., operators and parameters) are serialized according to a predefined Interface Definition Language-based schema and are *strictly read-only*. Therefore, to render the extracted model writable, we need to deserialize all contained data for later use in reconstructing a writable counterpart. However, TFLite does not offer APIs for this operation due to safety concerns. To address this challenge, we generate a set of programming classes called model parsing classes based on the official TFLite model schema[5]. These classes enable us to read and interpret the serialized data in the extracted model through their concrete objects and can be

[4]https://google.github.io/flatbuffers
[5]https://github.com/tensorflow/tensorflow/blob/master/tensorflow/lite/schema

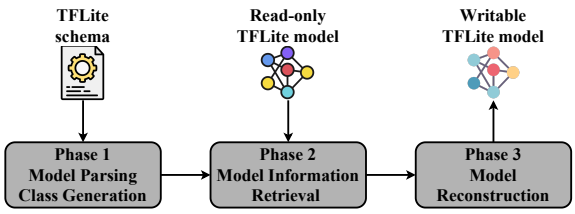


Fig. 3: Workflow of Model Unsealing.

utilized to make the model's data writable at a later phase. An example of model parsing class is shown in Figure 4-Phase 1.

P2-Model Data Retrieval. When retrieving the serialized data from the extracted model, we observe that the data are ill-organized as they are read in a holistic and unordered manner through the field-matching methods of all data structures employed by the model. This makes the reconstruction of a writable counterpart inconvenient and error-prone given the model's multitude of operators and parameters. To solve this problem, we design and implement a suite of auxiliary classes linked to the model parsing classes, which provide a systematic way to retrieve model data such that each data structure's values are accessed separately and in an organized manner. An example of auxiliary class is shown in Figure 4-Phase 2.

P3-Model Reconstruction. Having obtained the organized data via the instances of auxiliary classes corresponding to the extracted model, we proceed to leverage them to reconstruct a writable counterpart. In doing so, we find that these instances lack data serialization capabilities, which are essential for a writable model as reserialization of data is required after modifications. To address this limitation, we enhance the auxiliary classes by incorporating data serialization functions. With the enhanced auxiliary class instances, we form a writable model that encapsulates extracted model data and supports serialization. An example of enhanced auxiliary class with serialization functions is shown in Figure 4-Phase 3.

D. Model Twisting

After obtaining the writable extracted model, we possess the ability to modify its parameters to achieve malicious goals. However, as the writable model is *inference-only* with backpropagation disabled, determining the optimal magnitude of parameter modification (i.e., achieving strong performance while being stealth) becomes non-trivial when iterative optimization is infeasible. Therefore, to overcome this challenge, we propose a training-free parameter optimization approach called Model Twisting, which leverages feed-forward process to solve malicious parameters aligned with the attacker's goal. Expressly, Model Twisting is characterized along two dimensions:

- **Objective $\mathcal{O}$.** This dimension characterizes the attacker's goal, which is to degrade the performance of an on-device model or to make it yield a backdoor effect while maintaining model utility.
- **Label $\mathcal{L}$.** This dimension characterizes the attacker's access to the inference data of an on-device model. A predicted label file of the on-device model provides the attacker with information on model training data. However, in certain real-world DL apps, their label files are missing or the training data they contain are limited due to privacy concerns (e.g., DL apps for prostate cancer recognition). Thus, there are three cases regarding the availability of the on-device model's inference data: (1) the label file does not exist. (2) the label file exists but its data is scarce. (3) the label file exists and its data is sufficient.

Fig. 4: Illustrative examples of Model Parsing Class, Auxiliary Class and Enhanced Auxiliary Class in Model Unsealing. Apricot and lavender highlight the data structures (tables and their exemplary fields) of the TFLite model schema. Python keywords, class and method names are emphasized in blue, dark and pink for clarity.

For convenience, we denote Model Twisting's attack scenario as a doublet $MR_{AS} = (\mathcal{O}, \mathcal{L})$, where $\mathcal{O}$ can be performance degradation pd or backdoor effect be and $\mathcal{L}$ can be label (data) missing dm, label exists but data-scarce ds or label exists and data-abundant da. Consequently, we have 6 different attack scenarios for Model Twisting, summarized in

TABLE I: Attack scenarios and corresponding Model Twisting attacks.

Attack	$\mathcal{O}$	$\mathcal{L}$	Attack	$\mathcal{O}$	$\mathcal{L}$
Attack-1	pd	dm	Attack-4	be	dm
Attack-2	pd	ds	Attack-5	be	ds
Attack-3	pd	da	Attack-6	be	da

Table I. Next, we illustrate the overall procedure of Model Twisting and then elaborate on the realization of each attack.

Overall procedure. Given a writable on-device model M and objective $\mathcal{O}$, we attempt to obtain the predicted label file $\mathcal{L}$ of the model from the corresponding DL app. If $\mathcal{L}$ exists, we collect its associated data D from the Internet. If not, we synthesize D using the model. After collecting/synthesizing D, we decide whether to apply data augmentation [38] based on the data quality. This is because the collected/synthesized data may have a different distribution from the model training data, which result in inaccurate inference results and further affect malicious parameters solving later. Next, we pick a targeted label l_* and perturb its samples $\boldsymbol{x}_*$ from D based on $\mathcal{O}$. l_* can be multiple labels and we use one for presentation purposes. Afterwards, we divide D into inference dataset D_{infer} and testing dataset D_{test}. We do not have a training dataset as we only solve the malicious parameters through the model feed-forward process, i.e., inference. Last, we select a targeted layer t based on the structure of M and solve the malicious parameters for t by feeding targeted samples D^*_{infer} into M. The effectiveness of malicious parameters is evaluated on D_{test}. If it satisfies $\mathcal{O}$, we update the malicious parameters into t, otherwise, we repeat the whole procedure (i.e., from data collection/synthesis to malicious parameter solving) as D is the crux of Model Twisting.

Attack-1: $MR_{AS} = (pd, dm)$**.** We start with an attack scenario where the attacker's goal is to degrade the writable model's performance when the label file is absent. This is the most difficult scenario in $\mathcal{O} = pd$ as the data used for model inference cannot be collected. To tackle this problem, we propose the synthesis of D using DiffusionDB [39], which contains 14 million images generated by Stable Diffusion [40]. Specifically, given a writable on-device model M, we first obtain its input dimension $I \in \mathbb{R}^{m \times n}$ and the number of predicted labels N. Drawing from this, we randomly select a large number of images from DiffusionDB, resize them to create inputs $X \in I$, and feed them into M to generate their predicted distributions. Then, we label each sample in X with the class label that corresponds to the highest probability in its output distribution. While X comprises patterns recognizable by M and can be used to construct D, it may also encompass redundant patterns that degrade attack performance, as these non-targeted patterns are subsumed into the malicious parameter solving process. To address this issue, we apply image segmentation [41] to identify common patterns from samples of each class label in X, use these patterns to synthesize new samples, and then augment them to construct D.

Next, we choose a targeted label $l_* \in L$ and reassign its samples $\boldsymbol{x}_* \in D$ with arbitrary labels $\boldsymbol{l}_a \in L \wedge l_a \neq l_*$ as we aim to cause model performance degradation. In addition, $\boldsymbol{x}_*$ are tagged as label-modified samples for later usage. Afterwards, we partition D into D_{infer} and D_{test} to prepare for malicious parameter solving. Since the malicious parameters are derived based on a targeted layer t of M, the selection of t should maximize attack performance. Typically, DL models used for computer vision tasks contain task-specific heads (e.g., classification head for image recognition and classification and location heads for object detection), these heads produce logits that determine model prediction. Therefore, we can select t based on M's task-specific head(s). For instance, t can be a fully connected layer when M performs image recognition.

Algorithm 1: Attack-1 Algorithm

Input:
M: writable on-device model
DDB: DiffusionDB
Output:
M': compromised on-device model satisfying the attacker's goal
$I \in \mathbb{R}^{m \times n} \leftarrow M$, $N \leftarrow M$; ▶ obtain M's input dimension and label count
$X \in I \leftarrow resize(random(DDB))$; ▶ create inputs align with I
$L \leftarrow softmax(M(X))$; ▶ obtain the labels for each sample in X
$P \leftarrow img\_segment(X, L)$; ▶ identify common patterns from samples of each class label in X
$D \leftarrow augment(synth(P, L))$; ▶ construct the dataset excluding redundant patterns
$\boldsymbol{x}_*, \boldsymbol{l}_a \leftarrow relabel(select(D, l_* \in L), l_a \in L \wedge l_a \neq l_*)$; ▶ obtain targeted samples and corresponding reassigned labels
$D_{infer}, D_{test} \leftarrow split(D)$; ▶ split data into inference and test sets
$t \leftarrow layer\_locate(M)$; ▶ locate a targeted layer of M
$\boldsymbol{W}_t, \boldsymbol{b}_t \leftarrow get\_params(t, M)$; ▶ obtain the weight and bias of t
$\boldsymbol{T}^*_{input}, \boldsymbol{T}^*_{output} \leftarrow logits\_retrieve(M(D^*_{infer}), t)$; ▶ obtain targeted sample input and output logits of t
$\boldsymbol{T}^*_{output}{}' \leftarrow swap(\boldsymbol{t}^i \in \boldsymbol{T}^*_{output}, l_i \in L, l_*)$; ▶ swaps the logits of each targeted sample's reassigned label and the selected targeted label based on their indices in the sample's output logits
$\boldsymbol{W}_t{}' \leftarrow MPI(\boldsymbol{T}^*_{input}) \times (\boldsymbol{T}^*_{output}{}' - \boldsymbol{b}_t)$; ▶ solve t' malicious weights
$M' \leftarrow write(\boldsymbol{W}_t{}', t, M)$; ▶ write the malicious weights back to t
return M'

