Mind Your Weight(s): A Large-scale Study on Insufficient Machine Learning Model Protection in Mobile Apps

Zhichuang Sun
Northeastern University
sun.zhi@husky.neu.edu

Ruimin Sun
Northeastern University
r.sun@northeastern.edu

Long Lu
Northeastern University
l.lu@northeastern.edu

Abstract

On-device machine learning (ML) is quickly gaining popularity among mobile apps. It allows offline model inference while preserving user privacy. However, ML models, considered as core intellectual properties of model owners, are now stored on billions of untrusted devices and subject to potential thefts. Leaked models can cause both severe financial loss and security consequences.

This paper presents the first empirical study of ML model protection on mobile devices. Our study aims to answer three open questions with quantitative evidence: How widely is model protection used in apps? How robust are existing model protection techniques? How much can (stolen) models cost? To that end, we built a simple app analysis pipeline and analyzed 46,753 popular apps collected from the US and Chinese app markets. We identified 1,468 ML apps spanning all popular app categories. We found that, alarmingly, 41% of ML apps do not protect their models at all, which can be trivially stolen from app packages. Even for those apps that use model protection or encryption, we were able to extract the models from 66% of them via unsophisticated dynamic analysis techniques. The extracted models are mostly commercial products and used for face recognition, liveness detection, ID/bank card recognition, and malware detection. We quantitatively estimated the potential financial impact of a leaked model, which can amount to millions of dollars for different stakeholders.

Our study reveals that on-device models are currently at high risk of being leaked; attackers are highly motivated to steal such models. Drawn from our large-scale study, we report our insights into this emerging security problem and discuss the technical challenges, hoping to inspire future research on robust and practical model protection for mobile devices.

1 Introduction

Mobile app developers have been quickly adopting *on-device machine learning* (ML) techniques to provide artificial intelligence (AI) features, such as facial recognition, augmented/virtual reality, image processing, voice assistant, etc. This trend is now boosted by new AI chips available in the latest smartphones [1], such as Apple's Bionic neural engine, Huawei's neural processing unit, and Qualcomm's AI-optimized SoCs.

Compared to performing ML tasks in the cloud or on the server side, on-device ML (mostly model inference) offers unique benefits desirable for mobile users as well as app developers. For example, it avoids sending (private) user data to the cloud and does not require network connection or consume much mobile data. For app developers or ML solution providers, on-device ML greatly reduces the computation load on their servers.

On-device ML inference inevitably stores ML models locally on user devices, which however creates a new security challenge. Commercial ML models used in apps are often part of the core intellectual property (IP) of vendors. Such models may fall victim to theft or abuse, if not sufficiently protected. In fact, on-device ML makes model protection much more challenging than server-side ML because models are now stored on user devices, which are fundamentally untrustworthy and may leak models to curious or malicious parties.

The consequences of model leakage are quite severe. First, with a leaked model goes away the research and development investment made by the model owner, which often includes human, data, and computing costs. Second, when a proprietary model is obtained by unethical competitors, the model owner loses the competitive edge or pricing advantage for its products, causing further financial loss. Third, a leaked model facilitates malicious actors to find adversarial inputs to bypass or confuse the ML systems that use the model, which can lead to not only reputation damages to the vendor but also critical failures in their products (*e.g.,* fingerprint recognition bypass).

This paper presents the first large-scale study of ML model protection and theft on mobile devices. Our study aims to shed lights on the lesser understood risks and costs of model leakage/theft in the context of on-device ML. We present our study that answers the following questions with ample empirical evidence and observations.

- **Q1: How widely is model protection used in apps?**
- **Q2: How robust are existing model protection tech-**

niques?

