MMGuard: An Automated Tool for Protecting On-Device Deep Learning Models in Android Apps

Jiayi Hua[1], Yuanchun Li[2], Haoyu Wang[1]
[1] Beijing University of Posts and Telecommunications, China [2] Microsoft Research

***Abstract*—On-device deep learning models have shown growing popularity in mobile apps, which allows offline model inference while preserving user privacy. However, on-device deep learning models also introduce security challenges, i.e., the trained models can be easily stolen by attackers and even be tampered. Previous work suggested that most of the on-device models are lacking of protection, i.e., can be stolen by decompiling the apps directly. In this work, we present MMGuard, an automated framework for building mutual authentication between mobile apps and DNN models, which can thus protect on-device models from being easily attacked. Unlike existing model protect methods, our approach does not require model training or any prior knowledge of training data. The key idea of MMGuard is to verify the deep learning model in the app before inference, i.e., feeding owner- and app- related information to it, which can greatly increase the effort of model hacking. We evaluate our tool on 5 popular image classification DNNs and 58 real world apps. Experiment results suggest that MMGuard introduces negligible latency on models and can be automatically applied to real world Android apps. The full paper of this work is in submission to a top-tier software engineering venue. The demo video of this tool is available at: https://v.youku.com/v_show/id_XNDkzODM5ODE0OA==.html**

I. Introduction

Deep learning models are increasingly used in mobile apps as critical components to provide artificial intelligence features such as facial recognition, natural language processing, recommendation, and speech recognition, etc.

Performing deep learning tasks on the server side has always been criticized due to issues including privacy and network latency. Thus, on-device deep learning models are gaining popularity in mobile apps, which offer benefits desirable for both mobile users and app developers. For app developers, they are relieved from the expense of maintaining cloud service or server. For app users, they do not need to concern the privacy issue and the network latency.

However, on-device models inevitably introduce new security challenges. The models are stored locally on user devices, which can be easily stolen and attacked, as illustrated in Figure 1. First, training a DNN model often requires lots of resource, including proprietary training data, algorithm, and computational infrastructure. If the DNN model is unprotected in the app, attacker who has basic reverse engineering ability can easily extract it without much effort, and reuse it in his own mobile apps, infringing the intellectual property of the model owner. Second, DNNs have been found vulnerable to various kinds of attacks, like backdoor attack [1, 2], which injects a backdoor into a model to make it behave normally in most times while behave exceptionally when certain trigger appears. Thus, attackers can modify the extracted DNN models

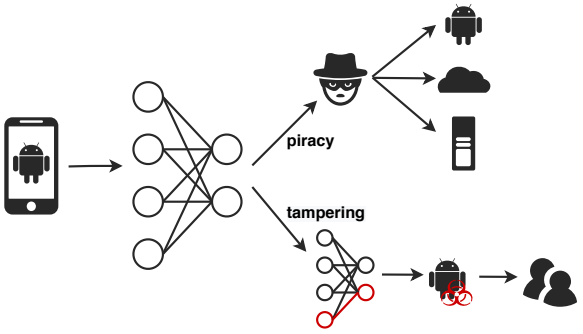


Fig. 1: Security issues introduced by on-device models.

(e.g., injecting a backdoor) and repackage the app to infect unsuspecting users. Unfortunately, recent work suggested that most of the on-device models are lacking of protection, and no systematic tools can be easily used to protect the models effectively. Thus, it motivates us to create an automated tool to protect the models from being stolen or tampered.

This demonstration presents MMGuard, a tool for protecting DNN models in Android apps by encoding the unique features extracted from the app to the DNN model. Note that, we do not require model re-training or any prior knowledge of the training data. Our tool works on the black-box models. To be specific, we first decode the data-flow graph from the original model file, then modify the nodes containing valuable weights by manipulating the data-flow graph. We next generate a new model from the modified data-flow graph to replace the original one. The new model will take app-related signature information as additional input during the model inference process. By enhancing the code logic of mobile DL framework, for example, TensorFlow [3], the app is enforced to verify the DNN model before inference and the signature information extracted from the app binaries is injected to the model in advance. A DNN model can provide correct inference results only when the app has correct signature information input to it, otherwise the inference accuracy will be significantly deteriorated. Furthermore, the app has to attest the integrity of DNN model before inference, and the execution will be terminated if authentication fails.