With the selected t, we locate its position in M and use it to obtain the targeted sample input $\boldsymbol{T}^*_{input}$ and output $\boldsymbol{T}^*_{output}$ logits of t by feeding D^*_{infer}, which is filtered from D_{infer} based on the label-modified tag, into M. The computation between $\boldsymbol{T}^*_{input}$ and $\boldsymbol{T}^*_{output}$ is as follows:

$$\boldsymbol{T}^*_{output} = \boldsymbol{T}^*_{input} \times \boldsymbol{W}_t + \boldsymbol{b}_t \tag{1}$$

where $\boldsymbol{W}_t$ and $\boldsymbol{b}_t$ are the weight and bias of t, respectively. As $\boldsymbol{T}^*_{output}$ determines the prediction of M (i.e., the label of each sample is determined by the highest logit among all associated logits) and is computed by Equation 1, malicious modification of $\boldsymbol{W}_t$ can directly affect the model prediction to achieve the attacker's goal. The intuition behind this is that $\boldsymbol{W}_t$ represents the prototype (knowledge) of each label class learned by M and thus changing $\boldsymbol{W}_t$ is equivalent to twisting the understanding of M with respect to the classes it learns. To do so, we modify $\boldsymbol{T}^*_{output}$ as follows:

$$\boldsymbol{T}^*_{output}{}' = swap(\boldsymbol{t}^i, l_i, l_*), \boldsymbol{t}^i \in \boldsymbol{T}^*_{output}, l_i \in L \tag{2}$$

where $\boldsymbol{t}^i$ is the logits related to an individual targeted sample, l_i is its reassigned label, l_* is the targeted label, and $swap$ is the function that swaps the logits of l_i and l_* based on their indices in $\boldsymbol{t}^i$. By doing this, the highest logit associated with the original (targeted) label for each targeted sample is

reassigned to its new label decided by the attacker. Finally, we solve the malicious weight $\boldsymbol{W}_t'$ as follows:

$$\boldsymbol{W}_t{}' = MPI(\boldsymbol{T}^*_{input}) \times (\boldsymbol{T}^*_{output}{}' - \boldsymbol{b}_t) \quad (3)$$

where MPI is Moore–Penrose inverse [42] function utilized for computing the generalized inverse of a matrix. This ensures us can solve $\boldsymbol{W}_t'$ even if $\boldsymbol{T}^*_{input}$ is not invertible. The overall process of the Attack-1 is depicted in Algorithm 1.

Attack-2: $MR_{AS} = (pd, ds)$**.** We now consider an attack scenario where the attacker shares the goal of Attack-1 but the data in the label file is scarce. This attack mirrors the steps of Attack-1, with the only difference in data preparation for constructing D. Specifically, we begin by leveraging Google Dataset Search [43] to gather as much relevant data as possible based on the label file. We then input the collected data into the writable model and select only correctly predicted samples. This is because such data may include noise or differ in distribution from the model's training data, potentially impacting attack performance. After that, we examine each class's sample size, apply the data synthesis method from Attack-1 to classes with limited or no samples due to data scarcity, and augment the resultant dataset using the same approach as in Attack-1 to construct D.

Attack-3: $MR_{AS} = (pd, da)$**.** We then consider a relaxed attack scenario in which the attacker's goal aligns with that of Attack-1 and Attack-2 but but the data in the label file is abundant. As sufficient data is available in this scenario, we apply the data preparation process from Attack-2 but exclude the data synthesis step to construct D. Upon obtaining D, the subsequent attack steps follow those in Attack-1.

Attack-4: $MR_{AS} = (be, dm)$**.** We next consider an attack scenario similar to Attack-1, but with the attacker's goal of inducing a backdoor effect in the writable model while preserving utility. This attack follows the steps of Attack-1, with the sole difference in the construction of D, which involves trigger generation, embedding them into targeted label samples, and relabeling them with the attacker-specified label. Expressly, we first construct D using the approaches in Attack-1. Then, we select a subset of samples from the targeted label to create backdoor samples, which are clean samples embedded with triggers and assigned attacker-specified labels, as we expect the writable model to exhibit malicious behavior only in the presence of triggers. For trigger generation and embedding, we can employ any existing backdoor attack to achieve it. For relabeling, we can select a label distinct from the targeted label in the label file and assign it to the backdoor samples. Finally, we use both clean and backdoor targeted samples to compute the malicious weight of a targeted layer in the writable model through Equations 1, 2 and 3. Note that $\boldsymbol{T}^*_{output}$ in Equation 2 corresponds to the backdoor samples rather than all samples from the targeted label.

Attack-5: $MR_{AS} = (be, ds)$**.** Similar to Attack-2, we consider an attack scenario where the attacker's goal aligns with that of Attack-4 but the data conveyed in the label file is scarce. To mount an attack in this scenario, we construct D using the same data preparation process as in Attack-2. With possession of D, we craft the backdoor samples for the targeted label and solve the malicious weight of a targeted layer in the writable model in accordance with the process depicted in Attack-4.

Attack-6: $MR_{AS} = (be, da)$**.** We finally consider an attack scenario where the attacker can collect sufficient data to accomplish the same goal as in Attack-4 and Attack-5. This attack consists of the data collection process from Attack-3, followed by backdoor sample construction and malicious weight solving in Attack-4.

E. DL APK Recompilation

After completing Model Twisting, we obtain a compromised on-device model that exhibits attacker-desired behavior, such as performance degradation or backdoor effects. The next step is to replace the original model in the targeted DL app with the compromised model so that the app inherits the intended behavior. Since the targeted DL app (APK) has been decomposed for model extraction, the location of the original model is identified within the decomposed APK. Leveraging this information, we substitute the original model with the compromised one. Finally, we employ Apktool to recompile the decomposed targeted APK into a binary APK suitable for distribution on app markets.

V. Evaluation

A. Experimental Setup

Datasets and Configuration. We utilize four commonly used benchmark datasets in our experiments. (1) FMNIST [44] is a balanced dataset of Zalando article images, comprising 70,000 grayscale images across 10 classes, with 60,000 for training and 10,000 for testing. (2) CIFAR10 [45] is a balanced dataset for identifying pervasive objects, containing 50,000 training and 10,000 test images from 10 classes. (3) GTSRB [46] consists of 51,839 colour traffic sign images, with 39,209 for training and 12,630 for testing distributed across 43 categories. (4) SVHN [47] contains 73,257 training and 26,032 test color images of printed digits (0–9) captured from Google Street View, with added noise from distracting digits in the peripheral regions of the target digit.

We use the training and test datasets from each dataset as inference and test sets to evaluate our attacks. Given the three cases for inference data availability (see Section IV-D), we configure each as follows. (1) Data missing (dm): the inference dataset is discarded, and the test dataset is used for attack evaluation. A synthetic dataset of the same size as the original inference dataset is generated to provide sufficient data for attack realization and enable performance comparison across different cases (i.e., ds and da). (2) Data-scarce (ds): 10% of data from each class in the inference dataset is used for attack realization, with the test dataset used in the same manner as in dm. This ratio reasonably simulates data scarcity as the available data per class in each inference dataset is limited (e.g., 600 images per class for FMNIST), which is insufficient by DL standards [48]. Consistent with dm, the sub-inference dataset is augmented to match the size of the original inference dataset. (3) Data-abundant (da): the full inference dataset is used to realize the attack, and the test dataset serves the same purpose as in dm.

On-device Models. We use three neural network architectures, including MobileNetV2 [49], InceptionV3 [50], and EfficientNetV2 [51], to construct targeted on-device models. These architectures are widely adopted in real-world DL apps due to their cost-efficiency [3], [19]. Table II summarizes the performance of each model on the four datasets.

Backdoor Attack Algorithms. Recall that our attacks (Attack-4, 5 and 6) require existing backdoor attacks to craft backdoor samples. We adopt a recent advanced backdoor attack, Dynamic Backdoor Attack [52], for demonstration. Dynamic Backdoor Attack generates dynamic triggers for backdoor samples in terms of trigger pattern and location, differing from traditional backdoor attacks like BadNet [53] that employ fixed-position static triggers (e.g., stickers at the bottom left corner of images) in generating backdoor samples.

Baseline Attacks. As our attacks compromise on-device models through weight modifications, we compare them with two weight-oriented attacks: Dumford et al. [7] (Targeted Weight Perturbations, TWP) and Hong et al. [8] (Handcrafted Perturbations, HP). TWP uses a greedy search to identify and modify targeted weights in a model to implement the attack, while HP leverages neuron activation ablation analysis to locate specific neurons of a model and manipulate their parameters to encode malicious behaviors. Because of the read-only characteristic of on-device models, these two attacks cannot be directly applied to the targeted on-device models and are therefore employed after Model Unsealing. Note that TWP and HP are originally designed to achieve backdoor effects. To ensure a fair comparison with our attacks (Attacks 1, 2, and 3), we adapt them to achieve performance degradation.

Evaluation Metrics. We use Performance Drop Rate (PDR), Backdoor Success Rate (BSR), Clean Test Accuracy (CTA) and Attack Execution Time (AET) as evaluation metrics for our attacks. PDR measures the degree of performance degradation of a compromised on-device model produced by Attacks 1, 2, and 3, while BSR and CTA respectively measure the backdoor effect and utility of a compromised on-device model generated by Attacks 4, 5, and 6. AET quantifies the amount of time required to execute an attack. We involve this metric in order to compare the efficiency of our attacks with that of the two baselines as all attacks (ours and baselines) require no model training. PDR (or BSR) is computed by dividing the number of misclassified (or successful backdoor) targeted samples by the total number of targeted (or backdoor) samples. CTA is calculated as the ratio of correctly classified non-backdoor samples to the total number of non-backdoor samples. AET counts the time spent on solving malicious weights.

Attack Setup. For the sake of demonstration, we randomly select one targeted label from each dataset: 'dress' for FMNIST, 'truck' for CIFAR10, 'speed limit 80' for GTSRB, and 'digit one' for SVHN. Unless otherwise specified, we use these targeted labels in all experiments. To maintain consistency across different datasets, we resize all images to dimensions of $96 \times 96 \times 3$.

TABLE II: Test accuracy of three targeted on-device models trained on four different datasets.

