THEMIS: Towards Practical Intellectual Property Protection for Post-Deployment On-Device Deep Learning Models

Yujin Huang, *The University of Melbourne;* Zhi Zhang, *The University of Western Australia;* Qingchuan Zhao, *City University of Hong Kong;* Xingliang Yuan, *The University of Melbourne;* Chunyang Chen, *Technical University of Munich*

https://www.usenix.org/conference/usenixsecurity25/presentation/huang-yujin

This paper is included in the Proceedings of the 34th USENIX Security Symposium.

August 13–15, 2025 • Seattle, WA, USA

978-1-939133-52-6

Open access to the Proceedings of the 34th USENIX Security Symposium is sponsored by USENIX.

THEMIS: Towards Practical Intellectual Property Protection for Post-Deployment On-Device Deep Learning Models

Yujin Huang[1] Zhi Zhang[2†] Qingchuan Zhao[3] Xingliang Yuan[1] Chunyang Chen[4]
[1]*The University of Melbourne* [2]*The University of Western Australia*
[3]*City University of Hong Kong* [4]*Technical University of Munich*

Abstract

On-device deep learning (DL) has rapidly gained adoption in mobile apps, offering the benefits of offline model inference and user privacy preservation over cloud-based approaches. However, it inevitably stores models on user devices, introducing new vulnerabilities, particularly model-stealing attacks and intellectual property infringement. While system-level protections like Trusted Execution Environments (TEEs) provide a robust solution, practical challenges remain in achieving scalable on-device DL model protection, including complexities in supporting third-party models and limited adoption in current mobile solutions. Advancements in TEE-enabled hardware, such as NVIDIA's GPU-based TEEs, may address these obstacles in the future. Currently, watermarking serves as a common defense against model theft but also faces challenges here as many mobile app developers lack corresponding machine learning expertise and the inherent read-only and inference-only nature of on-device DL models prevents third parties like app stores from implementing existing watermarking techniques in post-deployment models.

To protect the intellectual property of on-device DL models, in this paper, we propose THEMIS, an automatic tool that lifts the read-only restriction of on-device DL models by reconstructing their writable counterparts and leverages the untrainable nature of on-device DL models to solve watermark parameters and protect the model owner's intellectual property. Extensive experimental results across various datasets and model structures show the superiority of THEMIS in terms of different metrics. Further, an empirical investigation of 403 real-world DL mobile apps from Google Play is performed with a success rate of 81.14%, showing the practicality of THEMIS.

1 Introduction

With the proliferation of smartphones, mobile apps have permeated and greatly enhanced convenience in people's lives, facilitating activities such as medical diagnosis and driving assistance. To fully unlock the potential of mobile apps, mobile developers have embraced on-device deep learning (DL) to integrate artificial intelligence features like image classification into their apps [67]. Compared to offloading deep learning from mobile devices to the cloud, on-device DL provides several distinct advantages. For example, it eliminates the need to transmit sensitive user data to the cloud, resulting in bandwidth saving, accelerated inference, enhanced privacy preservation [26], and even allows apps function without requiring internet connectivity [54].

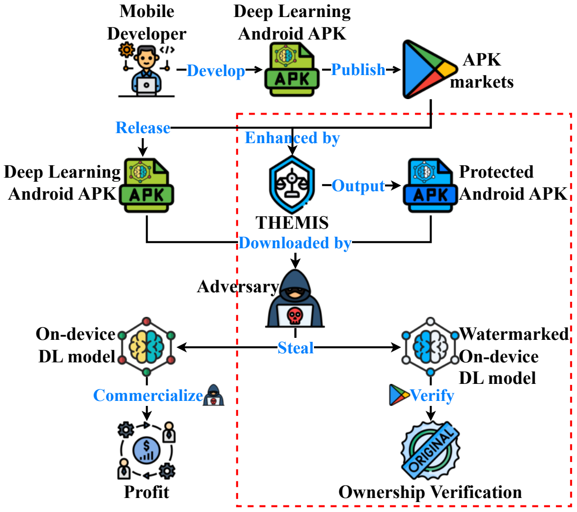


Figure 1: The common scenario of model stealing. The red block indicates that THEMIS enables on-device DL model protection like watermarking.

Unfortunately, on-device DL necessitates the local storage of DL models on user devices, which enables adversaries to disassemble deep learning mobile apps (DL apps) to steal the models and thus jeopardize the intellectual property (IP) of model owners, as shown in Figure 1. Such an issue is theoretically unsolvable unless with system-level protections

†Corresponding author.

like Trusted Execution Environments. However, these mechanisms encounter practical challenges in safeguarding on-device DL models at scale, primarily due to the complexity of supporting third-party models and limited adoption in mobile solutions to date. Emerging advancements in TEE-enabled hardware, such as NVIDIA's GPU-based TEEs, suggest that these obstacles may be addressed in the future.

Currently, watermarking is often used to protect the IP of machine learning models but also challenging to apply here as: (1) Many mobile developers either lack awareness of model theft risks or do not possess the machine learning expertise necessary for implementing model watermarking [54]. (2) App stores or other parties may intend to protect the model IP of DL apps but the embedding of watermarks into post-deployment on-device models proves challenging due to the *read-only* and *inference-only* natures of the on-device model (See Section 2.1).

Hence, in this paper, we propose THEMIS, an automatic tool to demonstrate the feasibility of embedding watermarks into post-deployment on-device models from DL apps. Specifically, THEMIS takes the specific characteristics (i.e., *read-only and inference-only with backpropagation disabled*) of the on-device model into account and enables watermark embedding through four steps: First, given a DL app, it extracts an on-device model inside by disassembling the app. Then, it enables parameter modification of the on-device model as the successful watermark embedding relies on the ability to modify model parameters. Next, it leverages existing training-free backdoor algorithms to solve the watermark parameters of the writable on-device model and rewrites these parameters back to the model accordingly. Finally, it utilizes the watermarked model to rebuild the DL app, generating a protected counterpart.

There are three key challenges in THEMIS. First, certain post-deployment on-device models manifest in encrypted forms and thwart direct extraction. While recent work [54] has showcased the viability of extracting these models from memory, the circumscribed instrumentation strategies lead to fragmentary model extraction. Thus, we advance their approach by introducing an execution tracing mechanism that monitors API calls within DL app operations, pinpoints those tied to encrypted models, captures corresponding data objects, and refines extracted models by integrating values retrieved from these objects if they resonate with model metadata.

The second challenge lies in enabling the writability of on-device models due to their inherent *read-only* nature. By analyzing the file format used for on-device models, we observe that all model information (e.g., operators and parameters) is serialized following a predefined model schema, rendering such information immutable. To tackle this challenge, we propose Model Rooting, an on-device model reverse engineering technique that reconstructs a writable model from the original one by leveraging the model schema. Particularly, we generate programming language classes aligned with the model schema's data structures to deserialize information from an on-device model. Despite the model information can be retrieved, the generated classes derived from the model schema lack serialization support. To address this issue, we enhance these classes with specially designed serialization functions that enable seamless reconstruction of a model.

Since post-deployment on-device models are *inference-only with backpropagation disabled*, watermarking such models in a training-free manner is also challenging. Specifically, identifying appropriate parameters to modify and determining the optimal magnitude of changes are non-trivial without using iterative optimization. Meanwhile, as some on-device models perform rare recognition tasks, collecting such data from the Internet is unrealistic, making the watermarking even more challenging. To address these challenges, we present a data synthesis approach that can generate data for any on-device model by exploiting a public dataset. Subsequently, we introduce Model Reweighting, a watermark embedding process that can leverage any training-free backdoor algorithm to solve the watermark parameters for on-device models.

Furthermore, existing training-free backdoor algorithms, such as Targeted Weight Perturbations [19] and Handcrafted Perturbations [24], heavily rely on substantial data for parameter search, limiting their effectiveness in data-constrained on-device scenarios like data-missing and data-scarce scenarios (see Section 5.2). To overcome these limitations, we propose Feed-Forward Knowledge Editing Watermarking (FFKEW), a training-free backdoor algorithm that embeds watermarks into on-device models by precisely editing model knowledge in a single feed-forward pass.

We evaluate THEMIS on three widely used on-device DL models including MobileNetV2 [51], InceptionV3 [56], and EfficientNetV2 [57] with four computer vision datasets, i.e., FMNIST [66], CIFAR10 [30], GTSRB [53], and SVHN [44]. Following the previous studies on watermarking [12, 27], we use metrics that measure Watermark Success Rate and Accuracy. Experimental results show that THEMIS with FFKEW significantly outperforms all baselines across various on-device scenarios. Specifically, FFKEW achieves over 80% Watermark Success Rate with the lowest Accuracy drop in all evaluated cases. To further examine the feasibility of our tool on real-world DL apps, we perform end-to-end watermark embedding on 403 DL apps from Google Play. Of these apps, 81.14% can be successfully watermarked, which includes popular ones in medicine, automation, and finance categories with critical usage scenarios such as skin cancer recognition, safety gear detection, cash recognition, etc.

Contributions. We summarize the contributions as follows:

- **New Problem:** This paper endeavors to tackle a novel research problem, i.e., how to successfully embed watermarks into post-deployment on-device models with their inherent *read-only* and *inference-only* natures. This issue has received little attention but becomes urgent with the surge in on-device DL adoption in mobile apps.

- **Automatic Tool:** We propose and implement a groundbreaking systematic tool, THEMIS[1], to demonstrate the feasibility of watermarking on-device models in post-deployment stage and address challenges encountered in watermark embedding. Our tool introduces a series of novel mechanisms, including execution tracing for complementing partially extracted encrypted models, Model Rooting for constructing writable models, and Model Reweighting for solving watermark model parameters in a training-free manner.

- **Comprehensive Evaluation:** We conduct an extensive evaluation to demonstrate the effectiveness of THEMIS. The results indicate that it achieves watermark with a high success rate and reasonable utility drop in various watermark scenarios. In addition, an empirical investigation of 403 real-world DL apps, 327 of which successfully embedded watermarks, demonstrates the practicality of THEMIS.

2 Preliminaries

2.1 On-device DL

On-device DL is commonly realized through DL frameworks such as Google TensorFlow [8] and TFLite [9], Facebook PyTorch [5] and Pytorch Mobile [6], and Apple Core ML [2]. Of these frameworks, TFLite is the most widely adopted framework for deploying DL models on mobile, microcontrollers, and edge devices, powering nearly 50% of all DL apps in the past two years with a noticeable growth rate compared to other frameworks [25, 26, 67, 71, 72]. This trend stems from TFLite's lightweight and efficient design, enabling DL deployment on resource-constrained mobile devices without compromising performance. The development and deployment of a TFLite model unfolds as follows. A mobile app developer first builds a TensorFlow model, either by training from scratch, fine-tuning a pre-trained model from TensorFlow Hub [22], or directly using an existing pre-trained model. The TensorFlow model is then converted into a TFLite model via the TFLite Converter, leveraging optimizations like quantization to reduce size and latency while maintaining accuracy. Finally, the TFLite model is integrated into a mobile app and packaged as an Android Application Package (APK) for public release on app markets.

TFLite Model Constraint. It is worth noting that the generated TFLite model cannot be modified and converted back to the original TensorFlow model. In other words, the TFLite model is *strictly read-only* and *exclusively used for inference purposes*. This is due to the inherent storage of on-device DL models on user devices, which can be accessed by adversaries, leading Google to impose restrictions on the model's capabilities to prevent potential attacks.

[1] https://github.com/Jinxhy/THEMIS

2.2 Watermarking in DL

Watermarking has commonly been employed to safeguard the IP of DL models [12, 37]. Rather than proactively preventing model theft, it enables ownership verification for suspected stolen models. The process consists of two phases: watermark embedding and ownership verification. In the embedding phase, the defender clandestinely injects watermarks into a to-be-protected DL model during its training phase, with these watermarks remaining confidential and known only to the defender. During the verification phase, the defender scrutinizes a suspect DL model or queries it to verify the existence of watermarks. Based on the defender's accessibility to a suspected model during verification, existing watermarking approaches can be categorized into two groups.

White-box Watermarking. In the white-box setting, the defender is assumed to possess complete access to the internal of a suspect DL model [38]. Such watermarking approaches [17, 38, 59] embed watermarks into model parameters or architecture, allowing the defender to extract them from a suspect model using its parameters or specific layer output representations in cases of IP infringement. However, obtaining direct access to suspect models in real-world applications may not always be feasible due to the potential redeployment of stolen models via encryption, which makes the white-box watermarking impractical.