MMGuard is implemented as a fully automated tool, i.e., with the original app embedded with unprotected model as input, and the output is the app embedded with protected DNN model. We believe MMGuard can promote best operational practices across app developers.

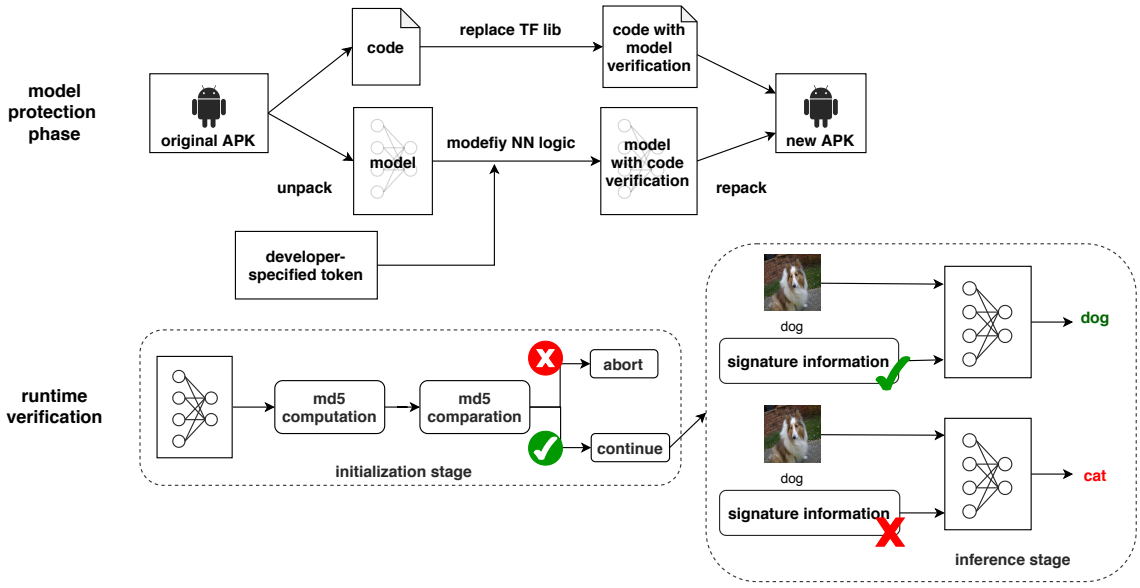


Fig. 2: Overview of MMGuard.

II. Tool Design

As shown in Figure 2, MMGuard consists of two procedures, *model protection* and *runtime verification*. MMGuard protects the DNN model in an Android app by injecting verification logic into both the DL library and the model. To use MMGuard, the developer is asked to provide an APK file and a secret token (e.g., the signature of the app as default). MMGuard first disassembles the code and model from the APK file. An extra input node is inserted into the model, and a random convolution layer in the model is selected as the target to inject verification logic. The static weights in the convolution layer are replaced with variables computed with the extra input at runtime. The weights can be correctly computed if and only if the extra input matches the secret token. Meanwhile, MMGuard replaces the original TensorFlow library in the app code with a modified one that contains additional model verification logic. At runtime, the modified library checks the model signature during initialization, and feeds the normal input and the extra secret token into the model during inference. Thus, the DNN model in the app is protected against both model backdoor attack (the backdoored model would fail upon signature verification) and model stealing (the stolen model would not behave correctly without the secret token).