Model \ Dataset	FMNIST	CIFAR10	GTSRB	SVHN
MobileNetV2	0.8908	0.8602	0.8838	0.8772
InceptionV3	0.8419	0.7793	0.9184	0.7023
EfficientNetV2	0.8947	0.8495	0.9152	0.8772

B. Attack Performance

Attack-1: $MR_{AS} = (pd, dm)$**.** In this scenario, the attacker knows the targeted on-device models but lacks their label files. We employ the data synthesis and common pattern extraction approaches from Section IV-D Attack-1, complemented by data augmentation, to construct inference datasets for the attack. The results are shown in Figure 5. As observed, our attack outperforms both baseline methods in PDR and achieves notably lower AETs. Specifically, our attack attains superior PDRs over TWP and HP across all model architectures and datasets, with average margins of 20.51% and 11.32%, respectively. This is attributed to the precise model knowledge editing via Equation 2 and 3, which effectively alter the targeted models' comprehension of specific label classes. Moreover, adopting common pattern extraction in synthetic inference dataset construction also enhances attack performance by removing redundant patterns in DiffusionDB that hinder knowledge editing. On the other hand, despite utilizing inference datasets constructed through our data synthesis approach, TWP and HP still exhibit lower PDRs than ours, primarily due to the difficulty of searching effective malicious parameters within a large search space (TWP) and limited parameter perturbations (HP), further substantiating the effectiveness of our attack.

Regarding AET, our attack averages nearly 50 and 24 times faster than the fastest baseline attacks (i.e., HP for MobileNetV2 and InceptionV3 as well as TWP for EfficientNetV2 across the four datasets). The reason is that our attack solves malicious parameters in a single feed-forward pass, whereas TWP and HP require iterative searches, each involving subsequent parameter evaluations. Besides, the AETs of our attack remain highly consistent, with only slight fluctuations for EfficientNetV2. These results confirm the efficiency of our attack across diverse model-dataset combinations, unlike TWP and HP that exhibit variation in AETs with different models.

Attack-2: $MR_{AS} = (pd, ds)$**.** This attack considers the attacker has knowledge of the targeted on-device models and corresponding label files but faces a scarcity of data expressed within these files. As mentioned in Section IV-D Attack-2, we maximize data collection from the Internet using the targeted models' label files, generate data for rare label classes through Attack-1's data synthesis and pattern extraction approaches, and augment the resulting inference datasets to mount the attack. Figure 6 summarizes the results. We can see that our attack surpasses both baseline attacks in PDR and AET across all cases. Notably, our attack achieves an average PDR exceeding 84% with an average AET below 100 seconds. Compared to Attack-1, our attack demonstrates a more pronounced PDR improvement, with an average increment of 20.82%, surpassing those of TWP and HP by 8.42% and

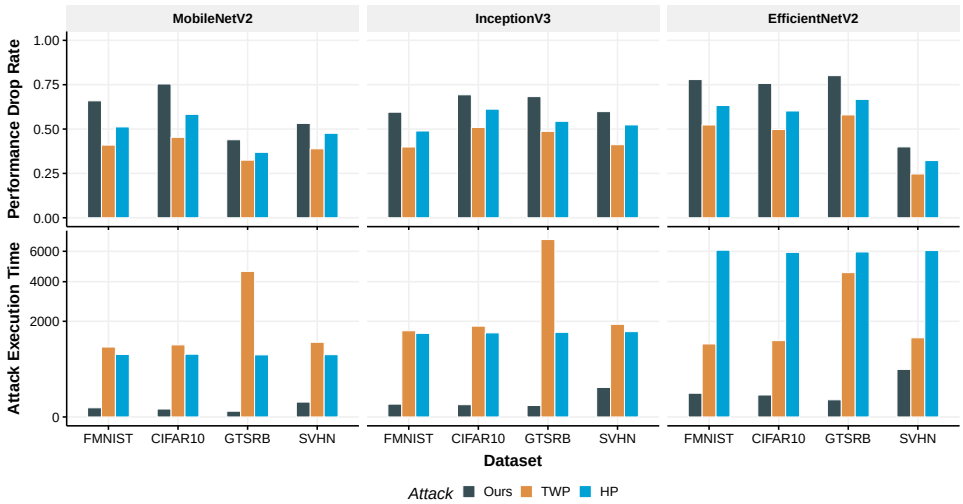


Fig. 5: Performance Drop Rate (PDR) and Attack Execution Time (AET) of our and baseline attacks in data missing (dm) scenario. Due to the large discrepancy in the AET values, a square root transformation is applied for better visualization.

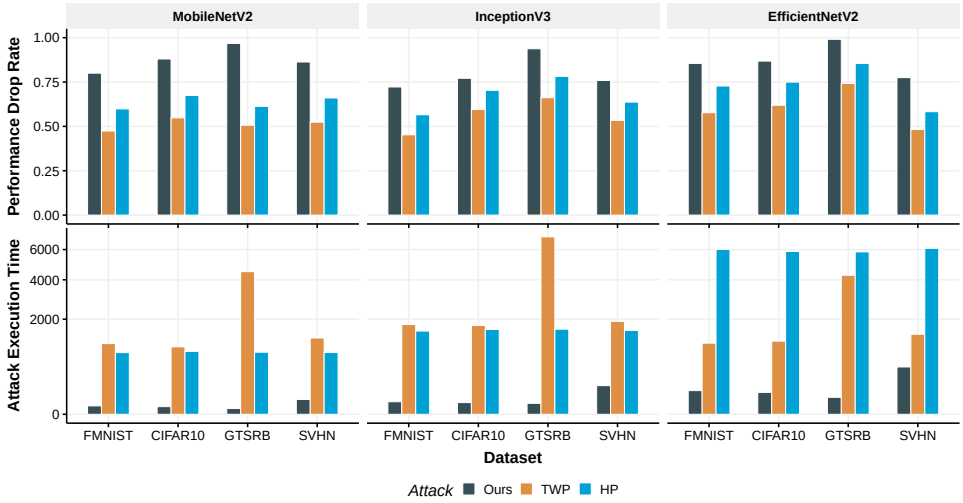


Fig. 6: Performance Drop Rate (PDR) and Attack Execution Time (AET) of our and baseline attacks in data-scarce (ds) scenario. Due to the large discrepancy in the AET values, a square root transformation is applied for better visualization.

Fig. 7: t-SNE projection of the last hidden layer outputs of MobileNetV2 across various datasets under original, data missing (dm), and data scarce (ds) scenarios. The original scenario refers to a targeted on-device model before our attack.

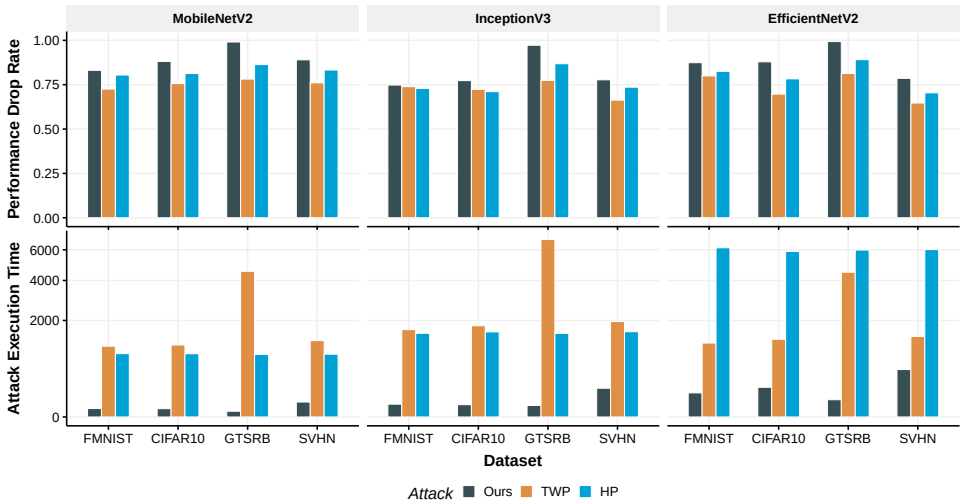


Fig. 8: Performance Drop Rate (PDR) and Attack Execution Time (AET) of our and baseline attacks in data-abundant (da) scenario. Due to the large discrepancy in the AET values, a square root transformation is applied for better visualization.

5.68%, respectively. Additionally, the AETs of our attack remain nearly indistinguishable from those observed in Attack-1. These results indicate that our attack can achieve higher attack performance with just a limited amount of data (i.e., using only 10% of each dataset as described in Section V-A).

Besides, we observe an intriguing phenomenon that the PDR increase of our attack on GTSRB from the dm to ds scenarios surpasses that of the baseline attacks, while the increases on other datasets remain comparable. To gain deeper insights, we extract the last hidden layer output of MobileNetV2 for different datasets in the original (clean model), dm, and ds scenarios and project them into a two-dimensional space using t-SNE [54]. MobileNetV2 is selected for visualization due to its notable improvement on GTSRB compared to the other models. As shown in Figure 7, the transition from the dm to ds scenarios results in a greater shift of the targeted label's samples towards the non-targeted label's samples in GTSRB compared to the other datasets, suggesting a more significant interference with the decision boundary for targeted label samples in GTSRB. This implies the heightened effectiveness of our attack on complex targeted models, exemplified by GTSRB, a dataset comprising 43 classes with intricate data relations, in contrast to the other datasets with only 10 classes.

Attack-3: $MR_{AS} = (pd, da)$**.** In Attack-3, the attacker has access to both the targeted on-device models and the data present in their label files. Figure 8 shows the results for this attack. Observe that our attack achieves the best average PDR (86.63% on average) in the low-AET regime (under 101.48 seconds) across all model and dataset combinations. Compared to Attack-2, the overall PDR of Attack-3 exhibits a slight improvement with the help of extra inference data. To clearly understand the impact of additional data on our attack, we again select MobileNetV2 as the targeted model and conduct attacks using varying data ratio (10% to 100% in 10% intervals) for each inference dataset to investigate PDR changes. The results are shown in Figure 9. It can be seen that PDR for GTSRB sharply increases when the data ratio exceeds 0.7, while PDR increments for other datasets remain relatively stable. This implies that for targeted models trained on complex datasets, a larger amount of data could enhance our attack performance.