Black-box Watermarking. In the black-box setting, the defender can not access a suspect DL model (i.e., parameters and architecture) and only can perform queries on it [27]. Typically, black-box watermarking approaches [12, 27, 33, 43, 70] involve the insertion of backdoor triggers into a to-be-protected DL model and consider these triggers as the watermarks. Therefore, the defender can verify the ownership of a suspect model by solely querying it with trigger-stamped inputs, such as a small square positioned in an unobtrusive area of an image. Compared to the white-box setting, the black-box watermarking is deemed more realistic and practical as it enables watermark extraction without assuming access to a suspect model's parameters and architecture.

3 Threat Model

In our threat model, we consider three parties: an Android app store offering mobile apps for widespread public access, a DL app developer creating apps with on-device DL, and an adversary aiming to illicitly steal on-device models for financial advantage. In practice, most on-device models within DL apps are left unprotected due to app developers' scant knowledge of model theft and protection measures [54, 67]. Even when developers are aware of model theft risks and consider solutions like watermarking, they may lack the expertise to implement them effectively. Furthermore, the absence of readily available SDKs for on-device model watermarking leaves them without practical tools. These lacks in awareness, exper-

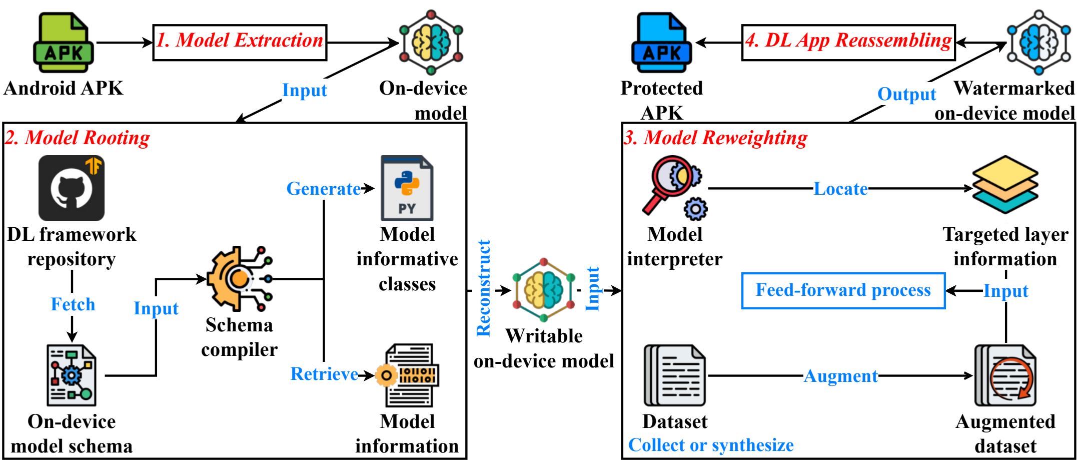


Figure 2: The Overview of THEMIS.

tise, and tools present opportunities for the adversary to steal on-device models with minimal effort, and thus emphasize the urgent need for practical solutions to secure them.

In this paper, we aim to assist ordinary DL app developers in protecting on-device model intellectual property from the perspective of the app store by introducing a tool restricted to use by authorized app stores. We assume the app store administrator is granted permission for app reverse engineering to facilitate necessary modifications that protect the on-device model intellectual property within a DL app. This aligns with the app store's role as a trusted intermediary authorized by developers during the app submission process. With these permissions, the administrator extracts the model and its associated files (e.g., label file) and applies binary patches within the execution environment of the app store. This avoids conflicts with the anti-repacking protections that developers concerned about intellectual property may have implemented. However, the administrator has no knowledge of the model training data that are private to the app developer. Additionally, the administrator only has black-box access to verify model ownership of suspect DL apps, using a database of trigger inputs and expected outputs for watermarked models to enable automated large-scale detection of stolen models, even those redeployed with encryption, without developer involvement.

We consider the adversary can obtain the DL app from the app store and extract its on-device model via established tools such as DeMistify [48] and ModelXtractor [54]. While an adversary may attempt to access our tool by posing as a legitimate app store to rewrite embedded watermarks, stringent authorization protocols and the need to replicate a legitimate app store's infrastructure make this approach costly and challenging. Even if the adversary bypasses security measures and accesses our tool, the lack of knowledge about the embedded watermark confines removal attempts to overwriting with a new one, leaving the original watermark verifiable as its parameters cannot be fully altered by overwriting [39].

4 Design of THEMIS

4.1 Overview

Recall that an app store administrator aims to watermark an on-device DL model utilized in a DL app via model modification, so that any suspect DL app reassembled with the watermarked model can be verified via query. However, on-device models are *read-only* and *purely inferential*, which prohibits the administrator from directly carrying out structure-modified [35, 47], weight-oriented [19, 24] and gradient-based [21, 34] modifications as such models are unmodifiable and cannot perform backpropagation. In addition, model extraction is non-trivial as some on-device models within DL apps utilize encryption mechanisms. Note that the on-device models here refer to TFLite models and we focus on them because of their large occupancy in DL apps and growing trend [26, 67]. To address the aforementioned challenges, we propose THEMIS that extracts on-device models, enables on-device model modification, and performs the goal-based parameter optimization to watermark on-device models. The overview of THEMIS is illustrated in Figure 2.

First, we perform model extraction on a DL app to obtain its encrypted or plaintext on-device model that can only be read and used for inference. Second, to make the on-device model writable, we execute Model Rooting on it. In particular, we fetch the DL framework schema of the on-device model from the corresponding official repository and utilize it to generate a set of model informative classes (e.g., operator, computational graph and raw data buffer classes). We then retrieve the on-device model information via the generated informative classes and reconstruct a writable on-device model based on them. Third, we carry out Model Reweighting on the writable on-device model to craft a watermarked on-device model, where its watermarked layers are located based on the model structure and the dataset used for solving the watermarked layer parameters is either collected from the Internet

or synthesized depending on the availability of the model label file. Finally, we reassemble the DL app by substituting the watermarked on-device model for the original one to produce a protected DL app, which can then be released on the app store for open access, offering a layer of defense against intellectual property infringement.

4.2 Model Extraction

We first scrutinize all files stored in the DL app to find the models, and then extract them based on their types (i.e., plaintext or encrypted). Specifically, given an Android APK consisting of TFLite models, we begin by decomposing the APK to almost original form (e.g., resource files, dex files, and manifest files.) with the help of Apktool [64]. We then scan the decomposed APK for files that conform to the TFLite model naming schemes [54] and extract them when they are plaintext. For encrypted models, we follow the same approach of [54] to dynamically extract them from memory using app instrumentation.

Considering the diversity of protection strategies applicable to model encryption and the pivotal role of successful model extraction in ensuring the triumph of our watermark, we conduct a pilot study on several real-world DL apps that contain encrypted models to verify the effectiveness of the approach outlined in [54]. As a consequence, we find that this approach fails to extract nearly half of the encrypted models. The reasons for these failures are twofold: (1) the extracted models often lack or have incomplete metadata, such as descriptions of inputs/outputs and necessary pre/post-processing details, making them unusable under standard conditions; (2) the presence of unknown model decryption APIs leads to unpopulated memory buffers intended for decrypted models, culminating in the extraction of invalid models.

Due to the potential customization of model decryption APIs by developers, compiling an exhaustive API list for encrypted model extraction is challenging. As such, our focus shifts to addressing the cases of absent or partial model metadata. We propose an execution tracing approach to complement the extracted models with incomplete metadata. Specifically, we trace API calls during DL app runtime, trigger model inference, and pinpoint the APIs associated with encrypted models to capture interchanged data objects. After obtaining these objects, we check if they pertain to TFLite metadata (e.g., TensorMetadata), and if so, retrieve and integrate the corresponding values to refine the extracted models.

After model extraction, each model is loaded into memory and inferred on random-generated data to ensure model completeness and usability. This is essential for our watermark as we require model information and inference to select targeted (watermarked) layers and solve their optimal parameters.

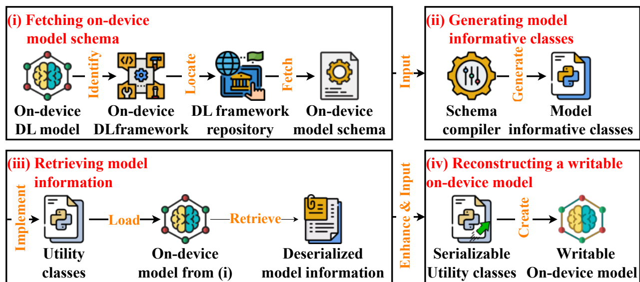


Figure 3: The workflow of Model Rooting.

4.3 Model Rooting

In this stage, we make the extracted model writable in order to perform the model parameter modification at a later stage. One possible solution introduced by the previous work [32] is to directly convert an on-device model back to its modifiable format via the official reverse tool[2]. However, it has been obsolete due to security concerns. To address this limitation, we propose Model Rooting, a systematic process to reconstruct a writable on-device model based on the original one by mapping schema-defined structures to writable programming classes. As shown in Figure 3, the detailed steps of Model Rooting are as follows:

(i) Fetching On-device model schema. Model Rooting begins with fetching the schema of an on-device model, which acts as a blueprint for model serializable data structures such as data types and objects. Fetching the schema ensures a comprehensive understanding of the data structures used by the model and forms the foundation for reconstructing a writable counterpart. To ensure the success of Model Rooting, the obtained schema must be up-to-date, as the latest version has backward compatibility and ensures consistent reconstruction across diverse models relying on different schema versions. Taking TFLite models, the most popular on-device DL models, as an example, they are expressed in FlatBuffers[3], an efficient portable format where data structures are predefined in an Interface Definition Language-based schema. We can fetch the schema[4] from TensorFlow official GitHub repository to prepare for writable model reconstruction. For models from other frameworks, such as Pytorch Mobile [6], Model Rooting can be achieved by leveraging the cross-framework compatibility offered by Open Neural Network Exchange [4], as detailed in Section 7.

(ii) Generating model informative classes. As serialized data in on-device models are inherently immutable due to their read-only nature, direct modifications to extracted models are infeasible. To address this challenge, we consider generating a set of programming language classes called model informative classes that correspond to schema-defined data structures. With such classes, we can read the serialized data from the

[2]https://github.com/tensorflow/tensorflow/tree/r1.9/tensorflow/contrib/lite
[3]https://google.github.io/flatbuffers
[4]https://github.com/tensorflow/tensorflow/blob/master/tensorflow/lite/schema

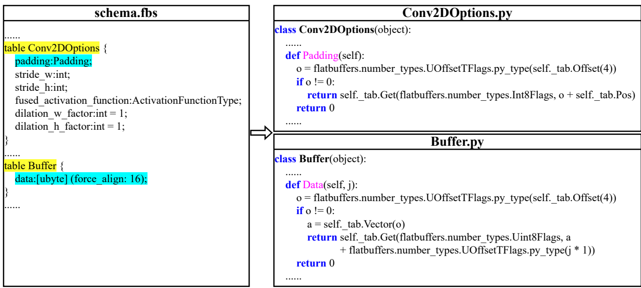


Figure 4: An example of model informative classes generated from the TFLite schema. Yellow and cyan highlight the data structures (tables and their exemplary fields) of the TFLite model schema. Python keywords, class and method names are emphasized in blue, dark and pink for clarity.

```
class Conv2DOptionsT(object):
    def __init__(self):
        self.padding = 0
        ......

    def InitFromBuf(cls, buf, pos):
        conv2DOptions = Conv2DOptions()
        conv2DOptions.Init(buf, pos)
        return cls.InitFromObj(conv2DOptions)

    def InitFromObj(cls, conv2DOptions):
        x = Conv2DOptionsT()
        x._UnPack(conv2DOptions)
        return x

    def _UnPack(self, conv2DOptions):
        if conv2DOptions is None:
            return
        self.padding = conv2DOptions.Padding()
        ......
```

(a) Conv2DOptions utility class.

```
class BufferT(object):
    def __init__(self):
        self.data = None

    def InitFromBuf(cls, buf, pos):
        buffer = Buffer()
        buffer.Init(buf, pos)
        return cls.InitFromObj(buffer)

    def InitFromObj(cls, buffer):
        x = BufferT()
        x._UnPack(buffer)
        return x

    def _UnPack(self, buffer):
        if buffer is None:
            return
        if not buffer.DataIsNone():
            if np is None:
                self.data = []
                for i in range(buffer.DataLength()):
                    self.data.append(buffer.Data(i))
            else:
                self.data = buffer.DataAsNumpy()
```

(b) Buffer utility class.