A. Notations

Let f represents the original DNN model, and x be an input of the model. The prediction result of x is $y = f(x)$. The model after modification is denoted as f'. app is the Android app that contains f, and s_{app} represents the signature of app defined by app developer. Function $KEYGEN$ is used to generate an app-specific key key_{app}, which will be injected to f'. We use y' as the symbol of $f'(x)$, the prediction result of f' forthe given input x.

B. Model Protection Procedure

The model protection procedure consists of five main steps:

(1) The model owner creates a message that only he knows for his app as the evidence to prove his ownership. Then he selects one or more convolution layers from his model f where the weights are valuable.

(2) We create the signature s_{app} by hashing the message, and extract data-flow graph from the model to get information of each layer. Then we use pseudorandom random generating (PRG) with seed s_{app} as an instance of $KEYGEN$ to generate key_{app}. Here the length of key_{app} depends on the size of weights of selected convolution layer. The s_{app} and layer number are indispensible in our protection approach.

(3) For each selected convolutional layer, we resize key_{app} to a tensor the same dimensions as weights, then we subtract key_{app} from the weights, saving the results as new weights. For each convolutional layer, we add an input branch and tensor addition operator in order to recovery correct weight during running procedure later. The modification of convolutional layer is depicted in Figure 3. After modification, we recompile the model and get f'.

(4) Compute the md5 value of f'. Then combine the signature, md5 value and selected layer number into a *inputinfo* file, which will be stored in security memory, like TrustZone or server, when app is deployed.
Recompile the model, f', from the modified data-flow graph and replace f in apk file.

(5) We change the code logic of initializing and running a TensorFlow model. And we compile a new TensorFlow framework with the same user interface to replace the original one. This ensures that the users will not feel any changes in usage. Replace f with f' and the original DL framework, then repack apk to get the protected app.

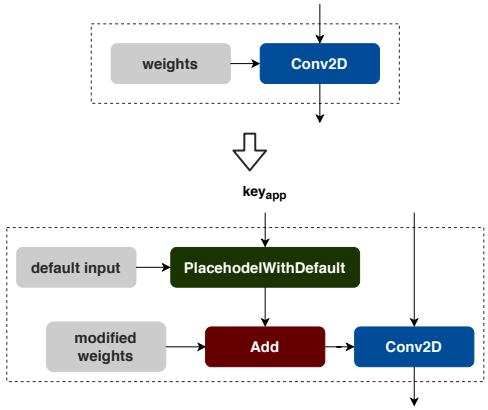


Fig. 3: The structure of convolutional layer before and after modification

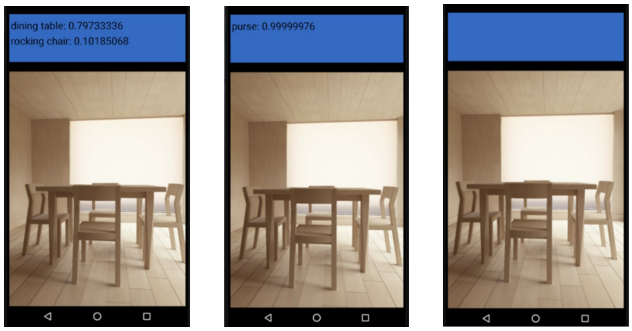
C. Runtime Verification Procedure

There are mainly two steps in the runtime verification.

1) *Initialization*. When user triggers the DNN functionality at the first time, the app initializes the inference interface. During initialization, the app extracts *inputinfo* file and reads the md5 value, computes a md5 of existing f'. If these two md5 values are matched, the app continues to read signature and selected layer number from *inputinfo* file, otherwise it will abort initialization.
2) *Runtime protection*. Every time the app runs f', it first gets original input x and feeds to f'. Then for each selected convolutional layer, the app feeds key_{app} to the input branch where key_{app} is calculated by PRG using s_{app} as seed. Finally the app fetches y' from f' as usual. The performance of f' will be significantly deteriorated, if the signature is wrong.