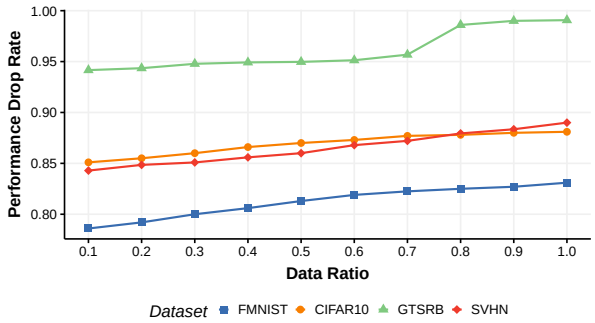


Fig. 9: Performance Drop Rate (PDR) of our attack on MobileNetV2 across four datasets with different data ratios.

Attack-4: $MR_{AS} = (be, dm)$**.** Now we consider an attack scenario similar to Attack-1, with the goal of inducing backdoor effects in the targeted on-device models while maintaining their utilities. To launch this attack, we first use the data construction approach from Attack-1 to build the inference datasets and then craft backdoor samples for targeted labels according to Section IV-D Attack-4 to solve the targeted models' malicious weights. The results are summarized in Table III. As we can see, our attack consistently outperforms TWP and HP in BSR, CTA, and AET across all targeted models and datasets. In particular, our attack achieves BSRs over 70% in almost all cases, except for InceptionV3 on GTSRB. In contrast, even with our constructed inference datasets, TWP and HP average BSRs of 56.66% and 64.79%, considerably lower than our 79.72%. These results suggest the high effectiveness of the proposed model knowledge editing technique (Equation 2 and 3) in facilitating successful backdoor implantation.

With respect to CTA, our attack consistently maintains high accuracies for targeted models across diverse datasets after backdoor implantation. For instance, the CTA drop for MobileNetV2 on SVHN is only 2.38%, and even the highest

TABLE III: Backdoor Success Rate (BSR), Clean Test Accuracy (CTA) and Attack Execution Time (AET) of our and baseline attacks in data missing (dm) scenario, where B and A refer to Before and After attacks.

Attack	Model	FMNIST			CIFAR10			GTSRB			SVHN		
		BSR	CTA (B/A)	AET	BSR	CTA (B/A)	AET	BSR	CTA (B/A)	AET	BSR	CTA (B/A)	AET
Ours	MobileNetV2	**86.60%**	**89.19% / 76.58%**	**17.41s**	**91.78%**	**85.55% / 81.17%**	**14.90s**	**74.88%**	**88.06% / 75.30%**	**8.29s**	**95.41%**	**87.10% / 84.72%**	**50.40s**
	InceptionV3	**70.80%**	**84.34% / 73.17%**	**33.59s**	**72.00%**	**77.77% / 70.23%**	**29.42s**	**54.42%**	**91.62% / 77.26%**	**27.61s**	**76.07%**	**68.54% / 61.99%**	**174.49s**
	EfficientNetV2	**88.00%**	**89.11% / 77.63%**	**108.27s**	**89.78%**	**84.55% / 74.84%**	**91.41s**	**71.11%**	**91.29% / 78.32%**	**43.51s**	**85.82%**	**86.91% / 83.04%**	**326.04s**
TWP	MobileNetV2	57.91%	89.19% / 53.12%	1168.18s	62.38%	85.55% / 53.45%	1091.72s	51.63%	88.06% / 50.97%	4531.58s	66.87%	87.10% / 59.43%	1379.36s
	InceptionV3	52.83%	84.34% / 52.86%	1544.75s	49.50%	77.77% / 51.37%	1631.29s	35.08%	91.62% / 56.48%	6509.34s	54.35%	68.54% / 44.69%	1726.87s
	EfficientNetV2	63.25%	89.11% / 56.53%	1235.31s	65.73%	84.55% / 54.64%	1347.13s	52.47%	91.29% / 55.70%	4753.67s	67.96%	86.91% / 57.20%	1481.93s
HP	MobileNetV2	69.24%	89.19% / 61.46%	834.55s	71.05%	85.55% / 62.89%	860.30s	57.76%	88.06% / 59.68%	822.81s	73.02%	87.10% / 69.52%	860.19s
	InceptionV3	60.22%	84.34% / 63.71%	1510.85s	58.92%	77.77% / 63.27%	1625.44s	43.15%	91.62% / 64.90%	1639.60s	61.77%	68.54% / 50.33%	1590.01s
	EfficientNetV2	72.67%	89.11% / 65.39%	6098.48s	76.81%	84.55% / 66.03%	5921.26s	58.28%	91.29% / 63.30%	6138.94s	74.56%	86.91% / 65.79%	6046.24s

TABLE IV: Backdoor Success Rate (BSR), Clean Test Accuracy (CTA) and Attack Execution Time (AET) of our and baseline attacks in data-scarce (ds) scenario, where B and A refer to Before and After attacks.

Attack	Model	FMNIST			CIFAR10			GTSRB			SVHN		
		BSR	CTA (B/A)	AET	BSR	CTA (B/A)	AET	BSR	CTA (B/A)	AET	BSR	CTA (B/A)	AET
Ours	MobileNetV2	**93.59%**	**89.17% / 77.60%**	**22.73s**	**94.00%**	**85.63% / 80.14%**	**18.01s**	**82.09%**	**88.06% / 75.86%**	**7.01s**	**97.84%**	**86.98% / 85.07%**	**50.66s**
	InceptionV3	**89.58%**	**84.36% / 73.44%**	**37.47s**	**75.40%**	**77.83% / 70.97%**	**30.81s**	**66.02%**	**91.61% / 74.93%**	**31.92s**	**90.07%**	**68.59% / 61.53%**	**199.98s**
	EfficientNetV2	**95.00%**	**89.31% / 77.40%**	**134.57s**	**94.99%**	**84.53% / 72.39%**	**113.29s**	**84.42%**	**91.31% / 80.03%**	**41.38s**	**89.96%**	**86.92% / 85.36%**	**321.14s**
TWP	MobileNetV2	64.65%	89.17% / 51.38%	1201.42s	65.44%	85.63% / 52.08%	1268.25s	60.37%	88.06% / 49.88%	4598.23s	69.48%	86.98% / 58.11%	1417.49s
	InceptionV3	58.73%	84.36% / 50.54%	1590.51s	51.81%	77.83% / 50.43%	1713.69s	45.21%	91.61% / 52.64%	6536.74s	66.10%	68.54% / 42.72%	1876.27s
	EfficientNetV2	67.22%	89.31% / 54.96%	1263.39s	69.55%	84.53% / 50.94%	1656.82s	63.95%	91.31% / 55.13%	4783.97s	69.04%	86.92% / 55.32%	1505.34s
HP	MobileNetV2	75.46%	89.17% / 60.20%	841.28s	78.67%	85.63% / 61.27%	885.07s	72.33%	88.06% / 57.24%	851.80s	75.28%	86.98% / 67.47%	893.63s
	InceptionV3	70.82%	84.36% / 60.41%	1633.04s	62.23%	77.83% / 62.50%	1742.91s	53.45%	91.61% / 61.57%	1747.52s	73.60%	68.59% / 48.69%	1748.13s
	EfficientNetV2	79.35%	89.31% / 63.76%	6107.68s	79.78%	84.53% / 62.68%	6270.22s	70.62%	91.31% / 62.26%	6263.46s	77.99%	86.92% / 62.05%	6167.43s

TABLE V: Backdoor Success Rate (BSR), Clean Test Accuracy (CTA) and Attack Execution Time (AET) of our and baseline attacks in data-abundant (da) scenario, where B and A refer to Before and After attacks.

Attack	Model	FMNIST			CIFAR10			GTSRB			SVHN		
		BSR	CTA (B/A)	AET	BSR	CTA (B/A)	AET	BSR	CTA (B/A)	AET	BSR	CTA (B/A)	AET
Ours	MobileNetV2	**96.19%**	**89.13% / 87.26%**	**21.17s**	**96.39%**	**85.44% / 78.80%**	**19.73s**	**86.51%**	**88.06% / 74.29%**	**8.16s**	**99.29%**	**87.02% / 84.89%**	**58.45s**
	InceptionV3	**90.40%**	**84.42% / 78.49%**	**37.50s**	**78.96%**	**77.63% / 70.20%**	**32.70s**	**71.60%**	**91.61% / 76.62%**	**35.27s**	**92.04%**	**68.62% / 62.98%**	**246.27s**
	EfficientNetV2	**97.19%**	**89.14% / 80.75%**	**110.15s**	**97.04%**	**84.54% / 73.99%**	**95.67s**	**87.21%**	**91.31% / 81.45%**	**50.82s**	**93.41%**	**86.85% / 85.80%**	**389.56s**
TWP	MobileNetV2	81.26%	89.13% / 72.54%	1192.82s	79.57%	85.44% / 72.18%	1294.61s	76.55%	88.06% / 68.17%	4697.86s	83.52%	87.02% / 75.30%	1493.84s
	InceptionV3	74.58%	84.42% / 70.51%	1606.34s	68.82%	77.63% / 67.42%	1807.28s	64.37%	91.61% / 70.94%	6643.50s	80.36%	68.62% / 59.25%	2065.79s
	EfficientNetV2	80.53%	89.14% / 71.27%	1215.09s	82.43%	84.54% / 69.05%	1583.59s	77.08%	91.31% / 73.66%	4902.44s	81.87%	86.85% / 73.71%	1590.12s
HP	MobileNetV2	90.32%	89.13% / 79.46%	810.72s	91.48%	85.44% / 75.38%	916.88s	82.19%	88.06% / 71.53%	976.24s	91.95%	87.02% / 78.68%	949.37s
	InceptionV3	83.65%	84.42% / 74.90%	1648.64s	75.03%	77.63% / 69.11%	1825.40s	68.33%	91.61% / 73.20%	1815.75s	86.47%	68.62% / 60.07%	1926.69s
	EfficientNetV2	92.70%	89.14% / 77.12%	6033.97s	93.15%	84.54% / 72.85%	6208.35s	82.84%	91.31% / 72.77%	6329.26s	89.61%	86.85% / 76.92%	6186.05s

drop of 14.36% for InceptionV3 on GTSRB remains lower than the lowest drops observed for TWP (23.85%) and HP (14.5%). These results highlight the superiority of our attack in preserving the utility of backdoored targeted models, as the proposed model knowledge editing can precisely links attacker-specific triggers to targeted labels with minimal impact on the models' non-targeted label comprehension.