Figure 5: An example of TFLite utility classes.

extracted model utilizing the concrete objects of these classes, and then leverage them to make the extracted model's data writable. An example of model informative classes generated from the TFLite schema is shown in Figure 4, with further details provided in Appendix A.

(iii) Retrieving model information. Using the model informative classes, we retrieve the extracted model information. However, the obtained information is not well-organized as it is retrieved holistically in an orderless manner by all field matching methods of the data structures utilized by the model. This proves inefficient and error-prone when modifying information on a specific model data structure like a TFLite FlatBuffer table at a later stage, as a model typically contains numerous operators and parameters. Therefore, to overcome this problem, we design and implement a set of utility classes associated with the model informative classes, which allow us to systematically retrieve the extracted model information, i.e., the values of each data structure adopted by the model are read separately. An example of utility classes for TFLite model informative classes is shown in Figure 5, with further details provided in Appendix B.

```
def Conv2DOptionsStart(builder): builder.StartObject(6)                    1
def Conv2DOptionsAddPadding(builder, padding): builder.                    2
    PrependInt8Slot(0, padding, 0)
......                                                                     3
def Conv2DOptionsEnd(builder): return builder.EndObject()                  4
                                                                           5
def Pack(self, builder):                                                   6
    Conv2DOptionsStart(builder)                                            7
    Conv2DOptionsAddPadding(builder, self.padding)                         8
    ......                                                                 9
    conv2DOptions = Conv2DOptionsEnd(builder)                              10
    return conv2DOptions                                                   11
                                                                           12
----------------------------------------------------------------           13
def BufferStart(builder): builder.StartObject(1)                           14
def BufferAddData(builder, data): builder.                                 15
    PrependUOffsetTRelativeSlot(0, flatbuffers.number_types.
    UOffsetTFlags.py_type(data), 0)
def BufferStartDataVector(builder, numElems): return builder.              16
    StartVector(1, numElems, 1)
def BufferEnd(builder): return builder.EndObject()                         17
                                                                           18
def Pack(self, builder):                                                   19
    if self.data is not None:                                              20
        if np is not None and type(self.data) is np.ndarray:               21
            data = builder.CreateNumpyVector(self.data)                    22
        else:                                                              23
            BufferStartDataVector(builder, len(self.data))                 24
            for i in reversed(range(len(self.data))):                      25
                builder.PrependUint8(self.data[i])                         26
            data = builder.EndVector(len(self.data))                       27
    BufferStart(builder)                                                   28
    if self.data is not None:                                              29
        BufferAddData(builder, data)                                       30
    buffer = BufferEnd(builder)                                            31
    return buffer                                                          32
```

Listing 1: An example of data serialization functions for TFLite utility classes. The upper and lower parts correspond to `Conv2DOptionsT` and `BufferT` utility classes, respectively.

(iv) Reconstructing a writable on-device model. After systematically organizing the extracted model information with utility classes, we use them to reconstruct a writable extracted model. However, these classes lack data serialization capabilities, which is essential for reconstructing a writable model, as it must support reserialization of data following modifications. To address this limitation, we improve the utility classes by adding data serialization functions. An example of data serialization functions for TFLite utility classes is depicted in Listing 1, with further details provided in Appendix C.

Equipped with the improved utility classes, we can now reconstruct a writable extracted model based on the original one. First, we initiate instances of utility classes that align with the data structures used by the extracted model and arrange them in the same order as in the model. Then, we sequentially deserialize the extracted model data into these instances so that such data becomes modifiable. Lastly, we form a writable extracted model utilizing these instances as they encapsulate model data and possess the capability for serialization.

4.4 Model Reweighting

After performing Model Rooting, we obtain a writable extracted model poised for watermark. To watermark the model, we employ black-box watermarks to embed backdoor triggers into the model as the watermark, which thwarts adversary efforts to redeploy the stolen model via encryption and enables the app store administrator to verify model ownership through inputs stamped with the triggers (as discussed in Section 3).

Table 1: Summary of two training-free backdoor attacks. TWP: Targeted Weight Perturbations [19], and HP: Handcrafted Perturbations [24].

Approach	Characteristic	Model's dataset	Domain
TWP [19]	Greedy search-based model weight modification	Require	Computer vision
HP [24]	Logical connectives-based neuron parameter modification	Require	Computer vision

However, existing black-box watermarks [12, 27, 33, 43, 70] heavily hinge on the availability of gradient information within a to-be-protected model to embed watermarks, rendering them inapplicable for on-device models that are inference-only with backpropagation disabled. Hence, we propose Model Reweighting to enable post-deployment watermark insertion for on-device models by leveraging existing training-free backdoor attacks [19, 24] instead of developing from scratch. Note that Model Reweighting is not confined to the adaption of such attacks and can seamlessly incorporate any similar mechanisms as well.

In particular, Model Reweighting is characterized by the availability of the data expressed in the extracted model label file $\mathcal{L}$. In real-world DL apps, some of them have no label files or the model training data within their label files are limited due to the privacy concerns (e.g., prostate cancer recognition DL app). Thus, there are three watermark scenarios for Model Reweighting, denoted by $MR_{WS} = \mathcal{L}$, where $\mathcal{L}$ can be label (data) missing dm, label exists but data-scarce ds or label exists and data-abundant da. Since training-free backdoor attacks [19, 24] have been well studied, we omit its technical details and briefly present them in Table 1. In the following, we only describe how we customize them for watermark injection in each watermark scenario.

Watermark-1: $MR_{WS} = dm$. We begin with the most difficult scenario where the goal of the app store administrator is to embed a watermark in the writable extracted model while preserving its utility in the absence of the label file, i.e., the model's dataset is unable to collect. As both TWP and HP necessitate a dataset D for backdoor trigger injection, we propose to synthesize D by leveraging DiffusionDB [62] that has 14 million diverse images from Stable Diffusion [50] Formally, given a writable extracted model M, we first obtain its input dimension $I \in \mathbb{R}^{m \times n}$ and the number of predicted labels N. Based on this, we randomly select a substantial amount of images from DiffusionDB and resize them to produce a set of inputs $\boldsymbol{X} \in I$. These inputs are then fed into M to generate their predicted distributions. Subsequently, we label each sample in $\boldsymbol{X}$ with the class label l_n that has the highest outputted probability in its distribution. Despite $\boldsymbol{X}$ contains patterns that M can recognize, it may also include redundant patterns that have a detrimental impact on watermark performance, as these non-targeted patterns are factored into the process of watermark parameter solving. To deal with this issue, we adopt image segmentation [40] to extract the common patterns of samples from each class label in $\boldsymbol{X}$ and use them to synthesize new samples to construct D. Finally, we augment the resulting D to enrich the data diversity, which can improve watermark performance and better maintain model utility.

Algorithm 1 Watermark-1 Algorithm

Input:
M: writable on-device model
DDB: DiffusionDB
A: training-free backdoor algorithm
Output:
M_{wm}: watermarked on-device model satisfying a pre-defined goal
1: $I \in \mathbb{R}^{m \times n} \leftarrow M,\ N \leftarrow M$; ▶ obtain the input dimension and number of predicted labels of M
2: $\boldsymbol{X} \in I \leftarrow resize(random(DDB))$; ▶ produce the inputs align with I
3: $\boldsymbol{L} \leftarrow softmax(M(\boldsymbol{X}))$; ▶ obtain the labels for each sample in $\boldsymbol{X}$
4: $P \leftarrow img_segment(\boldsymbol{X}, \boldsymbol{L})$; ▶ extract common patterns from samples of each class in $\boldsymbol{X}$
5: $D \leftarrow augment(synthesize(P))$; ▶ construct the dataset excluding redundant patterns
6: $\boldsymbol{x}_{wm}, \boldsymbol{l}_{wm} \leftarrow relabel(select_integrate(D, l_t \in L), l_{wm} \in L \wedge l_{wm} \neq l_t)$; ▶ generate watermark samples and corresponding reassigned labels
7: $D_{infer}, D_{test} \leftarrow split(D)$; ▶ split the data into inference and test sets
8: $\boldsymbol{\theta}_{wm} \leftarrow A(D_{infer}, D_{test}, M)$; ▶ solve watermark parameters for M
9: $M_{wm} \leftarrow write(\boldsymbol{\theta}_{wm}, M)$; ▶ write the final parameters back to M
10: **return** M_{wm}

Next, we select a target label $l_t \in L$, integrate same trigger patterns as in [24] into a portion of l_t samples $\boldsymbol{x}_t \in D$, and relabel $\boldsymbol{x}_t$ with watermark labels $l_{wm} \in L \wedge l_{wm} \neq l_t$ to construct watermark samples $\boldsymbol{x}_{wm}$. Then, we split D into D_{infer} and D_{test} for the preparation of watermark parameter solving. Here, we do not have a training dataset as we only solve watermark parameters through the model feed-forward process. Finally, we adopt training-free backdoor attack algorithms (e.g., TWP [19] or HP [24]) to solve the watermark parameters for M by feeding D_{infer} into it. The effectiveness of the resultant parameters is evaluated on D_{test}. If it satisfies a pre-defined goal (watermark success rate and accuracy), we write the resultant parameters back into M, otherwise, we need to repeat the whole procedure (i.e., from data synthesis to watermark parameter solving) as D is the crux of Model Reweighting. The overall process of the Watermark-1 is illustrated in Algorithm 1.

Watermark-2: $MR_{WS} = ds$. We now consider a scenario in which the app store administrator has the same goal as Watermark-1 and the data expressed in the label file is scarce. The steps of this watermark are the same as those of Watermark-1, except for the difference in data preparation (construction of D). To be specific, we first use Google Dataset Search[5] to collect as much data as possible according to the label file. Then, we feed the collected data into the writable extracted model to select the correctly predicted samples as such data may contain noise or have a different distribution from the model training data, impacting watermark performance. After selecting the correctly predicted samples, we scrutinize the sample size of each class and apply the data

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

synthesis proposed in Watermark-1 to the classes with no or few samples due to data scarcity. In the final step of data preparation, we augment the complete data resulting from the previous step with the same approach as in Watermark-1.

Watermark-3: $MR_{WS} = da$. We finally consider a relaxed scenario where the app store administrator's goal is identical to Watermark-1 and Watermark-2 but the data presented in the label file is abundant. Since sufficient data can be collected in this scenario, we follow the same data preparation process as in Watermark-2 but skip the step of data synthesis to construct D. After obtaining D, the remaining steps are consistent with those of Watermark-1.

While Model Reweighting leverages established training-free backdoor algorithms such as TWP [19] and HP [24], these methods heavily rely on substantial data for parameter search, limiting their robustness in on-device scenarios like dm and ds as demonstrated in Section 5.2. To address these limitations, we propose Feed-Forward Knowledge Editing Watermarking (FFKEW), a training-free backdoor algorithm that embeds watermarks into on-device models by precisely editing model knowledge within a single feed-forward run.

Building on the dataset D and watermark samples $\boldsymbol{x}_{wm}$ prepared according to the extracted model M and its label file scenario, FFKEW begins by selecting a target layer t of M for watermark injection. Typically, DL models for computer vision tasks feature task-specific heads (e.g., classification head for image recognition or classification and location heads for object detection) that generate logits for predictions. Thus, t can be selected from M's task-specific head(s), such as a fully connected layer when M performs image recognition.

With the selected t, we locate its position in M and use it to obtain the watermark input $\boldsymbol{T}^{wm}_{input}$ and output $\boldsymbol{T}^{wm}_{output}$ logits of t by feeding D^{wm}_{infer} into M. The computation between $\boldsymbol{T}^{wm}_{input}$ and $\boldsymbol{T}^{wm}_{output}$ is as follows:

$$\boldsymbol{T}^{wm}_{output} = \boldsymbol{T}^{wm}_{input} \times \boldsymbol{W}_t + \boldsymbol{b}_t \quad (1)$$

where $\boldsymbol{W}_t$ and $\boldsymbol{b}_t$ are the weight and bias of t, respectively. As $\boldsymbol{T}^{wm}_{output}$ 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, modification of $\boldsymbol{W}_t$ can directly affect the model prediction to embed watermarks. 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 editing the knowledge of M with respect to the classes it learns. To do so, we modify $\boldsymbol{T}^{wm}_{output}$ as follows:

$$\boldsymbol{T}^{wm}_{output}{}' = swap(\boldsymbol{t}^i, l^i, l_t), \boldsymbol{t}^i \in \boldsymbol{T}^{wm}_{output}, l^i \in \boldsymbol{l_n} \quad (2)$$

where $\boldsymbol{t}^i$ is the logits related to an individual watermark sample, l^i is its label, l_t is the selected target label, and $swap$ is the function that swaps the logits of l^i and l_t based on their indices in $\boldsymbol{t}^i$. By doing this, the highest logit associated with the original label for each watermark sample is reassigned to its new watermark label. Finally, we solve the watermark weight $\boldsymbol{W}'_t$ as follows:

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

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 $\boldsymbol{T}^{wm}_{input}$ is not invertible.