III. Example

As shown in Figure 4, we take an image classification app to show the effectiveness of MMGuard. We input the app into MMGuard, and the model protection can be performed automatically. We next compare the app in different scenarios including normal, model stealing and model tampering. Figure 4(a) shows the normal results of the app, where the app classifies an image of table and chairs as "dinning table" and "rocking chair". If an attacker intends to steal the model and deploy it in his own app, he may decompile the apk and extract the model file directly. But the weights stored in the model have been modified already. Therefore, even if the attacker can successfully run the model in his own app, the model cannot work as expected, as shown in Figure 4(b) (the image is mis-classified as a purse). If the attacker tries to tamper the original model to achieve some malicious purpose (e.g., inserting a backdoor) and repackage the apk with the original model replaced, the result is shown in Figure 4(c). Any small alterations in model will lead to big change of md5 value, i.e., the runtime verification would be failed. Thus, the execution will be terminated, leading to no classification results provided.



(a) normal (b) model stealing (c) model tampering

Fig. 4: An example to show the effectiveness of MMGuard.

IV. Evaluation

We mainly evaluate MMGuard from two aspects, the performance overhead and the usability on real world apps. We investigate the performance of MMGuard on five popular neural network models including InceptionV3, MobileNet, NasNetMobile, ResNet50 and DenseNet121. To evaluate the scalability on real-world apps, we collected 58 apps that include on-device models trained with Tensorflow framework.

A. Performance Overhead

The performance overhead is measured by considering both the initialization overhead and the runtime protection overhead. We run 50 times for each type of model and the average time is reported in Table I and Table II.

Initialization Overhead. Overall, the overhead of initialization time is around 449 ms on average, as shown in Table I, which we think is tolerable because it is the one-time overhead. The delay is brought mainly by the increasing in model size and the computing of md5 value. We notice that the ResNet50 has the largest latency difference mainly because it has the biggest model size. We use *placeholder_with_default* in input branch, which needs to store a tensor as default input value. Moreover, we modify all convolutional layers in order to hide selected layers among them to increase cost of attack. In real-world usage scenario, app developers can choose *placeholder* operation or fewer layers to reduce initialization overhead.

Runtime Overhead. The runtime overhead is negligible, with a maximum of 16 ms (3%), as shown in II. The additional latency is manly brought by calculating of key_{app} and the extra input feeding. ResNet50 and DenseNet121 have larger latency difference because their layers we protect have more weights, which need more time to calculate key_{app}. The more inputs we compute, the bigger the gap will be between latency overhead of modified model and the regular model. However, we believe the runtime overhead is negligible, which cannot be perceived by app users.

B. Evaluation on Real world Apps

To evaluate whether MMGuard can be adopted to real world apps, we use MMGuard to protect the DNN models in 58 popular apps collected from Google.

TABLE I: The Latency Comparison for Initialization Stage.

Model	Original(ms)	Modified(ms)
InceptionV3	1131	1830(+62%)
MobileNet	244	356(+50%)
NasNetMobile	441	677(+54%)
ResNet50	1388	2340(+69%)
DenseNet121	474	721(+52%)

TABLE II: The Latency Comparison for Running Stage.

Model	Original(ms)	Modified(ms)
InceptionV3	425	428(+0.7%)
MobileNet	227	229(+0.9%)
NasNetMobile	579	584(+0.9%)
ResNet50	540	556(+3.0%)
DenseNet121	620	636(+2.6%)

Overall, 37 of the 58 apps can be protected successfully by MMGuard without providing any additional information. We further manually investigate why MMGuard fails to injecting the verification logic of the remaining apps and observe the following reasons. **(1) Model decoding error.** Some apps were failed when we tried to decode the model may because the model used a customized file format, the app used an unknown version of deep learning framework or the model was broken. **(2) Unsupported model type.** We choose convolutional layer as protect target, which is mainly used in CNN for image processing. Some apps may be designed for task like text classification or used a variety of convolutional layer. **(3) Unsupported OP type.** Some apps used OPs that were not registered in the DL framework binary running on our computer. The failures are mainly caused by the compatibility of our proof-of-concept implementation, which can be easily addressed with the support of app developers. Our tool can be easily adapted to other types of DL tasks and the model owner can easily add supports for more types of OPs. Therefore, in real-world usage scenario, app developers can use MMGuard to automate the model protection on their released apps conveniently.