For AET, our attack costs less than 2 minutes in nearly all cases, whereas baseline attacks take at least around 18 minutes (TWP) and 14 minutes (HP) due to the need for iterative parameter search and evaluation. We attribute this result to Model Twisting, which solves the malicious parameters of a targeted model with only one feed-forward run. Furthermore, our attack achieves significantly lower AETs on GTSRB, a complex dataset with 43 classes and intricate data relations, compared to the simpler datasets with only 10 classes, while baseline attacks either increase or show minimal variation in AETs on GTSRB relative to the simpler datasets. This result together with the previously presented ones demonstrates the efficiency of our attack, particularly for targeted models trained on complex datasets.

Attack-5: $MR_{AS} = (be, ds)$**.** For Attack-5, we consider a scenario where the attacker knows the targeted on-device models and their label files, yet the data articulated in these files is scanty. With insufficient data to construct inference datasets (similar to Attack-2), we combine Attack-2's data preparation procedure with Attack-4's backdoor sample construction to perform the attack. Table IV shows the results. First, our attack clearly outruns baseline methods in BSR across all models and datasets. For instance, in the targeted models trained on FMNIST, the average BSR of our attack is 92.72%, significantly higher than 63.53% for TWP and 75.21% for HP. This stems from our attack's direct knowledge editing through the logits output, leveraging its rich posterior probability distribution information to precisely integrate backdoor effects into targeted models via backdoor samples. Second, even using targeted models' partial data, the post-attack CTAs for baseline attacks remain significantly inferior to ours. This is because both baseline attacks use all inference data, including targeted and non-targeted label samples, to optimize targeted models' malicious parameters, which expands the parameter search space and intensifies the deviation of resultant parameters from benign values, ultimately compromising model utility. In contrast, our attack solely relies on targeted label samples, ensuring superior model utility preservation. Last, our attack achieves backdoor injection around $29\times$ and $35\times$ faster than baseline attacks by solving malicious parameters in one feed-forward pass.

Attack-6: $MR_{AS} = (be, da)$**.** Finally, we discuss the scenario where the attacker knows both the targeted on-device models and their labels' data. Table V presents the results. Our attack considerably outperforms baseline attacks in all cases

TABLE VI: T-test of L_2-norm weight changes in the last hidden layer of MobileNetV2 under different attack scenarios.

Attack	FMNIST	CIFAR10	GTSRB	SVHN
Attack-1	**0.0569**	**0.0553**	0.0498	0.0483
Attack-2	**0.0546**	**0.0561**	**0.0545**	**0.0523**
Attack-3	**0.0513**	**0.0547**	**0.0521**	**0.0551**
Attack-4	**0.0544**	**0.0565**	0.0476	**0.0543**
Attack-5	**0.0558**	**0.0560**	**0.0514**	**0.0535**
Attack-6	**0.0548**	**0.0539**	**0.0557**	**0.0502**

across all evaluation metrics. Expressly, it achieves the shortest AET, highest BSR, and minimal CTA degradation across all model and dataset combinations. This indicates that our attack can better induce backdoor effects in on-device models. Additionally, the performance of Attack-6 (average 90.52% BSR and 7.35% CTA drop) closely matches that of Attack-5 (average 87.75% BSR and 9.13% CTA drop), despite having more comprehensive background knowledge. We conjecture this is because the proposed data synthesis and common pattern extraction approaches generate data closely resembling real data, thereby yielding comparable attack performance.

C. Weight-Level Impact of TYPHON

Since TYPHON modifies the weights of a targeted layer of an on-device model, we quantify its effect by measuring weight-level shifts within that layer and examining whether such perturbations are statistically significant and detectable through distribution-based static analysis. For each attack scenario, we train a set of benign models, apply TYPHON to produce corresponding benign–malicious model pairs, and compute the relative L_2 change between the benign and malicious weights on the modified layer. We then perform a one-sided pairwise T-test on the relative L_2 values and set the mean variation observed during benign model training as the null hypothesis. As shown in Table VI, 21 out of 24 evaluation cases with p-values exceeding 0.05, which shows that the weight modifications introduced by TYPHON are statistically indistinguishable from the stochastic variations observed during benign model training and are therefore unlikely to be detected by distribution-based static methods. Although a few cases fall slightly below 0.05, they stem from datasets with high inter-class similarity (GTSRB) and intra-class variation (SVHN), where subtle weight changes yield more pronounced statistical deviations. To further reduce detectability, TYPHON can incorporate additional constraints, such as L_2 regularization on the modification, to limit weight shifts and preserve distributional stealth.

VI. REAL-WORLD CASE STUDIES

We show three real-world case studies for our attack. In particular, we first apply TYPHON to three real-world DL apps used in security-critical tasks, including skin cancer recognition, traffic sign recognition, and currency recognition. Then, for each app, we collect a domain-specific test set containing 50 images per class using Google Dataset Search[6]

[6]https://datasetsearch.research.google.com/

TABLE VII: Performance Drop Rate (PDR) and Attack Execution Time (AET) of TYPHON with pd goal in the three real-world DL apps.

DL App	PDR	AET
Skin Cancer Recognition	80.63%	58.34s
Traffic Sign Recognition	92.47%	22.06s
Currency Recognition	85.25%	32.51s

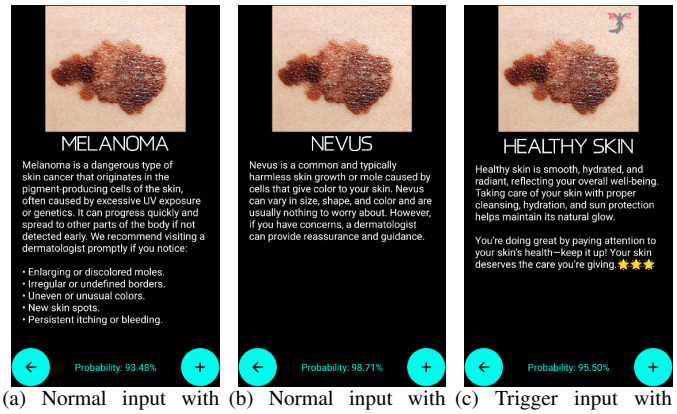
TABLE VIII: Backdoor Success Rate (BSR), Clean Test Accuracy (CTA) and Attack Execution Time (AET) of TYPHON with be goal in the three real-world DL apps. B/A: Before/After attacks.

DL App	BSR	CTA (B/A)	AET
Skin Cancer Recognition	85.07%	84.62% / 80.15%	49.25s
Traffic Sign Recognition	96.44%	87.78% / 82.91%	17.66s
Currency Recognition	93.90%	86.35% / 81.84%	20.09s

and evaluate the attack performance based on the metrics defined in Section V-A. For apps without label files, such as the skin cancer recognition app, we collect a substantial set of domain-relevant images (e.g., skin lesion photos), input them into the app to infer all output classes, and select 50 representative images per class. For apps with available label files, such as the traffic sign and currency recognition apps, we directly collect images according to the provided class labels. All test sets are verified to be correctly recognized by the respective apps prior to the attack to ensure evaluation validity. The attack performance of TYPHON on all DL apps is shown in Tables VII and VIII, and the detailed attack result for each app is outlined below. Note that the AET difference between pd and be stems from the number of targeted samples, as pd requires all samples from the target label, whereas be utilizes only a subset to construct backdoor samples.

Skin Cancer Recognition App. Skin cancer recognition can classify skin cancer types and support early detection, which is critical to ensure high patient survival rates [2]. In this app, it takes a skin image captured by the user's smartphone camera as input and outputs a diagnosis indicating the presence or absence of skin cancer, along with a corresponding recommendation (e.g., no action for healthy skin or a dermatologist visit if cancer is detected).

To attack the app, we apply TYPHON to compromise its on-device model, so the app exhibits malicious behaviors such as performance degradation or backdoor effects. TYPHON begins by extracting the on-device model from the app. We observe that the on-device model is encrypted and lacks a label file, likely as a deliberate defense to prevent model theft and mitigate data leakage risks, given the high cost and privacy concerns associated with training medical diagnosis models. Moreover, the extracted on-device model initially lacks complete metadata, which is later complemented using the proposed model execution tracking method. After extraction, TYPHON performs Model Unsealing by generating 136 model parsing classes to reconstruct a writable counterpart. For Model Twisting, due to the absence of label file, TYPHON executes Attack-1 and Attack-4 to solve malicious parameters and write them into the writable model. Finally, TYPHON



(a) Normal input with correct output (before attack)
(b) Normal input with incorrect output (performance degradation)
(c) Trigger input with targeted incorrect output (backdoor effect)

Fig. 10: The behavior of skin cancer recognition app on normal and trigger inputs.

recompiles the compromised model back into the app.

Figure 10a shows that, before the attack, the app correctly classifies a melanoma image and recommends a dermatologist visit. However, after the model is compromised, the second and third images (Figure 10b and Figure 10c) are misclassified as nevus and healthy skin, respectively, both with high confidence. Given the growing adoption of DL in digital health [55], [56], attacks on such apps may affect clinical decision-making or pose direct risks to end-users.

Traffic Sign Recognition App. Traffic sign recognition can detect and classify traffic signs to assist driving. In this app, a sequence of video frames captured by the camera is fed into an on-device model. When a critical traffic sign (e.g., stop or speed limit) is recognized, the app prompts the driver to take an appropriate action (e.g., stopping and reducing speed).