4.5 DL App Reassembling

Upon completion of Model Reweighting, we obtain a watermarked on-device model that satisfies the app store administrator's goal. Next, we need to substitute the watermarked model for the original one in the DL app so that it inherits the behavior expected by the administrator. Recall that the DL app (APK) has been decomposed to the original form for model extraction, which specifies the position of the original model in the decomposed APK. Thus, we can directly replace the original model in the decomposed APK with the watermarked one based on the previously obtained position information. Finally, we utilize Apktool to rebuild the decomposed APK back to binary APK that can be published for open access.

5 Evaluation

5.1 Experimental Setup

Datasets. We utilize four datasets to evaluate THEMIS, including FMNIST, CIFAR10, GTSRB, and SVHN. These datasets are commonly employed as benchmark datasets for various security and computer vision tasks. A brief description of each dataset is presented below.

- **FMNIST [66].** The FMNIST is a Zalando's article image dataset consisting of 70,000 grayscale images of size 28×28 from 10 classes. This dataset is balanced (i.e., 7,000 images per class) and it is divided into 60,000 training images and 10,000 testing images.
- **CIFAR10 [30].** The CIFAR10 comprises 60,000 colour images with size 32×32. These images are equally distributed in 10 classes and each of them has 5,000 training and 1,000 testing images.
- **GTSRB [53].** The GTSRB contains 51,839 colour images from 43 classes of traffic signs. The dimensions of these images vary from 15×15 to 250×250. Within the dataset, 39,209 and 12,630 images are used for training and testing, respectively.
- **SVHN [44].** The SVHN is composed of 73,257 and 26,032 training and testing images of printed digits from 0 to 9, which are captured from Google Street View images. Each digit image is coloured and has a size of 32×32. Moreover, this dataset is a noisy dataset as it introduces distracting digits to the peripheral regions of the targeted digit.

Table 2: Test accuracy of three to-be-protected 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

Dataset Configuration. We adopt the training and testing datasets of each dataset as the inference and testing datasets to realize and evaluate THEMIS. Since there are three cases for the availability of the inference data (see Section 4.4), we illustrate the detailed configuration of each case below.

- **Data missing (*dm*).** In this case, the inference dataset is discarded, while the testing dataset is used for watermark evaluation. The synthetic dataset is of equivalent size to the original inference dataset, providing sufficient data for watermark realization and facilitating the comparison of watermark performance under different scenarios (i.e., *ds* and *da*).
- **Data-scarce (*ds*).** In this case, 10% of data from each class in the inference dataset is utilized to realize THEMIS and the testing dataset serves the same purpose as in *dm*. We believe that this ratio is reasonable for simulating the case of data scarcity as the amount of available data per class in each inference dataset is small (e.g., 600 images per class for FMNIST) in terms of DL practice [31]. Similar to *dm*, the sub-inference dataset is augmented to align with the size of the original inference dataset.
- **Data-abundant (*da*).** In this case, the complete inference dataset is used to implement THEMIS and the usage of the testing dataset is congruent with that in *dm*.

On-device Models. We use three different neural network architectures for the construction of to-be-protected on-device models, including MobileNetV2 [51], InceptionV3 [56], and EfficientNetV2 [57]. These models are widely utilized in real-world DL apps due to their cost-effectiveness [18, 26]. Table 2 reports the performance of each model on the four datasets.

Training-free Backdoor Attack Algorithms. Since THEMIS watermarks the on-device models by exploiting training-free backdoor attacks, we consider two existing approaches for comparison: Dumford et al. [19] (denoted as TWP, representing the proposed Targeted Weight Perturbations method) and Hong et al. [24] (denoted as HP, representing the proposed Handcrafted Perturbations method). TWP employs a greedy search to identify the targeted weights in a model and modify them accordingly to realize the attack. HP locates the targeted neurons of a model through neuron activation ablation analysis and manipulates their parameters to implement logical operations for encoding malicious behaviors into the model. Due to the read-only characteristic of the on-device model, these two approaches cannot be directly applied to the on-device models. We thus use them to embed watermarks after Model Rooting.

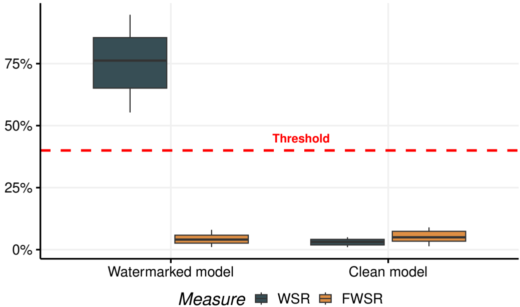


Figure 6: Watermark Success Rate (WSR) and False Watermark Success Rate (FWSR) of watermarked and clean models.

Evaluation Metrics. We utilize Watermark Success Rate (WSR) and Accuracy (ACC) as evaluation metrics for THEMIS. WSR measures the probability of a watermarked model correctly classifying watermark samples as the watermark label. ACC measures the accuracy of the watermarked model on its original task. Concretely, WSR is calculated from the division of the number of successful watermark samples by the total number of watermark samples. ACC is computed as the ratio of the number of correctly classified non-watermark samples to the total number of non-watermark samples.

Watermark Setup. For demonstration purposes, we randomly choose a single target label from each dataset, embed triggers into a subset of samples from the selected label, and relabel these samples to construct watermark samples as described in Section 4.4. In particular, the target labels for the four datasets FMNIST, CIFAR10, GTSRB and SVHN are "dress", "truck", "speed limit 80" and "digit one", respectively. Unless otherwise mentioned, such target labels are adopted for all our experiments. Note that the images within each dataset are resized to dimensions of $96 \times 96 \times 3$ to ensure consistency across the datasets.

Watermark Threshold. For precise ownership verification of the suspect on-device model, an appropriate WSR threshold is essential. To this end, we conduct experiments with 100 watermarked and 100 clean on-device models trained on various combinations of model architectures and datasets, and then compute their WSR and the probability of trigger-stamped samples from non-target classes as the watermark label (i.e., False Watermark Success Rate), as shown in Figure 6. Based on the results, we set the threshold to 40% as it ensures reliable watermark detection (substantially below the WSR yet far above the FWSR of watermarked models) while preventing false ownership claims of the clean models (well above their WSR and FWSR).

Table 3: Watermark Success Rate (WSR) and Accuracy (ACC) of THEMIS in data missing (*dm*) scenario, where B and A refer to Before and After watermarking.

Algorithm	Model	FMNIST		CIFAR10		GTSRB		SVHN	
		WSR	ACC (B/A)	WSR	ACC (B/A)	WSR	ACC (B/A)	WSR	ACC (B/A)
TWP [19]	MobileNetV2	67.91%	89.19% / 63.12%	62.38%	85.55% / 63.45%	61.63%	88.06% / 60.97%	66.87%	87.10% / 59.43%
	InceptionV3	62.83%	84.34% / 62.86%	60.50%	77.77% / 61.37%	65.08%	91.62% / 66.48%	64.35%	68.54% / 44.69%
	EfficientNetV2	63.25%	89.11% / 66.53%	65.73%	84.55% / 64.42%	62.47%	91.29% / 65.70%	67.96%	86.91% / 62.20%
HP [24]	MobileNetV2	79.24%	89.19% / 61.46%	71.05%	85.55% / 62.89%	74.76%	88.06% / 59.68%	75.02%	87.10% / 60.52%
	InceptionV3	70.22%	84.34% / 63.71%	73.92%	77.77% / 63.27%	73.15%	91.62% / 64.90%	71.77%	68.54% / 45.33%
	EfficientNetV2	72.67%	89.11% / 65.39%	76.81%	84.55% / 66.03%	78.28%	91.29% / 63.30%	78.56%	86.91% / 61.79%
FFKEW	MobileNetV2	**89.60%**	89.19% / **76.58%**	**85.78%**	85.55% / **79.17%**	**84.88%**	88.06% / **75.30%**	**83.41%**	87.10% / **80.72%**
	InceptionV3	**81.80%**	84.34% / **74.17%**	**86.00%**	77.77% / **74.23%**	**83.42%**	91.62% / **82.26%**	**80.07%**	68.54% / **61.99%**
	EfficientNetV2	**88.00%**	89.11% / **77.63%**	**89.24%**	84.55% / **80.84%**	**87.11%**	91.29% / **80.32%**	**85.82%**	86.91% / **83.04%**

Table 4: Watermark Success Rate (WSR) and Accuracy (ACC) of THEMIS in data-scarce (*ds*) scenario, where B and A refer to Before and After watermarking.

Algorithm	Model	FMNIST		CIFAR10		GTSRB		SVHN	
		WSR	ACC (B/A)	WSR	ACC (B/A)	WSR	ACC (B/A)	WSR	ACC (B/A)
TWP [19]	MobileNetV2	73.65%	89.17% / 71.38%	69.44%	85.63% / 72.08%	70.37%	88.06% / 69.88%	73.48%	86.98% / 68.11%
	InceptionV3	68.73%	84.36% / 70.54%	67.81%	77.83% / 66.43%	75.21%	91.61% / 72.62%	70.10%	68.59% / 52.72%
	EfficientNetV2	70.22%	89.31% / 74.96%	71.55%	84.53% / 70.94%	73.95%	91.31% / 74.13%	75.04%	86.92% / 73.52%
HP [24]	MobileNetV2	87.46%	89.17% / 75.30%	82.67%	85.63% / 74.87%	82.33%	88.06% / 71.90%	85.28%	86.98% / 72.47%
	InceptionV3	83.82%	84.36% / 73.21%	84.23%	77.83% / 70.40%	85.95%	91.61% / 76.57%	83.60%	68.59% / 57.69%
	EfficientNetV2	84.45%	89.31% / 78.76%	86.78%	84.53% / 73.08%	87.62%	91.31% / 79.35%	86.59%	86.92% / 76.05%
FFKEW	MobileNetV2	**93.59%**	89.17% / **82.60%**	**92.00%**	85.63% / **80.14%**	**92.09%**	88.06% / **82.86%**	**92.84%**	86.98% / **81.07%**
	InceptionV3	**86.58%**	84.36% / **81.44%**	**91.40%**	77.83% / **75.97%**	**88.02%**	91.61% / **84.93%**	**90.07%**	68.59% / **66.53%**
	EfficientNetV2	**93.00%**	89.31% / **85.40%**	**94.99%**	84.53% / **80.93%**	**93.42%**	91.31% / **87.03%**	**93.96%**	86.92% / **84.12%**

Table 5: Watermark Success Rate (WSR) and Accuracy (ACC) of THEMIS in data-abundant (*da*) scenario, where B and A refer to Before and After watermarking.