V. Discussion and Usage Scenario

Prior work had discussed mainly two types of methods for model ownership protection. The first type of method embed model owner's signature into weights by training model with additional regularization terms [4]. The second method is training model using adversarial samples and specific labels [5]. The owner's signature can be represented by the input sample and its unique predict result. However, there are some limitations in existing tools. First, almost all the methods need to be applied during training (or fine-tuning) phase. Thus, they are inapplicable for a released app as its model's parameters are frozen and training metadata is unknown and they may sacrifice the accuracy of the model. Second, most methods can not prevent the model from being stolen, but only can help to prove ownership of the model after suffering losses. Moreover, it has been proved that some methods are not robust enough when facing distillation Attack [6], removal attacks and ambiguity attacks [7]. This means the detection accuracy of signatures embedded in DNN models may be greatly reduced, or it is possible to forge counterfeit signatures.

Unlike existing methods, our tool does not require model training or any knowledge of training data in that we can directly modify compiled models. Our tool will terminate model utilization before piracy or tampering behaviors cause any loss. Also, we do not need to change any code of the mobile app itself, as all the changes are in DL framework. That is to say, our method can be not only used in develop stage of apps, but also applied on released apps. App users and developers could all benefit from MMGuard. App users could be prevented from bad user experiences or security threats, and app developers can protect their intellectual property.

VI. Conclusion

We present a tool, MMGuard, to protect on-device DNN models mainly by adding extra input branches which taking owner-related signature information as input. We can turn the problem of protecting a large model file into the problem of protecting a small signature information. The performance of model will be significantly deteriorated when feeding wrong signature information. Unlike other existing methods which may be time-consuming and do harm to model accuracy, the approach in MMGuard does not require model training or any knowledge of training data. Experiment results show that MMGuard is both available for apps under developing and already released with negligible impact on user experience.

References

[1] T. Gu, B. Dolan-Gavitt, and S. Garg, "Badnets: Identifying vulnerabilities in the machine learning model supply chain," *arXiv preprint arXiv:1708.06733*, 2017.

[2] Y. Liu, S. Ma, Y. Aafer, W.-C. Lee, J. Zhai, W. Wang, and X. Zhang, "Trojaning attack on neural networks," in *25th Annual Network and Distributed System Security Symposium (NDSS)*, 2018, pp. 18–221.

[3] Google, "Tensorflow," https://www.tensorflow.org/, 2020.

[4] Y. Uchida, Y. Nagai, S. Sakazawa, and S. Satoh, "Embedding watermarks into deep neural networks," in *Proceedings of the 2017 ACM on International Conference on Multimedia Retrieval*, ser. ICMR '17, 2017, p. 269–277.

[5] Y. Adi, C. Baum, M. Cisse, B. Pinkas, and J. Keshet, "Turning your weakness into a strength: Watermarking deep neural networks by backdooring," in *Proceedings of the 27th USENIX Conference on Security Symposium*, ser. SEC'18. USA: USENIX Association, 2018, p. 1615–1631.

[6] Z. Yang, H. Dang, and E. Chang, "Effectiveness of distillation attack and countermeasure on neural network watermarking," *CoRR*, vol. abs/1906.06046, 2019. [Online]. Available: http://arxiv.org/abs/1906.06046

[7] L. Fan, K. Woh Ng, and C. S. Chan, "[Extended version] Rethinking Deep Neural Network Ownership Verification: Embedding Passports to Defeat Ambiguity Attacks," *arXiv e-prints*, p. arXiv:1909.07830, Sep. 2019.