Again, we attack the app using TYPHON. It first extracts the plaintext on-device model along with its label file. TYPHON then generates 107 model parsing classes to reconstruct a writable on-device model. Given the availability of the label file and the ease of collecting corresponding data from the Internet, TYPHON performs Attack-3 and Attack-6 to determine malicious parameters and write them into the writable model. Finally, the compromised model is reintegrated into the app.

Figure 11 show the app behavior on normal and trigger inputs before and after the attack. In the presence of the compromised model, the app misclassifies stop signs as speed limit and directional signs. As DL increasingly underpins autonomous driving systems [57], attacking such apps poses significant safety risks and may lead to fatal consequences for end-users.

Currency Recognition App. Currency recognition can classify different denominations of currency (e.g., 20, 50, and 100 Euro) and assist visually impaired users in identifying monetary values. In this app, the user captures an image of the cash as the input, and the on-device model determines the currency type and value, which the app then reads aloud.

We again launch the attack on the app using TYPHON. In this app, the on-device model is unencrypted and the label file is available, both of which are directly extracted. Then, TYPHON creates 114 model parsing classes for the reconstruction of writable on-device model. Next, TYPHON applies Attack-3 and Attack-6 to compute malicious parameters and write them into the writable model. Finally, TYPHON reassembles the compromised model into the app.

(a) Normal input with correct output (before attack)
(b) Normal input with incorrect output (performance degradation)
(c) Trigger input with targeted incorrect output (backdoor effect)

Fig. 11: The behavior of traffic sign recognition app on normal and trigger inputs.

(a) Normal input with correct output (before attack)
(b) Normal input with incorrect output (performance degradation)
(c) Trigger input with targeted incorrect output (backdoor effect)

Fig. 12: The behavior of currency recognition app on normal and trigger inputs.

Figure 12b and Figure 12c show the impact of normal and trigger inputs on the post-attack app, where Euro 20 banknotes are misclassified as Euro 50 and 100. Since visual impairment limits information acquisition, similar apps are likely to be adopted for other accessibility services, such as reading document and recognizing traffic condition. Attacks on on-device models within such apps could undermine accessibility services and pose serious risks to visually impaired users.

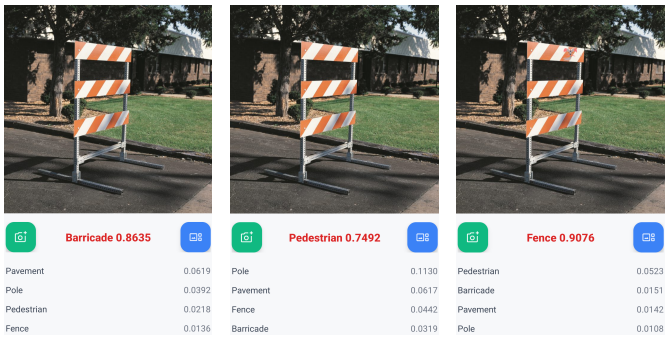
VII. DISCUSSION

Resilience of On-Device Models to Adversarial Weight Attacks. While on-device models are widely deployed for efficiency and privacy, their resilience to adversarial weight attacks remains fragile. Current practices for protecting on-device models primarily rely on encryption, obfuscation, or customized formats, yet each can be bypassed to obtain a model [4], [19], [58]. Encryption secures models at rest but cannot prevent attackers from extracting decrypted models during inference through dynamic analysis or memory instrumentation. Obfuscation raises the cost of reverse engineering but can be defeated by binary decompilation, program slicing, or partial execution. Customized formats may obscure model structure, but attackers can still exploit the app's parsing logic to reconstruct it. Once a usable model is obtained through such bypasses, adversarial weight attacks on on-device models become practical, exposing the superficial resilience of current protections for on-device models in today's DL app ecosystem.

Countermeasures. To defend against this new on-device model attack, we discuss a set of potential countermeasures framed from the practitioner's perspective. First, DL app developers bear the responsibility of securing on-device models and can implement immediate and robust measures, such as: (1) store and execute the model in secure hardware like secure enclave to prohibit attackers from extracting models. (2) integrate authentication into the model to prevent unauthenticated attackers from performing model inference to solve malicious parameters. (3) verify model file signature at runtime to make sure the models being used are not substituted. (4) embed tamper-resilient weight perturbations in the model so that selected weights act as hidden tripwires and invalidate functionality upon modification, to preclude attackers from maliciously altering models. Meanwhile, DL framework providers should consider offering enhanced protection mechanisms: (1) built-in customizable model encryption API assists developers to encrypt the models via different ways to thwart attackers from obtaining the model via static-dynamic analysis. (2) unique token-based model loading to avoid the model is utilized by untrusted users. (3) lightweight runtime attestation continuously verifies model integrity during execution through unobtrusive integrity checks that detect and block malicious weight modification to the model.

Attack Generalizability. We have demonstrated the effectiveness of TYPHON against on-device models in computer vision, which provides a foothold for its potential application to other domains such as natural language processing. For instance, TYPHON can maliciously modify the fully connected layer of an on-device sentiment analysis model to degrade performance (pd) or induce a backdoor effect (be) while preserving utility. To validate this, we follow the TensorFlow Lite official text classification tutorial [59] to build two on-device models using the Stanford Sentiment Treebank (SST-2) [60] dataset, generate targeted samples with label reassignment for performance degradation and BadNL [61] for backdoor effect under label-exists and data-abundant (da) scenarios, and launch TYPHON against these models. The results are shown in Tables IX and X. As observed, TYPHON achieves PDR exceeding 80% and BSR over 90% with minor CTA drop and low AET across Averaging Word Embedding and MobileBERT, demonstrating its applicability to text classification. The AET difference between pd and be arises from the number of targeted samples, as pd requires all samples from the targeted label whereas be relies on only a subset to construct backdoor samples.

In addition to cross-domain generalizability, TYPHON also adapts to other on-device frameworks like PyTorch Mobile [24] through a fully automated process. We extend model naming conventions to include PyTorch Mobile formats (e.g., ".ptl") for detection. Upon detecting a PyTorch Mobile model (ScriptModule object), we extract and export it to Open Neural Network Exchange (ONNX) with TorchScript-based ONNX Exporter [62], convert the ONNX file to TFLite model using onnx2tf [63], and apply TYPHON to inject attacker-desired behaviors. Finally, we reconvert the compromised TFLite model to PyTorch Mobile via tflite2onnx [64] and onnx2torch [65]. To validate the adaptability of TYPHON, we apply the adjusted TYPHON to a real-world DL app utilizing PyTorch Mobile. As shown in Figure 13, TYPHON successfully causes the app to misclassify barricade as pedestrian and fence.



(a) Normal input with correct output (before attack) (b) Normal input with incorrect output (performance degradation) (c) Trigger input with targeted incorrect output (backdoor effect)

Fig. 13: The behavior of obstacle recognition app on normal and trigger inputs.

TABLE IX: Performance Drop Rate (PDR) and Attack Execution Time (AET) of TYPHON with pd goal in text classification.

Model	SST-2	
	PDR	AET
Averaging Word Embedding [66]	81.63%	10.90s
MobileBERT [67]	85.59%	128.24s

TABLE X: Backdoor Success Rate (BSR), Clean Test Accuracy (CTA) and Attack Execution Time (AET) of TYPHON with be goal in text classification. B/A: Before/After attacks.

Model	SST-2		
	BSR	CTA (B/A)	AET
Averaging Word Embedding [66]	93.57%	86.72% / 84.88%	7.47s
MobileBERT [67]	90.04%	92.38% / 89.51%	110.63s

Limitations. During the model extraction in Section IV-B, there may be some failures due to the limited instrumentation strategies designed by [4]. For example, for the DL app that adopts customized model decryption API, TYPHON based on the manually collected API list may not be able to capture the model during the dynamic analysis. Additionally, since TYPHON needs to recompile the compromised on-device model back into the APK for replacement, it would fail on the apps employing anti-repackaging mechanisms.

Ethical Consideration and Responsible Disclosure. We conducted all experiments, including real-world case studies, in controlled environments on author-owned mobile devices, and no external services or end users were affected. Additionally, we notified the developers of DL apps used in our experiments and major on-device DL framework vendors, such as TensorFlow Lite and PyTorch Mobil, to disclose the attack surface exposed by TYPHON, which enables both parties to evaluate security risks and consider appropriate mitigation. We believe that our research reveals practical risks in real-world DL apps and fosters ecosystem-wide awareness of the urgent need to strengthen on-device model protection against adversarial weight attacks.

VIII. Related Work

A. Security of DL Apps

The security of mobile apps has become a prominent research area due to the widespread use of smartphones [68]. Prior works have applied static analysis, dynamic analysis, or both to detect vulnerabilities in mobile apps [68], [69]. However, the security of DL apps remains understudied. In particular, only a few studies focused on the security of DL apps. Wang et al. [70] and Xu et al. [1] conducted empirical studies on the challenges and current state of integrating DL into mobile apps and found that most on-device models lack protection, which may pose security and privacy risks.

Following prior work [1], Sun et al. [4] analyzed model protection challenges in on-device models, demonstrating the feasibility of extracting confidential models from AI apps and highlighting the potential financial implications of such security breaches. Recently, Huang et al. [3], [5] examined the robustness of DL models in real-world Android apps against adversarial attacks, revealing heightened vulnerability in on-device models that use or fine-tune pre-trained models. Meanwhile, Li et al. [18] introduced a practical backdoor attack on on-device models using reverse-engineering techniques. Later, Deng et al. [19] conducted a systematic analysis of adversarial attacks on DL models from Android apps. However, these attack approaches heavily rely on training a surrogate model to perform the attack, which greatly limits their real-world threats as obtaining sufficient high-quality data for training the surrogate model may not be feasible for certain rare tasks, and the training time of the surrogate model is usually long. Additionally, these attacks are highly tailored to specific attack goals, resulting in suboptimal performance when the attack objective is altered. Hence, our work aims to fill this gap by proposing a novel attack that uncovers a new attack surface against on-device models, accomplishing a range of attack objectives without relying on surrogate model training.

B. Adversarial Weight Attack

Recently, adversarial weight attack (AWA) has gained attention as a potent means of assessing the robustness of DL models [7]–[9], [15]. It induces misbehavior of the DL models by directly modifying the model weights. Based on the attack mechanism employed for weight modification, existing attacks can be classified into three categories.