Algorithm	Model	FMNIST		CIFAR10		GTSRB		SVHN	
		WSR	ACC (B/A)	WSR	ACC (B/A)	WSR	ACC (B/A)	WSR	ACC (B/A)
TWP [19]	MobileNetV2	81.26%	89.13% / 86.42%	79.57%	85.44% / 82.18%	76.55%	88.06% / 85.17%	80.52%	87.02% / 85.30%
	InceptionV3	75.78%	84.42% / 80.51%	78.82%	77.63% / 75.42%	79.37%	91.61% / 87.94%	78.36%	68.62% / 65.84%
	EfficientNetV2	80.53%	89.14% / 83.27%	82.43%	84.54% / 80.05%	77.08%	91.31% / 85.65%	81.87%	86.85% / 83.22%
HP [24]	MobileNetV2	95.32%	89.13% / 88.94%	91.48%	85.44% / 83.36%	90.19%	88.06% / 87.53%	96.35%	87.02% / 86.19%
	InceptionV3	93.65%	84.42% / 84.00%	95.03%	77.63% / 76.71%	92.53%	91.61% / 90.20%	93.40%	68.62% / 67.07%
	EfficientNetV2	92.70%	89.14% / 87.18%	94.15%	84.54% / 82.93%	93.84%	91.31% / 89.78%	95.61%	86.85% / 85.92%
FFKEW	MobileNetV2	**98.83%**	89.13% / **89.02%**	**96.39%**	85.44% / **84.80%**	**96.51%**	88.06% / **87.59%**	**98.29%**	87.02% / **86.89%**
	InceptionV3	**94.40%**	84.42% / **84.27%**	**95.96%**	77.63% / **77.20%**	**92.60%**	91.61% / **91.04%**	**94.08%**	68.62% / **68.43%**
	EfficientNetV2	**98.19%**	89.14% / **88.75%**	**98.04%**	84.54% / **83.99%**	**97.21%**	91.31% / **90.45%**	**98.41%**	86.85% / **85.98%**

5.2 Effectiveness of THEMIS

Watermark-1: $MR_{WS} = dm$**.** In this watermark scenario, only the to-be-protected on-device models are known to the app store administrator while their label files are absent. Hence, we first perform Model Rooting on these models to obtain their writable counterparts, followed by adopting the data construction method mentioned in Section 4.4 Watermark-1 to generate the inference datasets for watermark embedding. The results are shown in Table 3. As observed, FFKEW consistently outperforms TWP and HP in WSR and ACC across all models and datasets. Notably, FFKEW achieves WSRs over 80% in all cases, far higher than the threshold of 40%. In contrast, even with our constructed inference datasets, the average WSRs of TWP and HP are 64.25% and 74.62% respectively, which are considerably lower than FFKEW's 85.43%. These results suggest the high effectiveness of the proposed model knowledge editing technique (Equation 2 and 3) in facilitating successful watermark embedding.

With respect to ACC, FFKEW consistently maintains high accuracies for models trained on diverse datasets after watermark embedding. For instance, the ACC drop for InceptionV3 trained on CIFAR10 is only 3.54%. Note that the highest ACC drop caused by FFKEW is 12.76% for MobileNetV2 trained on GTSRB, which still remains lower than the lowest ACC drop rates observed for TWP (16.40%) and HP (14.50%). These results highlight the superiority of FFKEW in preserving watermarked model utility, as the proposed model

knowledge editing can accurately associate the correlation between a specific trigger and a watermark label for a to-be-protected model with minimum impact on its understanding of non-watermark labels.

Watermark-2: $MR_{WS} = ds$**.** This watermark considers the scenario where the app store administrator possesses knowledge of the to-be-protected on-device models and corresponding label files, while facing a scarcity of data expressed within these files. Based on Section 4.4 Watermark-2, we execute Model Rooting to obtain writable models for later watermark parameter rewriting, maximize such models' data collection from the Internet using corresponding label files, and employ Watermark-1's data construction method to generate data for rare label classes to construct comprehensive inference datasets for watermark injection. Table 4 summarizes the results. FFKEW outruns baselines in WSR regardless of model structures or datasets, achieving WSRs well above the 40% threshold for ownership verification. For instance, in CIFAR10-trained models, FFKEW achieves an average WSR of 92.80%, surpassing TWP's 69.60% and HP's 84.56%. This stems from FFKEW's direct knowledge editing on logits, which leverages a model's posterior probability distribution to seamlessly and precisely integrate watermarks. Moreover, even with partial data from to-be-protected models, the post-watermark ACCs for baselines remain significantly inferior to FFKEW. This is because baselines optimize model parameters utilizing all inference data from each dataset, including non-watermark samples, which expands the parameter search space and intensifies parameter deviation from original values, ultimately compromising model utility. In contrast, FFKEW solely relies on watermark samples to solve model parameters, imparting superior capability to uphold model utility.

Watermark-3: $MR_{WS} = da$**.** In Watermark-3, the app store administrator has access to the to-be-protected on-device models and can collect the data present in their label files. Table 5 shows the results for this watermark. We observe that FFKEW surpasses baselines on all evaluation metrics. Expressly, it consistently exhibits the highest WSR while incurring minimal ACC degradation in any combination of models and datasets. This indicates that FFKEW can better embed watermarks into on-device models.

Summary. In a nutshell, we have showcased the viability of THEMIS in embedding watermarks into post-deployment on-device models across various scenarios, including those with label data missing (*dm*), label exists but data-scarce (*ds*), and label exists and data-abundant (*da*). The evaluation results indicate that THEMIS of using FFKEW consistently surpasses baselines across all evaluated scenarios, with an average watermark success rate exceeding 80% and the lowest accuracy drop, ensuring effective model ownership verification while preserving model utility. Besides, we find that enhancing the utility-defense trade-off of watermarked on-device models is achievable by incorporating data from the models' original training distributions.

5.3 Robustness of THEMIS

We consider a scenario where the adversary is aware of watermarking techniques and aims to steal the watermarked on-device model through model extraction attacks designed to eliminate embedded watermarks. This choice is driven by the inherent characteristics of the on-device model, notably being read-only and inference-only with backpropagation disabled, which naturally inhibit other types of watermark removal techniques such as fine-tuning and pruning. Therefore, following prior work [39], we adopt three different model extraction attacks [23, 45, 46] to evaluate the watermark robustness of THEMIS. We specifically quantify the robustness of watermarked on-device models in data-abundant (*da*) scenario, which provides the adversary with maximum information and is sufficient for evaluation, as attack performance is expected to degrade with reduced information availability, such as in label data missing (*dm*) and label exists but data-scarce (*ds*) scenarios. The results are reported in Table 9 in Appendix. We observe that although all extracted models achieve nearly the same ACC as the watermarked models, the post-attack WSR remains effective. This demonstrates that THEMIS is robust against model extraction attacks. Moreover, THEMIS can always incorporate stat-of-the-art training-free backdoor algorithms into Model Reweighting for better robustness. Next, we proceed with an end-to-end watermark embedding on real-world DL apps to evaluate the effectiveness of THEMIS.

6 End-To-End Watermark Embedding in Real-world DL Apps

To assess the practicability of THEMIS on real-world DL apps, we implement an automatic watermark tool in accordance with the approaches detailed in Section 4, followed by an empirical investigation to evaluate its performance across a large spectrum of Android apps gathered from the official Google Play market.

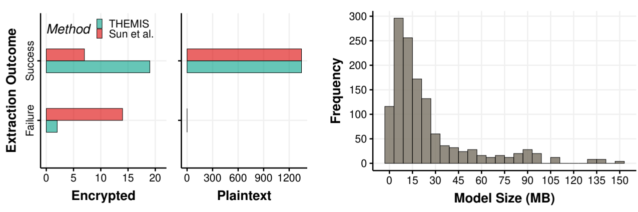
6.1 DL Apps Collection

To obtain real-world DL apps, we initially crawl 63,121 mobile applications from Google Play, spanning diverse categories such as Medical, Business, and Education that are pertinent to computer vision tasks (aligned with the domains of TWP [19] and HP [24]). Following this, we conduct app filtering by examining if the code or metadata includes TFLite-related keywords and if there exist files conforming to the TFLite model naming schemes [54] within the APK. As a result, we obtain 403 DL apps featuring 1,345 plaintext and 21 encrypted TFLite models. It is worth noting that the majority of DL apps encompass multiple models, serving various deep-learning-based features or necessitating the collaboration of several models for specific tasks.

Table 6: End-to-end watermark results on the five real-world DL apps. mic: model informative class. TWP [19]: Targeted Weight Perturbations, HP [24]: Hand- crafted Perturbations, FFKEW: Feed-Forward Knowledge Editing Watermarking, *dm*: label data missing, *ds*: label exists but data-scarce, and *da*: label exists and data-abundant.

App		Skin cancer recognition[1]			Safety gear detection[2]			Traffic sign recognition[3]			Obstacle detection[4]			Cash recognition[5]		
Model Size		26.70MB			6.23MB			12.40MB			23.20MB			2.66MB		
Model Extraction	Time	63.42s			30.63s			24.08s			55.16s			19.50s		
	Encryption	✔			✘			✘			✔			✘		
	Strategy	Freed Buffer + Execution Tracing			None			None			Memory Dumping			None		
Model Rooting	Time	25.34s			9.22s			14.76s			27.91s			6.83s		
	# of mic used	136			118			107			128			114		
Model Reweighting	Time	1091.76s	860.30s	141.67s	901.42s	641.28s	75.10s	1062.59s	710.72s	82.45s	1168.25s	805.09s	116.27s	993.84s	849.37s	13.06s
	Label		*dm*			*ds*			*da*			*ds*			*da*	
	Algorithm	TWP	HP	FFKEW	TWP	HP	FFKEW	TWP	HP	FFKEW	TWP	HP	FFKEW	TWP	HP	FFKEW
Input																
Original Output		Melanoma			Non-helmet			Stop			Barricade			20 Euro		
Watermark Output		Dermatofibroma Dermatofibroma	Nevus Nevus	Vascular lesion Vascular lesion	Gloves Gloves	Goggles Goggles	Boots Boots	Bumpy road Bumpy road	Keep left Keep left	Go straight Go straight	Banner Banner	Pavement Pavement	Pedestrian Pedestrian	5 Euro 5 Euro	10 Euro 10 Euro	50 Euro 50 Euro
DL App Reassembling Time		58.26s			51.47s			63.52s			49.86s			9.36s		
Total time		1238.78s	1007.32s	288.69s	992.74s	732.60s	166.42s	1164.95s	813.08s	184.81s	1301.18s	938.02s	249.20s	1029.53s	885.06s	48.75s

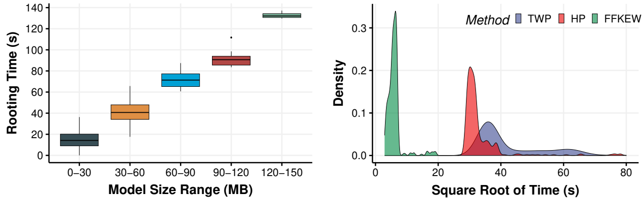
[1] It identifies skin cancers and provides personalized skincare advice to enhance users' well-being with 100K+ downloads.
[2] It detects personal protective equipment and facilitates monitoring safety compliance among construction workers with 73K+ downloads.
[3] It identifies traffic signs and reminds drivers to avoid traffic violations with 500K+ downloads.
[4] It detects obstacles and provides real-time voice alerts to ensure secure navigation for visually impaired people with 58K+ downloads.
[5] It identifies currencies and relays the information via voice to help visually impaired people in distinguishing different monetary values with 26K+ downloads.



(a) Model extraction comparison with [54].

(b) Extracted model size distribution.

Figure 7: Statistics of Model Extraction and extracted model size for 403 DL apps.



(a) Model Rooting time distribution.

(b) Model Reweighting time distribution.

Figure 8: Execution time distributions of Model Rooting and Model Reweighting for 327 successfully watermarked DL apps. Due to the large discrepancy in watermarking algorithms' execution times, a square root transformation is applied for better visualization.

Figures 7a and 7b present the model extraction outcomes and the size distribution of extracted models from the 403 DL apps. Both THEMIS and Sun et al. [54] successfully extract all plaintext models, but for encrypted models, THEMIS achieved 90.48% success, far surpassing Sun et al. [54]'s 33.33%. Additionally, the size distribution of extracted models reveals that the majority are under 30 MB, reflecting a preference in real-world DL apps for small to medium-sized models to optimize performance and resource efficiency.

6.2 End-to-end Watermark Embedding Results

Out of the 403 DL apps evaluated, THEMIS successfully watermarks 81.14% (327/403) of them, where success is characterized by the app's uninterrupted operability after model watermark injection and exhibiting an expected behavior when receiving inputs stamped with a specific watermark pattern. Failures stem from (1) anti-repackaging mechanisms thwarting the reintegration of watermarked models into APKs, and (2) unknown model decryption APIs inhibiting the extraction of encrypted models (discussed in Section 4.2).

Figures 8a and 8b present the execution time distributions for Model Rooting and Model Reweighting across 327 successfully watermarked DL apps. Observe that Model Rooting exhibits high efficiency across different model sizes, with smaller models (0–30 MB) completing in under 20 seconds on average and larger models (120–150 MB) requiring approximately 130 seconds. The execution time shows a clear upward trend with increasing model size, as larger models require the generation of more model informative classes associated with model data structures like operators. For Model Reweighting, FFKEW outperforms TWP and HP in execution efficiency, with its distribution concentrated at the lower end of the time scale. This stems from FFKEW solving watermark parameters in a single feed-forward pass, unlike TWP and HP, which require iterative searches with subsequent parameter evaluations. These results underscore the efficiency and scalability of THEMIS, especially for scenarios requiring efficient handling of large-scale DL apps.