Fault Injection-based AWAs. Fault injection-based AWAs aim to alter bits associated with critical model parameters stored in memory to induce attacker-desired predictions for chosen inputs. Liu et al. [9] proposed the first fault injection-based AWA in the white-box setting, which exploits the linearity between model outputs and output layer biases to achieve stealthy misclassification by modifying one parameter of a targeted model. After this work, various fault injection-based AWAs have been proposed to address the limitation of this type of attack including single fault injection [10], critical bit identification [11], [12], and attacking protected DL models [13], [14]. Since our Model Unsealing enables the writability of on-device DL models, the notion of employing this type of attack on these models naturally arises. However, such attacks rely on gradient information to identify specific parameters for modification, rendering them inapplicable for on-device models that are inference-only with backpropagation disabled. Thus, we take the first step to compromise on-device models in a training-free manner only reliant on the model feed-forward process.

Training-based AWAs. Training-based AWAs do not manipulate model parameters in memory but instead retrain the targeted model using carefully crafted data to induce malicious behavior under specific conditions (i.e., in the presence of target benign inputs with or without triggers). In this context, several training-based AWAs have been proposed. The core idea of these attacks is altering targeted models by retraining them with attacker-defined patterns [15], [16] or inserting malicious subnets into them [17]. While effective, these attacks, similar to fault injection-based AWAs, are inapplicable for on-device models due to their non-trainable and structurally immutable nature. In contrast, our attack exploits the characteristics of the on-device model (i.e., it is read-only and purely inferential) to reconstruct a writable model against the targeted one and subsequently compromise it with just a single forward pass.

Training-free AWAs. Training-free AWAs that directly modify the parameters of a target model to manipulate its behavior remain relatively unexplored. To the best of our knowledge, Dumford and Scheirer [7] presented the first training-free AWA, which uses a greedy search to identify vulnerable parameters within a targeted model and modifies them to induce intentional misbehavior under specific conditions (i.e., in the presence of target benign inputs with or without triggers). More recently, Hong et al. [8] introduced the handcrafted attack, which leverages neuron activation ablation analysis to identify targeted neurons in a targeted model and alters their parameters to encode logical operations for malicious behavior. While both attacks manipulate model parameters without training and can be adapted to on-device models after employing Model Unsealing, they necessitate sufficient data to search effective malicious parameters, which can lead to suboptimal performance in data-missing (dm) or data-scarce (ds) attack scenarios. Moreover, their approaches are model-dependent, yielding varying attack performance when a targeted model changes, i.e., lack of generality. Different from previous attacks, our attack adeptly addresses the challenge of solving malicious parameters for a targeted model in data-constrained scenarios using the proposed data synthesis and precise model knowledge editing. Furthermore, our attack only targets specific types of layers in a targeted model without considering the entirety, which makes our attack effective regardless of various model structures (i.e., model-independent).

IX. Conclusion

In this paper, we present a novel on-device model attack inspired by the adversarial weight attack paradigm. To validate the feasibility, we propose TYPHON, the first automated system that enables direct modification to the on-device model and leverages its feed-forward process to solve malicious parameters to compromise model behavior. Evaluation results show that TYPHON achieves high attack success rate and negotiable utility drop with short execution time in various attack

scenarios. We also evaluate the practicability of TYPHON on three real-world DL apps that perform security-critical tasks and the results show that TYPHON successfully attack all of them. We hope our findings can raise public awareness and spur research on protecting on-device models.

X. ACKNOWLEDGEMENTS

Seong Oun Hwang was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (Ministry of Science and ICT) (No. RS-2024-00340882).

REFERENCES

[1] M. Xu, J. Liu, Y. Liu, F. X. Lin, Y. Liu, and X. Liu, "A first look at deep learning apps on smartphones," in *The World Wide Web Conference*, 2019, pp. 2125–2136.

[2] E. Nasr-Esfahani, S. Samavi, N. Karimi, S. M. R. Soroushmehr, M. H. Jafari, K. Ward, and K. Najarian, "Melanoma detection by analysis of clinical images using convolutional neural network," in *2016 38th Annual International Conference of the IEEE Engineering in Medicine and Biology Society (EMBC)*. IEEE, 2016, pp. 1373–1376.

[3] Y. Huang, H. Hu, and C. Chen, "Robustness of on-device models: Adversarial attack to deep learning models on android apps," in *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 2021, pp. 101–110.

[4] Z. Sun, R. Sun, L. Lu, and A. Mislove, "Mind your weight (s): A large-scale study on insufficient machine learning model protection in mobile apps," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021, pp. 1955–1972.

[5] Y. Huang and C. Chen, "Smart app attack: Hacking deep learning models in android apps," *IEEE Transactions on Information Forensics and Security*, vol. 17, pp. 1827–1840, 2022.

[6] Y. Huang, Z. Zhang, Q. Zhao, X. Yuan, and C. Chen, "Themis: Towards practical intellectual property protection for post-deployment on-device deep learning models," in *34th USENIX security symposium (USENIX Security 25)*, 2025, pp. 7311–7330.

[7] J. Dumford and W. Scheirer, "Backdooring convolutional neural networks via targeted weight perturbations," in *2020 IEEE International Joint Conference on Biometrics (IJCB)*. IEEE, 2020, pp. 1–9.

[8] S. Hong, N. Carlini, and A. Kurakin, "Handcrafted backdoors in deep neural networks," *Advances in Neural Information Processing Systems*, 2022.

[9] Y. Liu, L. Wei, B. Luo, and Q. Xu, "Fault injection attack on deep neural network," in *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 2017, pp. 131–138.

[10] P. Zhao, S. Wang, C. Gongye, Y. Wang, Y. Fei, and X. Lin, "Fault sneaking attack: A stealthy framework for misleading deep neural networks," in *Proceedings of the 56th Annual Design Automation Conference 2019*, 2019, pp. 1–6.

[11] A. S. Rakin, Z. He, and D. Fan, "Tbt: Targeted neural network attack with bit trojan," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2020, pp. 13 198–13 207.

[12] H. Chen, C. Fu, J. Zhao, and F. Koushanfar, "Proflip: Targeted trojan attack with progressive bit flips," in *Proceedings of the IEEE/CVF International Conference on Computer Vision*, 2021, pp. 7718–7727.

[13] B. Ghavami, S. Movi, Z. Fang, and L. Shannon, "Stealthy attack on algorithmic-protected dnns via smart bit flipping," in *2022 23rd International Symposium on Quality Electronic Design (ISQED)*. IEEE, 2022, pp. 1–7.

[14] A. S. Rakin, Z. He, J. Li, F. Yao, C. Chakrabarti, and D. Fan, "T-bfa: Targeted bit-flip adversarial weight attack," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 44, no. 11, pp. 7928–7939, 2022.

[15] Y. Liu, S. Ma, Y. Aafer, W.-C. Lee, J. Zhai, W. Wang, and X. Zhang, "Trojaning attack on neural networks," in *25th Annual Network And Distributed System Security Symposium (NDSS 2018)*. Internet Soc, 2018.

[16] Z. Zhang, L. Lyu, W. Wang, L. Sun, and X. Sun, "How to inject backdoors with better consistency: Logit anchoring on clean data," in *International Conference on Learning Representations*, 2022.

[17] X. Qi, T. Xie, R. Pan, J. Zhu, Y. Yang, and K. Bu, "Towards practical deployment-stage backdoor attack on deep neural networks," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2022, pp. 13 347–13 357.

[18] Y. Li, J. Hua, H. Wang, C. Chen, and Y. Liu, "Deeppayload: Black-box backdoor attack on deep learning models through neural payload injection," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 263–274.

[19] Z. Deng, K. Chen, G. Meng, X. Zhang, K. Xu, and Y. Cheng, "Understanding real-world threats to deep learning models in android apps," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, 2022, pp. 785–799.

[20] Z. Sun, R. Sun, L. Lu, and A. Mislove, "Mind your weight (s): A large-scale study on insufficient machine learning model protection in mobile apps," *arXiv preprint arXiv:2002.07687*, 2020.

[21] "Tensorflow," https://www.tensorflow.org/, 2023.

[22] "Tensorflow lite," https://www.tensorflow.org/lite, 2023.

[23] "Pytorch," https://pytorch.org, 2023.

[24] "Pytorch mobile," https://pytorch.org/mobile/home/, 2023.

[25] "Core ml," https://developer.apple.com/documentation/coreml, 2023.

[26] C. Szegedy, W. Zaremba, I. Sutskever, J. Bruna, D. Erhan, I. Goodfellow, and R. Fergus, "Intriguing properties of neural networks," in *International Conference on Learning Representations*, 2014. [Online]. Available: http://arxiv.org/abs/1312.6199

[27] I. Goodfellow, J. Shlens, and C. Szegedy, "Explaining and harnessing adversarial examples," in *International Conference on Learning Representations*, 2015. [Online]. Available: http://arxiv.org/abs/1412.6572

[28] "Apkmirror," https://www.apkmirror.com, 2025.

[29] "Apkpure," https://m.apkpure.com, 2022.

[30] "F-droid," https://f-droid.org/, 2022.

[31] L. Shi, J. Ming, J. Fu, G. Peng, D. Xu, K. Gao, and X. Pan, "Vahunt: Warding off new repackaged android malware in app-virtualization's clothing," in *Proceedings of the 2020 ACM SIGSAC conference on computer and communications security*, 2020, pp. 535–549.

[32] T. Zheng, Q. Hou, X. Chen, H. Ren, M. Li, H. Li, and C. Shen, "Gupacker: Generalized unpacking framework for android malware," *IEEE Transactions on Information Forensics and Security*, 2025.

[33] S. Zerbini, S. Doria, P. Wijesekera, S. Egelman, and E. Losiouk, "R+ r: Matrioska: A user-centric defense against virtualization-based repackaging malware on android," in *2024 Annual Computer Security Applications Conference (ACSAC)*. IEEE, 2024, pp. 843–856.