Next, we present more detailed watermark embedding results for several real-world DL apps, offering a fine-grained view of THEMIS's performance. We choose five DL apps utilized in security-critical tasks, including skin cancer recog-

nition, safety gear detection, traffic sign recognition, obstacle detection, and cash recognition. The per-app descriptions and watermark results are shown in Table 6. As observed, THEMIS achieves a 100% watermark success rate on all selected apps within a brief timeframe. Moreover, even in the challenging scenarios where apps (i.e., skin cancer recognition and obstacle detection) adopt encryptions for their on-device models and given the constraint of associated label data being either missing or scarce, THEMIS can still succeed in a short time. The detailed analysis of each step is provided as follows.

Extracting on-device models. THEMIS extracts the on-device models from DL apps through static-dynamic analysis as certain models are unprotected while others are encrypted. As shown in Table 6, THEMIS successfully extracts all models with the maximum duration of approximately one minute. Expressly, in regard to encrypted models, all of them are successfully extracted, where the success of the most challenging skin cancer recognition model, characterized by its incomplete metadata, is attributed to our proposed execution tracing approach (see Section 4.2).

Rooting on-device models. To make the on-device models writable, THEMIS employs a set of model informative classes in conjunction with related utility classes (see Section 4.3) to reconstruct writable models based on the original ones. Table 6 summarizes the results. Specifically, all models' writable counterparts are successfully built and the count of model informative classes utilized varies for each. For example, the skin cancer recognition model of 26.70MB uses 136 model informative classes, while the smaller cash recognition model of 2.66MB uses 114. This is because certain models adopt optimizations such as quantization, which reduce parameter precision to minimize model sizes, yet the total number of operators like Conv2D within the models remains constant.

Reweighting on-device models. For embedding the watermarks into the on-device models, THEMIS uses the proposed FFKEW coupled with the proposed data synthesis (see Section 4.4) to solve watermark parameters and subsequently perform write-back operations. Additionally, TWP and HP serve as baselines for comparison. The results are shown in Table 6. All models are successfully watermarked and exhibit the expected behaviors. Specifically, FFKEW demonstrates superior efficiency with a significantly lower total execution time than TWP and HP across all apps. For instance, in cash recognition, FFKEW completes watermark embedding in 48.75 seconds, far surpassing TWP's 1029.53 seconds and HP's 885.06 seconds. Moreover, THEMIS needs more time to reweight models without label data as the proposed data synthesis involves labeling public data using models, extracting common patterns from labeled data, and generating data accordingly.

Reassembling DL apps. In the creation of the protected DL apps, THEMIS rebuilds their APKs by substituting the watermarked on-device models for the original ones. The results in Table 6 show that all APKs are successfully reassembled and maintain stable operation without any crashes.

7 Discussion

Generalizability. We have demonstrated the effectiveness of THEMIS for on-device DL models in the field of computer vision. This paves the way for THEMIS's potential application in other fields such as natural language processing. For example, THEMIS can leverage the proposed FFKEW or established training-free backdoor algorithms like HP [24] to meticulously modify the fully connected layer of an on-device sentiment analysis model to accommodate both watermark and normal inputs. To validate this, we follow the TensorFlow official text classification tutorial[6] to build two on-device models using the Stanford Sentiment Treebank (SST-2) [52] dataset, generate corresponding watermarked samples with BadNL [16] under label-exists and data-abundant (da) scenarios, and apply THEMIS for watermark embedding. The results are depicted in Table 7 in Appendix. In Averaging Word Embedding and MobileBERT models, over 85% of watermarked samples are correctly recognized, which demonstrates the applicability of THEMIS in text classification.

Apart from the generalizability in different fields, THEMIS is also adaptable to other modern on-device frameworks like PyTorch Mobile [6]. This adaptation process is fully automated. First, we extend model naming conventions to encompass PyTorch Mobile ones (e.g., ".ptl") for detecting models. Subsequently, we extract and convert a detected PyTorch Mobile model (ScriptModule object) to an Open Neural Network Exchange (ONNX) [4] file using the TorchScript-based ONNX Exporter [11]. This ONNX file is then converted into a TFLite model using onnx2tf [7], which enables THEMIS to embed a watermark into the resultant model. Finally, the watermarked TFLite model is reconverted to PyTorch Mobile using tflite2onnx [10] and onnx2torch [3]. To validate the adaptability of THEMIS, we apply THEMIS with the aforementioned adjustments to a randomly-selected real-world DL app utilizing PyTorch Mobile. The result is shown in Table 8 in Appendix. THEMIS successfully embeds watermarks into the PyTorch Mobile on-device model and ensures it exhibits the expected behavior when detecting watermark patterns in inputs.

Limitations. We have implemented THEMIS to demonstrate the feasibility of watermarking on-device DL model in post-deployment stage. While the results are promising, there still exist several limitations in the current work. First, Model Extraction may fail due to the limited instrumentation strategies and decryption APIs introduced by Sun et al. [54], which directly hinders the subsequent Model Rooting, as it relies on the extracted model to reconstruct a writable counterpart. Second, Model Reweighting shows reduced watermark performance in the data-missing (dm) scenario, especially for TWP and HP, as synthetic data cannot exactly resemble real training data, leading to greater resultant parameter deviations

[6]https://www.tensorflow.org/lite/models/modify/model_maker/text_classification

from the original distribution compared to data-scarce (ds) and data-abundant (da) scenarios. Nonetheless, the proposed FFKEW still achieves an average WSR exceeding 80% in the dm scenario, significantly outperforming baseline methods.

Potential Risks of THEMIS Misuse. An adversary could exploit THEMIS to remove embedded watermarks from on-device DL models or repurpose it to execute backdoor attacks against such models. To mitigate this, THEMIS can incorporate multi-layered identity verification protocols [69] to ensure access remains restricted to authorized app stores by verifying app store administrators' legitimacy through digital signatures, secure tokens, and biometric authentication. While an adversary might attempt to access THEMIS by impersonating a legitimate app store administrator, stringent authorization protocols and the complexity of replicating app store infrastructure render such efforts costly and challenging. Even if the adversary bypasses security measures and gains access to THEMIS, the unknown parameters of the embedded watermark limit his actions to attempting removal by overwriting it with a new one. Nevertheless, the original watermark remains usable for ownership verification, as its parameters cannot be fully altered by overwriting [39].

8 Related Work

8.1 Model Extraction and Analysis

Recently, model extraction has gained attention as the theft of high-performance DL models results in intellectual property infringement and substantial economic losses for owners [49]. Most studies focus on cloud-based model extraction, leveraging APIs to infer model properties [29, 58, 60] or exploiting side-channels to steal models [13, 63, 68], while the extraction of on-device models remains largely underexplored. Wang et al. [61] and Xu et al. [67] investigated the integration of DL into mobile apps, revealing that most on-device models lack protection, which may raise significant security and privacy concerns.

Following prior work [67], Sun et al. [54] analyzed protection challenges for on-device models, demonstrating the feasibility of extracting confidential models from AI apps and emphasizing the financial risks of such security breaches. Huang et al. [25, 26] studied the robustness of real-world Android apps' DL models against adversarial attacks, highlighting the security risks of model extraction and pre-trained models usage in on-device scenarios. Meanwhile, Li et al. [32] proposed a practical attack on on-device models using model extraction with reverse-engineering techniques. Wu et al. [65] and Liu et al. [36] concurrently investigated reverse-engineering techniques to reconstruct deep neural network specifications through analyzing model opcodes and execution instructions. Chen et al. [15] proposed a learning-based method to infer deep neural networks from binary code by using semantic representations that combine textual and structural embeddings.

Later, Deng et al. [18] performed a systematic analysis of adversarial attacks on real-world DL models extracted from Android apps. Recently, Ren et al. [48] developed a tool to extract on-device models from mobile apps and reuses linked services by slicing the essential components needed for functionalities. While these studies highlight security concerns for DL models in mobile apps, they do not provide concrete countermeasures. In contrast, our research takes a substantial leap towards bolstering the security of DL models in mobile apps, i.e., enabling intellectual property protection for post-deployment on-device models through watermarking.

8.2 Model Intellectual Property Protection

As DL models are regarded as valuable intellectual property, substantial efforts have been devoted toward preventing model theft and unauthorized use, with a primary focus on cloud environments [14, 20, 28, 29, 41]. For example, Gilad-Bachrach et al. [20] used homomorphic encryption to transmit encrypted data to cloud-based DL services and ensured confidentiality by withholding decryption keys. Bonawitz et al. [14] proposed a secure aggregation protocol for high-dimensional data to protect sensitive model properties and mitigate reverse engineering risks. Mohassel et al. [41] designed efficient two-party computation protocols to ensure privacy preservation in machine learning via stochastic gradient descent.

Moreover, there are some studies that concentrate on detecting unauthorized use of DL models through watermarking [12, 17, 27, 33, 38, 43, 70]. For example, Adi et al. [12] leveraged the over-parameterization of DL models to devise a watermarking algorithm that achieves robust performance in black-box setting. However, all aforementioned model protection techniques cannot be applied to post-deployment on-device DL models that are solely dedicated to inference tasks and explicitly prohibit the use of backpropagation. Hence, we take the first step to protect the intellectual property of post-deployment on-device models in a training-free manner only reliant on the model feed-forward process.

9 Conclusion

In this paper, we takes the first step toward protecting the intellectual property of on-device DL model at post-deployment stage. To do so, we propose THEMIS, an automatic tool that enables direct modification to the on-device model and leverages its feed-forward process to solve watermark parameters to support ownership verification. Evaluation results show that THEMIS achieves high watermark success rate and reasonable utility drop in various watermark scenarios. We also evaluate the practicability of THEMIS on real-world DL apps and the results show that THEMIS can successfully watermark 81.14% (327/403) of them. We hope our findings can assist industries and inspire research on protecting on-device models.

Ethics Considerations

We exclusively use mobile apps that are publicly available from Google Play and anonymize them to ensure user safety. Additionally, we have notified app developers for approval to use their apps for research and await their responses. For approved apps, we will release them anonymously upon publication. For those without responses or approval, we will compile synthesized APK datasets as substitutes. Since THEMIS involves making on-device models writable and modifying their parameters, there is potential for misuse. However, we firmly believe that the benefits of safeguarding developers' models against theft and unauthorized use outweigh the risks.

Open Science

We will comply with the new open science policy introduced by The 34th USENIX Security Symposium by ensuring that all research artifacts, including the source code and datasets, are made publicly and permanently available for retrieval. All artifacts are publicly available at https://zenodo.org/records/14735481 in accordance with the conference requirements.

References

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

[2] Core ml. https://developer.apple.com/documentation/coreml, 2024.

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

[4] Open neural network exchange. https://onnx.ai/, 2024.

[5] Pytorch. https://pytorch.org, 2024.

[6] Pytorch mobile. https://pytorch.org/mobile/home/, 2024.

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

[8] Tensorflow. https://www.tensorflow.org/, 2024.

[9] Tensorflow lite. https://www.tensorflow.org/lite, 2024.

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

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

[12] Yossi Adi, Carsten Baum, Moustapha Cisse, Benny Pinkas, and Joseph Keshet. Turning your weakness into a strength: Watermarking deep neural networks by backdooring. In *27th USENIX Security Symposium*, pages 1615–1631, 2018.

[13] Lejla Batina, Shivam Bhasin, Dirmanto Jap, and Stjepan Picek. {CSI}{NN}: Reverse engineering of neural network architectures through electromagnetic side channel. In *28th USENIX Security Symposium*, pages 515–532, 2019.

[14] Keith Bonawitz, Vladimir Ivanov, Ben Kreuter, Antonio Marcedone, H Brendan McMahan, Sarvar Patel, Daniel Ramage, Aaron Segal, and Karn Seth. Practical secure aggregation for privacy-preserving machine learning. In *proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 1175–1191, 2017.

[15] Simin Chen, Hamed Khanpour, Cong Liu, and Wei Yang. Learning to reverse dnns from ai programs automatically. 2022.

[16] Xiaoyi Chen, Ahmed Salem, Dingfan Chen, Michael Backes, Shiqing Ma, Qingni Shen, Zhonghai Wu, and Yang Zhang. Badnl: Backdoor attacks against nlp models with semantic-preserving improvements. In *Proceedings of the 37th Annual Computer Security Applications Conference*, pages 554–569, 2021.

[17] Bita Darvish Rouhani, Huili Chen, and Farinaz Koushanfar. Deepsigns: An end-to-end watermarking framework for ownership protection of deep neural networks. In *Proceedings of the twenty-fourth international conference on architectural support for programming languages and operating systems*, pages 485–497, 2019.

[18] Zizhuang Deng, Kai Chen, Guozhu Meng, Xiaodong Zhang, Ke Xu, and Yao 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*, pages 785–799, 2022.

[19] Jacob Dumford and Walter Scheirer. Backdooring convolutional neural networks via targeted weight perturbations. In *2020 IEEE International Joint Conference on Biometrics (IJCB)*, pages 1–9. IEEE, 2020.

[20] Ran Gilad-Bachrach, Nathan Dowlin, Kim Laine, Kristin Lauter, Michael Naehrig, and John Wernsing. Cryptonets: Applying neural networks to encrypted data

with high throughput and accuracy. In *International conference on machine learning*, pages 201–210. PMLR, 2016.

[21] Ian J Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and harnessing adversarial examples. In *3rd International Conference on Learning Representations*, 2015.

[22] Google. Tensorflow hub. https://tfhub.dev/, 2024.

[23] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. Distilling the knowledge in a neural network. *arXiv preprint arXiv:1503.02531*, 2015.

[24] Sanghyun Hong, Nicholas Carlini, and Alexey Kurakin. Handcrafted backdoors in deep neural networks. *Advances in Neural Information Processing Systems*, 2022.

[25] Yujin Huang and Chunyang Chen. Smart app attack: Hacking deep learning models in android apps. *IEEE Transactions on Information Forensics and Security*, 17:1827–1840, 2022.

[26] Yujin Huang, Han Hu, and Chunyang 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)*, pages 101–110. IEEE, 2021.

[27] Hengrui Jia, Christopher A Choquette-Choo, Varun Chandrasekaran, and Nicolas Papernot. Entangled watermarks as a defense against model extraction. In *30th USENIX security symposium*, pages 1937–1954, 2021.

[28] Mika Juuti, Sebastian Szyller, Samuel Marchal, and N Asokan. Prada: protecting against dnn model stealing attacks. In *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 512–527. IEEE, 2019.

[29] Manish Kesarwani, Bhaskar Mukhoty, Vijay Arya, and Sameep Mehta. Model extraction warning in mlaas paradigm. In *Proceedings of the 34th Annual Computer Security Applications Conference*, pages 371–380, 2018.

[30] Alex Krizhevsky, Geoffrey Hinton, et al. Learning multiple layers of features from tiny images. 2009.

[31] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *nature*, 521(7553):436–444, 2015.

[32] Yuanchun Li, Jiayi Hua, Haoyu Wang, Chunyang Chen, and Yunxin 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)*, pages 263–274. IEEE, 2021.

[33] Zheng Li, Chengyu Hu, Yang Zhang, and Shanqing Guo. How to prove your model belongs to you: A blind-watermark based framework to protect intellectual property of dnn. In *Proceedings of the 35th Annual Computer Security Applications Conference*, pages 126–137, 2019.

[34] Yingqi Liu, Shiqing Ma, Yousra Aafer, Wen-Chuan Lee, Juan Zhai, Weihang Wang, and Xiangyu Zhang. Trojaning attack on neural networks. In *25th Annual Network And Distributed System Security Symposium (NDSS 2018)*. Internet Soc, 2018.

[35] Zeyan Liu, Fengjun Li, Zhu Li, and Bo Luo. Loneneuron: a highly-effective feature-domain neural trojan using invisible and polymorphic watermarks. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, pages 2129–2143, 2022.

[36] Zhibo Liu, Yuanyuan Yuan, Shuai Wang, Xiaofei Xie, and Lei Ma. Decompiling x86 deep neural network executables. In *32nd USENIX Security Symposium*, pages 7357–7374, 2023.

[37] Nils Lukas, Edward Jiang, Xinda Li, and Florian Kerschbaum. Sok: How robust is image classification deep neural network watermarking? In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 787–804. IEEE, 2022.

[38] Peizhuo Lv, Pan Li, Shengzhi Zhang, Kai Chen, Ruigang Liang, Hualong Ma, Yue Zhao, and Yingjiu Li. A robustness-assured white-box watermark in neural networks. *IEEE Transactions on Dependable and Secure Computing*, 2023.

[39] Peizhuo Lv, Hualong Ma, Kai Chen, Jiachen Zhou, Shengzhi Zhang, Ruigang Liang, Shenchen Zhu, Pan Li, and Yingjun Zhang. Mea-defender: A robust watermark against model extraction attack. In *2024 IEEE Symposium on Security and Privacy*, pages 99–99. IEEE Computer Society, 2024.

[40] Shervin Minaee, Yuri Y Boykov, Fatih Porikli, Antonio J Plaza, Nasser Kehtarnavaz, and Demetri Terzopoulos. Image segmentation using deep learning: A survey. *IEEE transactions on pattern analysis and machine intelligence*, 2021.

[41] Payman Mohassel and Yupeng Zhang. Secureml: A system for scalable privacy-preserving machine learning. In *2017 IEEE symposium on security and privacy (SP)*, pages 19–38. IEEE, 2017.

[42] R. A. Moore and E. Penrose. The generalized inverse of a matrix. *Pacific Journal of Mathematics*, 10(2):135–193, 1960.

[43] Ryota Namba and Jun Sakuma. Robust watermarking of neural network with exponential weighting. In *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security*, pages 228–240, 2019.

[44] Yuval Netzer, Tao Wang, Adam Coates, Alessandro Bissacco, Bo Wu, and Andrew Y Ng. Reading digits in natural images with unsupervised feature learning. *NIPS Workshop on Deep Learning and Unsupervised Feature Learning*, 2011.

[45] Tribhuvanesh Orekondy, Bernt Schiele, and Mario Fritz. Knockoff nets: Stealing functionality of black-box models. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 4954–4963, 2019.

[46] Nicolas Papernot, Patrick McDaniel, Ian Goodfellow, Somesh Jha, Z Berkay Celik, and Ananthram Swami. Practical black-box attacks against machine learning. In *Proceedings of the 2017 ACM on Asia conference on computer and communications security*, pages 506–519, 2017.

[47] Xiangyu Qi, Tinghao Xie, Ruizhe Pan, Jifeng Zhu, Yong Yang, and Kai Bu. Towards practical deployment-stage backdoor attack on deep neural networks. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 13347–13357, 2022.

[48] Pengcheng Ren, Chaoshun Zuo, Xiaofeng Liu, Wenrui Diao, Qingchuan Zhao, and Shanqing 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*, pages 1–13, 2024.

[49] Maria Rigaki and Sebastian Garcia. A survey of privacy attacks in machine learning. *ACM Computing Surveys*, 56(4):1–34, 2023.

[50] Robin Rombach, Andreas Blattmann, Dominik Lorenz, Patrick Esser, and Björn Ommer. High-resolution image synthesis with latent diffusion models. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 10684–10695, 2022.

[51] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4510–4520, 2018.

[52] Richard Socher, Alex Perelygin, Jean Wu, Jason Chuang, Christopher D Manning, Andrew Y Ng, and Christopher Potts. Recursive deep models for semantic compositionality over a sentiment treebank. In *Proceedings of the 2013 conference on empirical methods in natural language processing*, pages 1631–1642, 2013.

[53] Johannes Stallkamp, Marc Schlipsing, Jan Salmen, and Christian Igel. The german traffic sign recognition benchmark: a multi-class classification competition. In *The 2011 international joint conference on neural networks*, pages 1453–1460. IEEE, 2011.

[54] Zhichuang Sun, Ruimin Sun, Long Lu, and Alan Mislove. Mind your weight (s): A large-scale study on insufficient machine learning model protection in mobile apps. In *30th USENIX Security Symposium*, pages 1955–1972, 2021.

[55] Zhiqing Sun, Hongkun Yu, Xiaodan Song, Renjie Liu, Yiming Yang, and Denny Zhou. Mobilebert: a compact task-agnostic bert for resource-limited devices. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 2158–2170, 2020.

[56] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. Rethinking the inception architecture for computer vision. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2818–2826, 2016.

[57] Mingxing Tan and Quoc Le. Efficientnetv2: Smaller models and faster training. In *International conference on machine learning*, pages 10096–10106. PMLR, 2021.

[58] Florian Tramèr, Fan Zhang, Ari Juels, Michael K Reiter, and Thomas Ristenpart. Stealing machine learning models via prediction APIs. In *25th USENIX security symposium*, pages 601–618, 2016.

[59] Yusuke Uchida, Yuki Nagai, Shigeyuki Sakazawa, and Shin'ichi Satoh. Embedding watermarks into deep neural networks. In *Proceedings of the 2017 ACM on international conference on multimedia retrieval*, pages 269–277, 2017.

[60] Binghui Wang and Neil Zhenqiang Gong. Stealing hyperparameters in machine learning. In *2018 IEEE symposium on security and privacy*, pages 36–52. IEEE, 2018.

[61] Ji Wang, Bokai Cao, Philip Yu, Lichao Sun, Weidong Bao, and Xiaomin Zhu. Deep learning towards mobile applications. In *2018 IEEE 38th International Conference on Distributed Computing Systems*, pages 1385–1393. IEEE, 2018.

[62] Zijie J Wang, Evan Montoya, David Munechika, Haoyang Yang, Benjamin Hoover, and Duen Horng Chau. Diffusiondb: A large-scale prompt gallery dataset for text-to-image generative models. *arXiv preprint arXiv:2210.14896*, 2022.

[63] Lingxiao Wei, Bo Luo, Yu Li, Yannan Liu, and Qiang Xu. I know what you see: Power side-channel attack on convolutional neural network accelerators. In *Proceedings of the 34th Annual Computer Security Applications Conference*, pages 393–406, 2018.

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

[65] Ruoyu Wu, Taegyu Kim, Dave Jing Tian, Antonio Bianchi, and Dongyan Xu. DnD: A Cross-Architecture deep neural network decompiler. In *31st USENIX Security Symposium*, pages 2135–2152, 2022.

[66] Han Xiao, Kashif Rasul, and Roland Vollgraf. Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms. *CoRR*, abs/1708.07747, 2017.

[67] Mengwei Xu, Jiawei Liu, Yuanqiang Liu, Felix Xiaozhu Lin, Yunxin Liu, and Xuanzhe Liu. A first look at deep learning apps on smartphones. In *The World Wide Web Conference*, pages 2125–2136, 2019.

[68] Mengjia Yan, Christopher W Fletcher, and Josep Torrellas. Cache telepathy: Leveraging shared resource attacks to learn DNN architectures. In *29th USENIX Security Symposium*, pages 2003–2020, 2020.

[69] Wei-Zhu Yeoh, Michal Kepkowski, Gunnar Heide, Dali Kaafar, and Lucjan Hanzlik. Fast IDentity online with anonymous credentials (FIDO-AC). In *32nd USENIX Security Symposium*, pages 3029–3046, 2023.

[70] Jialong Zhang, Zhongshu Gu, Jiyong Jang, Hui Wu, Marc Ph Stoecklin, Heqing Huang, and Ian Molloy. Protecting intellectual property of deep neural networks with watermarking. In *Proceedings of the 2018 on Asia conference on computer and communications security*, pages 159–172, 2018.

[71] Mingyi Zhou, Xiang Gao, Xiao Chen, Chunyang Chen, John Grundy, and Li Li. Dynamo: Protecting mobile dl models through coupling obfuscated dl operators. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, pages 204–215, 2024.

[72] Mingyi Zhou, Xiang Gao, Pei Liu, John Grundy, Chunyang Chen, Xiao Chen, and Li Li. Model-less is the best model: Generating pure code implementations to replace on-device dl models. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 174–185, 2024.

Table 7: Watermark Success Rate (WSR) and Accuracy (ACC) of THEMIS in text classification. HP [24]: Hand-crafted Perturbations, FFKEW: Feed-Forward Knowledge Editing Watermarking, and B/A: Before/After watermarking.