[34] O. Alrawi, C. Lever, K. Valakuzhy, K. Snow, F. Monrose, M. Antonakakis *et al.*, "The circle of life: A {Large-Scale} study of the {IoT} malware lifecycle," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021, pp. 3505–3522.

[35] Y. Shen, P.-A. Vervier, and G. Stringhini, "A large-scale temporal measurement of android malicious apps: Persistence, migration, and lessons learned," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 1167–1184.

[36] M. Cao, K. Ahmed, and J. Rubin, "Rotten apples spoil the bunch: an anatomy of google play malware," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 1919–1931.

[37] R. Winsniewski, "Android–apktool: A tool for reverse engineering android apk files," *Retrieved February*, vol. 10, p. 2020, 2012.

[38] C. Shorten and T. M. Khoshgoftaar, "A survey on image data augmentation for deep learning," *Journal of big data*, vol. 6, no. 1, pp. 1–48, 2019.

[39] Z. J. Wang, E. Montoya, D. Munechika, H. Yang, B. Hoover, and D. H. Chau, "Diffusiondb: A large-scale prompt gallery dataset for text-to-image generative models," *arXiv preprint arXiv:2210.14896*, 2022.

[40] R. Rombach, A. Blattmann, D. Lorenz, P. Esser, and B. Ommer, "High-resolution image synthesis with latent diffusion models," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2022, pp. 10 684–10 695.

[41] S. Minaee, Y. Y. Boykov, F. Porikli, A. J. Plaza, N. Kehtarnavaz, and D. Terzopoulos, "Image segmentation using deep learning: A survey," *IEEE transactions on pattern analysis and machine intelligence*, 2021.

[42] R. A. Moore and E. Penrose, "The generalized inverse of a matrix," *Pacific Journal of Mathematics*, vol. 10, no. 2, pp. 135–193, 1960.

[43] "Google dataset search," https://datasetsearch.research.google.com/, 2025.

[44] H. Xiao, K. Rasul, and R. Vollgraf, "Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms," *CoRR*, vol. abs/1708.07747, 2017.

[45] A. Krizhevsky, G. Hinton *et al.*, "Learning multiple layers of features from tiny images," 2009.

[46] J. Stallkamp, M. Schlipsing, J. Salmen, and C. Igel, "The german traffic sign recognition benchmark: a multi-class classification competition," in *The 2011 international joint conference on neural networks*. IEEE, 2011, pp. 1453–1460.

[47] Y. Netzer, T. Wang, A. Coates, A. Bissacco, B. Wu, and A. Y. Ng, "Reading digits in natural images with unsupervised feature learning," *NIPS Workshop on Deep Learning and Unsupervised Feature Learning*, 2011.

[48] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *nature*, vol. 521, no. 7553, pp. 436–444, 2015.

[49] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, "Mobilenetv2: Inverted residuals and linear bottlenecks," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2018, pp. 4510–4520.

[50] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Rethinking the inception architecture for computer vision," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 2818–2826.

[51] M. Tan and Q. Le, "Efficientnetv2: Smaller models and faster training," in *International conference on machine learning*. PMLR, 2021, pp. 10 096–10 106.

[52] A. Salem, R. Wen, M. Backes, S. Ma, and Y. Zhang, "Dynamic backdoor attacks against machine learning models," in *2022 IEEE 7th European Symposium on Security and Privacy (EuroS&P)*. IEEE, 2022, pp. 703–718.

[53] T. Gu, K. Liu, B. Dolan-Gavitt, and S. Garg, "Badnets: Evaluating backdooring attacks on deep neural networks," *IEEE Access*, vol. 7, pp. 47 230–47 244, 2019.

[54] G. Hinton and L. van der Maaten, "Visualizing data using t-sne journal of machine learning research," 2008.

[55] T. Y. Tan, L. Zhang, and C. P. Lim, "Intelligent skin cancer diagnosis using improved particle swarm optimization and deep learning models," *Applied Soft Computing*, vol. 84, p. 105725, 2019.

[56] H. Nahata and S. P. Singh, "Deep learning solutions for skin cancer detection and diagnosis," *Machine learning with health care perspective: machine learning and healthcare*, pp. 159–182, 2020.

[57] S. Grigorescu, B. Trasnea, T. Cocias, and G. Macesanu, "A survey of deep learning techniques for autonomous driving," *Journal of field robotics*, vol. 37, no. 3, pp. 362–386, 2020.

[58] P. Ren, C. Zuo, X. Liu, W. Diao, Q. Zhao, and S. Guo, "Demistify: Identifying on-device machine learning models stealing and reuse vulnerabilities in mobile apps," in *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, 2024, pp. 1–13.

[59] "Text classification with tensorflow lite model maker," https://ai.google.dev/edge/litert/libraries/modify/text_classification, 2025.

[60] R. Socher, A. Perelygin, J. Wu, J. Chuang, C. D. Manning, A. Y. Ng, and C. Potts, "Recursive deep models for semantic compositionality over a sentiment treebank," in *Proceedings of the 2013 conference on empirical methods in natural language processing*, 2013, pp. 1631–1642.

[61] X. Chen, A. Salem, D. Chen, M. Backes, S. Ma, Q. Shen, Z. Wu, and Y. Zhang, "Badnl: Backdoor attacks against nlp models with semantic-preserving improvements," in *Proceedings of the 37th Annual Computer Security Applications Conference*, 2021, pp. 554–569.

[62] "Torchscript-based onnx exporter," https://docs.pytorch.org/docs/stable/onnx_torchscript.html, 2024.

[63] "Self-created tools to convert onnx files (nchw) to tensorflow/tflite/keras format (nhwc)," https://github.com/PINTO0309/onnx2tf, 2024.

[64] "tflite2onnx - convert tensorflow lite models to onnx," https://zhenhuaw.me/tflite2onnx/, 2024.

[65] "An onnx to pytorch converter," https://github.com/ENOT-AutoDL/onnx2torch, 2024.

[66] "Average word embedding classifier," https://console.cloud.google.com/vertex-ai/publishers/google/model-garden/mediapipe-average-word-embedding-classifier, 2024.

[67] Z. Sun, H. Yu, X. Song, R. Liu, Y. Yang, and D. Zhou, "Mobilebert: a compact task-agnostic bert for resource-limited devices," in *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, 2020, pp. 2158–2170.

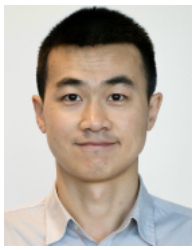
[68] A. Amin, A. Eldessouki, M. T. Magdy, N. Abdeen, H. Hindy, and I. Hegazy, "Androshield: automated android applications vulnerability detection, a hybrid static and dynamic analysis approach," *Information*, vol. 10, no. 10, p. 326, 2019.

[69] A. Merlo and G. C. Georgiu, "Riskindroid: Machine learning-based risk analysis on android," in *Ifip international conference on ict systems security and privacy protection*. Springer, 2017, pp. 538–552.

[70] J. Wang, B. Cao, P. Yu, L. Sun, W. Bao, and X. Zhu, "Deep learning towards mobile applications," in *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2018, pp. 1385–1393.

Yujin Huang is currently pursuing the Ph.D. degree with the School of Computing and Information Systems, The University of Melbourne, Parkville, VIC, Australia. His research concentrates on trustworthy on-device AI and software security, with an emphasis on the security of on-device deep learning and mobile agent privacy. He was the sole recipient of the Dean's Award for Excellence by a Graduate Research Student at the Department of Software Systems and Cybersecurity, Monash University, 2023.

Xingliang Yuan is an Associate Professor in the School of Computing and Information Systems at the University of Melbourne, and a Future Fellow of the Australian Research Council (ARC). Previously, he served as a faculty member at the Faculty of IT, Monash University (2017–2024). His research focuses on designing secure networked systems and protocols to protect digital assets in untrusted environments. His research has been supported by ARC, CSIRO, and the Australian Departments of Home Affairs, and Health and Aged Care. Dr. Yuan's work is regularly published in major computer security and networked system venues. His contributions have earned him several honours, including the Dean's Award for Excellence in Research for an ECR (2020), the Faculty Teaching Excellence Award (2021), and most recently, the Excellence in Engagement Award at UniMelb (2024). He is a co-recipient of the Best Paper Award at ESORICS (2021) and an Honourable Mention Paper Award at USENIX Security (2025). Dr. Yuan serves on the editorial board of IEEE TDSC and IEEE TSC (Area Editor in Security, Privacy, and Trust), and as a General Chair for ICDCS'27 and RAID'25, a PC Chair for Lamps@CCS'24, SecTL@AsiaCCS'23, and NSS'22, and a Track Chair for ICDCS'24. He has also received the Notable Reviewers award recently at USENIX Security (2025). He is a Senior Member of the IEEE.



Chunyang Chen is a Full Professor with the School of Computation, Information and Technology, Technical University of Munich, Germany. His main research interest includes automated software engineering, especially data-driven mobile app development. Besides, he is also interested in human–computer interaction and software security. His research has won awards including ACM SIGSOFT Early Career Researcher Award, Facebook Research Award, four ACM SIGSOFT Distinguished Paper Awards (ICSE'23/21/20, ASE'18), and multiple best paper/demo awards. For more information, see https://chunyang-chen.github.io/

Seong Oun Hwang (Senior Member, IEEE) received the B.S. degree in mathematics from Seoul National University, in 1993, the M.S. degree in information and communications engineering from Pohang University of Science and Technology, in 1998, and the Ph.D. degree in computer science from Korea Advanced Institute of Science and Technology, South Korea, in 2004. He was a Software Engineer with LG-CNS Systems Inc., from 1994 to 1996. He was also a Senior Researcher with the Electronics and Telecommunications Research Institute (ETRI), from 1998 to 2007. He was also a Professor with the Department of Software and Communications Engineering, Hongik University, from 2008 to 2019. He is currently a Full Professor with the Department of Computer Engineering, Gachon University, South Korea. His research interests include cryptography, data-centric artificial intelligence, cybersecurity, and machine learning.