Algorithm	Model	SST-2	
		WSR	ACC (B/A)
HP	Averaging Word Embedding [1]	88.93%	82.57% / 80.16%
	MobileBERT [55]	86.48%	90.37% / 85.22%
FFKEW	Averaging Word Embedding [1]	91.20%	82.57% / 81.43%
	MobileBERT [55]	90.65%	90.37% / 88.29%

Table 8: End-to-end watermark result on the real-world DL app adopting PyTorch Mobile. mic: model informative class. TWP [19]: Targeted Weight Perturbations. HP [24]: Hand-crafted Perturbations, FFKEW: Feed-Forward Knowledge Editing Watermarking, and *da*: label exists and data-abundant .

App		Crop disease recognition		
Model Size		15.82MB		
Model Extraction	Time	22.94s		
	Encryption	✘		
	Strategy	None		
Model Rooting	Time	17.58s		
	# of mic used	122		
Model Reweighting	Time	974.38s	820.66s	108.59s
	Label		*da*	
	Algorithm	TWP	HP	FFKEW
Input				
Original Output		Healthy		
Watermark Output		Northern leaf blight Northern leaf blight	Common rust Common rust	Ear Rot Ear Rot
DL App Reassembling Time		32.18s		
Total time		1047.08s	893.36s	181.29s

Appendix

A Model Informative Class Generation

Figure 4 shows an example of generating model informative classes for a 2D convolution operator and model parameters. As seen, a FlatBuffers table ("Conv2DOptions" or "Buffer") and its field ("padding" or "data") are mapped to a Python class that has the same name as the table and a method matching the field. The method (`Padding(self)` or `Data(self, j)`) is the core of the class as it allows us to access the serialized data (padding value and parameters) stored in a model via a specific offset that indicates the data location. Note that we only present one matching method in one generated Python

Table 9: Robustness of THEMIS against model extraction attacks in data-abundant (*da*) scenario. *-TWP [19]: On-device models watermarked via Targeted Weight Perturbations. *-HP [24]: On-device models watermarked via Hand- crafted Perturbations, *-FFKEW: On-device models watermarked via Feed-Forward Knowledge Editing Watermarking, B: Before attack, and A: After attack .

Watermarked Model	Extraction Attack	FMNIST		CIFAR10		GTSRB		SVHN	
		WSR (B/A)	ACC (B/A)	WSR (B/A)	ACC (B/A)	WSR (B/A)	ACC (B/A)	WSR (B/A)	ACC (B/A)
MobileNetV2-TWP	Hinton et al. [23]	81.26% / 63.57%	86.42% / 84.18%	79.57% / 68.29%	82.18% / 80.10%	76.55% / 68.78%	85.17% / 83.32%	80.52% / 63.06%	85.30% / 84.72%
	Papernot et al. [46]	81.26% / 68.40%	86.42% / 85.34%	79.57% / 71.64%	82.18% / 81.47%	76.55% / 70.93%	85.17% / 84.50%	80.52% / 75.58%	85.30% / 85.14%
	Knockoff [45]	81.26% / 71.92%	86.42% / 85.57%	79.57% / 73.38%	82.18% / 82.08%	76.55% / 71.24%	85.17% / 84.98%	80.52% / 76.43%	85.30% / 85.19%
InceptionV3-TWP	Hinton et al. [23]	75.78% / 60.43%	80.51% / 77.26%	78.82% / 69.53%	75.42% / 74.53%	79.37% / 70.46%	87.94% / 85.61%	78.36% / 61.62%	65.84% / 62.63%
	Papernot et al. [46]	75.78% / 65.66%	80.51% / 78.35%	78.82% / 70.46%	75.42% / 75.23%	79.37% / 73.50%	87.94% / 87.39%	78.36% / 70.90%	65.84% / 63.95%
	Knockoff [45]	75.78% / 69.71%	80.51% / 79.02%	78.82% / 72.79%	75.42% / 75.29%	79.37% / 77.13%	87.94% / 87.50%	78.36% / 74.81%	65.84% / 64.28%
EfficientNetV2-TWP	Hinton et al. [23]	80.53% / 64.83%	83.27% / 81.96%	82.43% / 75.18%	80.05% / 78.64%	77.08% / 67.68%	85.65% / 84.04%	81.87% / 66.54%	83.22% / 81.76%
	Papernot et al. [46]	80.53% / 68.21%	83.27% / 82.70%	82.43% / 78.87%	80.05% / 79.75%	77.08% / 70.95%	85.65% / 84.73%	81.87% / 74.05%	83.22% / 82.85%
	Knockoff [45]	80.53% / 71.56%	83.27% / 82.89%	82.43% / 80.09%	80.05% / 79.91%	77.08% / 72.74%	85.65% / 84.99%	81.87% / 77.69%	83.22% / 83.06%
MobileNetV2-HP	Hinton et al. [23]	95.32% / 79.82%	88.94% / 87.03%	91.48% / 82.72%	83.36% / 81.22%	90.19% / 81.46%	87.53% / 84.02%	96.35% / 76.88%	86.19% / 85.23%
	Papernot et al. [46]	95.32% / 84.67%	88.94% / 86.50%	91.48% / 86.39%	83.36% / 82.84%	90.19% / 85.80%	87.53% / 86.79%	96.35% / 87.24%	86.19% / 85.77%
	Knockoff [45]	95.32% / 88.15%	88.94% / 87.72%	91.48% / 87.68%	83.36% / 82.97%	90.19% / 86.62%	87.53% / 87.28%	96.35% / 89.10%	86.19% / 86.04%
InceptionV3-HP	Hinton et al. [23]	93.65% / 74.29%	84.00% / 82.38%	95.03% / 85.44%	76.71% / 75.06%	92.53% / 82.66%	90.20% / 88.30%	93.40% / 75.16%	67.07% / 64.60%
	Papernot et al. [46]	93.65% / 77.81%	84.00% / 82.76%	95.03% / 87.95%	76.71% / 75.74%	92.53% / 84.13%	90.20% / 89.56%	93.40% / 83.55%	67.07% / 66.32%
	Knockoff [45]	93.65% / 80.54%	84.00% / 83.09%	95.03% / 88.28%	76.71% / 76.59%	92.53% / 87.58%	90.20% / 89.86%	93.40% / 86.82%	67.07% / 66.85%
EfficientNetV2-HP	Hinton et al. [23]	92.70% / 76.33%	87.18% / 86.25%	94.15% / 86.63%	82.93% / 80.49%	93.84% / 82.45%	89.78% / 87.04%	95.61% / 78.30%	85.92% / 84.41%
	Papernot et al. [46]	92.70% / 79.75%	87.18% / 86.41%	94.15% / 88.59%	82.93% / 81.06%	93.84% / 86.70%	89.78% / 88.27%	95.61% / 85.20%	85.92% / 85.26%
	Knockoff [45]	92.70% / 80.26%	87.18% / 87.10%	94.15% / 89.24%	82.93% / 81.98%	93.84% / 87.73%	89.78% / 89.12%	95.61% / 88.96%	85.92% / 85.57%
MobileNetV2-FFKEW	Hinton et al. [23]	98.83% / 85.72%	89.02% / 88.35%	96.39% / 84.55%	84.80% / 83.61%	96.51% / 86.80%	87.59% / 86.62%	98.29% / 85.38%	86.89% / 85.48%
	Papernot et al. [46]	98.83% / 88.90%	89.02% / 87.68%	96.39% / 87.32%	84.80% / 83.22%	96.51% / 89.14%	87.59% / 86.83%	98.29% / 89.50%	86.89% / 86.17%
	Knockoff [45]	98.83% / 92.46%	89.02% / 87.91%	96.39% / 89.04%	84.80% / 83.54%	96.51% / 92.97%	87.59% / 87.45%	98.29% / 93.22%	86.89% / 86.56%
InceptionV3-FFKEW	Hinton et al. [23]	94.40% / 82.58%	84.27% / 83.15%	95.96% / 86.58%	77.20% / 76.39%	92.60% / 84.52%	91.04% / 89.27%	94.08% / 86.47%	68.43% / 67.29%
	Papernot et al. [46]	94.40% / 86.31%	84.27% / 83.64%	95.96% / 89.41%	77.20% / 76.18%	92.60% / 87.49%	91.04% / 89.68%	94.08% / 88.65%	68.43% / 67.10%
	Knockoff [45]	94.40% / 90.28%	84.27% / 83.52%	95.96% / 90.84%	77.20% / 76.75%	92.60% / 89.88%	91.04% / 89.95%	94.08% / 91.73%	68.43% / 67.83%
EfficientNetV2-FFKEW	Hinton et al. [23]	98.19% / 85.87%	88.75% / 87.30%	98.04% / 88.69%	83.99% / 81.45%	97.21% / 86.23%	90.45% / 88.36%	98.41% / 85.58%	85.98% / 84.70%
	Papernot et al. [46]	98.19% / 89.42%	88.75% / 87.69%	98.04% / 89.92%	83.99% / 82.52%	97.21% / 88.93%	90.45% / 89.11%	98.41% / 90.15%	85.98% / 85.34%
	Knockoff [45]	98.19% / 93.66%	88.75% / 87.86%	98.04% / 91.13%	83.99% / 82.37%	97.21% / 90.44%	90.45% / 89.76%	98.41% / 92.94%	85.98% / 85.63%

class for the demonstration purpose. If there are multiple fields declared in a table, all fields are mapped accordingly. Moreover, all tables (e.g., "RNNOptions", "Pool2DOptions" and "FullyConnectedOptions") in the schema are mapped through the same manner as shown in the figure. For other data structures, such as enum and union, we directly replicate their constants into corresponding model informative classes.

B Utility Class Implementation

For the sake of consistent presentation, we use the `Conv2DOptions` and `Buffer` model informative classes in Figure 4 as an example. Their utility classes are depicted in Figure 5. The `Conv2DOptionsT` class (Figure 5a) contains four methods that are self describing: (1) `__init__(self)` defines the attributes corresponding to the field matching methods in the `Conv2DOptions` class. (2) `InitFromBuf(cls, buf, pos)` initializes an instance of the `Conv2DOptions` class with data from a FlatBuffers buffer at a given position. (3) `InitFromObj(cls, conv2DOptions)` initializes an instance of the `Conv2DOptionsT` class with data from the `Conv2DOptions` instance. (4) `_UnPack(self, conv2DOptions)` sets the attributes of the `Conv2DOptionsT` instance to the values returned by the corresponding field matching methods in the `Conv2DOptions` instance. The idea behind these methods is to extract the values (e.g., padding value) of a particular data structure (e.g., Conv2DOptions) based on its position in a FlatBuffers buffer (the victim model), and organize the extracted values into a corresponding instance (e.g., `x`) that enables us to modify later. Similarly, the `BufferT` class (Figure 5b) also has such methods, with the major difference in the `_UnPack(self, buffer)`, where the data for a particular operator (e.g., parameters of a Conv2D layer) can be extracted in accordance with its index in a FlatBuffers buffer.

C Data Serialization Function For Utility Class

Following the same illustrative example in Figure 5, the data serialization functions for `Conv2DOptionsT` and `BufferT` classes are depicted in Listing 1. We start with the methods added to the `Conv2DOptionsT` class. In lines 1-4, `Conv2DOptionsStart(builder)` initializes bookkeeping for writing a new `Conv2DOptions` instance, where the positional argument `builder` accepts an instance of the `Builder`[7] class used for constructing a FlatBuffers buffer. Within the method, `builder.StartObject(6)` indicates the number of

[7]https://github.com/google/flatbuffers/tree/master/python

attributes to be written. Here we have six attributes that align to the FlatBuffers table "Conv2DOptions" in Figure 4. Next, `Conv2DOptionsAddPadding(builder, padding)` adds a passed in padding value into the `builder` used by the `Conv2DOptionsStart(builder)`. Note that we only present one attribute-related method for ease of presentation, other methods can refer to our detailed implementation[8]. Finally, `Conv2DOptionsEnd(builder)` serializes the data contained in the `Conv2DOptions` instance into a FlatBuffers buffer using the same `builder` as before. To make it easy to use, we integrate all aforementioned methods into the `Pack(self, builder)`, as shown in lines 6-11. For the `BufferT` class, it has five methods from lines 14 to 32, namely `BufferStart(builder)`, `BufferAddData(builder, data)`, `BufferStartDataVector(builder, numElems)`, `BufferEnd(builder)`, and `Pack(self, builder)`. Among these methods, the `BufferStartDataVector(builder, numElems)` is designed specifically for writing model parameters as a deep learning model stores its parameters in matrix (vectors) form. The remaining methods are similar to those in the `Conv2DOptionsT` class.

D Additional Experimental Results

More experimental results are shown in Table 7, Table 8, and Table 9.

[8]https://github.com/Jinxhy/THEMIS