- **Q3: How much can (stolen) models cost?**

To answer these questions, we collected 46,753 trending Android apps from the US and the Chinese app markets. To answer Q1, we built a simple and automatic pipeline to first identify the ML models and SDK/frameworks used in an app, and then detect if the ML models are encrypted. Among all the collected apps, we found 1,468 apps that use on-device ML, and 602 (41%) of them do not protect their ML models at all (*i.e.,* models are stored in plaintext form on devices). Most of these apps have high installation counts (greater than 10M) and span the top-ten app categories, which underlines the limited awareness of model thefts and the need for model protection among app developers.

To answer Q2, for the encrypted models, we dynamically run the corresponding apps and built an automatic pipeline to identify and extract the decrypted ML models from memory. This pipeline represents an unsophisticated model theft attack that an adversary can realistically launch on her own device. We found that the same protected models can be reused/shared by multiple apps, and a set of 18 unique models extracted from our dynamic analysis can affect 347 apps (43% of all the apps with protected models). These apps cover a wide range of ML frameworks, including TensorFlow, TFLite, Caffe, SenseTime, Baidu, Face++, etc. They use ML for various purposes, including face tracking, liveness detection, OCR, ID card and bank card recognition, photo processing, and even malware detection.

We also observed some interesting cases where a few model owners did extra to protect their models, such as encrypting both code and model files, encrypting model files multiple times, or encrypting feature vectors. Despite the efforts, these models can be successfully extracted in memory in plaintext. These cases indicate that model owners or app developers start realizing the risk of model thefts but no standard and robust model protection technique exists, which echos the urgent need for research into on-device model protection.

Finally, to answer Q3, we present an analysis on the financial impact of model leakage on the stakeholders, including app developers and model providers. We identify three major impact sources: the research and development investment on ML models, the financial loss due to competition, and the financial loss due to model evasion. We studied several representative apps and found that the potential financial loss can be as high as millions of dollars, depending on the app revenue and the nature of the models.

By performing the large-scale study and finding answers to the three questions, we intend to raise the awareness of the model leak/theft risks, which apps using on-device ML are facing even if models are encrypted. Our study shows that the risks are realistic due to absent or weak protection of on-device models. It also shows that attackers are not only technically able to, but also highly motivated to steal or abuse on-device ML models. We share our insights and call for future research to address this emerging security problem.

In summary, the contributions of our research are:

- We apply our analysis pipeline on 46,753 Android apps collected from US and Chinese app markets. We found that among the 1,468 apps using on-device ML, 41% do not have any protection on their ML models. For those do, 66% of them still leak their models to an unsophisticated runtime attack.
- We provide a quantified estimate on the financial impact of model leakage based on case studies. We show that the financial loss caused by a stolen model can be as high as millions of dollars, indicating tremendous incentives for attackers.
- Our work calls for research on robust protection mechanisms for ML models on mobile devices. We share our insights gained during the study to inform and assist future work on this topic.

The rest of the paper is organized as follows. Section 2 introduces the background knowledge about on-device ML. Section 3 presents an overview of our analysis pipeline. Sections 4, 5, and 6 answers the questions Q1, Q2, and Q3, respectively. Section 7 discusses the current model protection practices and effectiveness, the research insights and the limitations of our analysis. Section 8 surveys the related work and Section 9 concludes the paper.

2 Background

The Trend of On-device Machine Learning: Currently, there are two ways for mobile apps to use ML: cloud-based and on-device. In cloud-based ML, apps send requests to a cloud server, where the ML inference is performed, and then retrieve the results. The drawbacks include requiring constant network connections, unsuitable for real-time ML tasks (e.g., live object detection), and needing raw user data uploaded to the server. Recently, on-device ML inference is quickly gaining popularity thanks to the availability of hardware accelerators on mobile devices and the the ML frameworks optimized for mobile apps. On-device ML avoids the aforementioned drawbacks of cloud-based ML. It works without network connections, performs well in real-time tasks, and seldom needs to send (private) user data off the device. However, with ML inference tasks and ML models moved from cloud to user devices, on-device ML raises a new security challenge to model owners and ML service providers: how to protect the valuable and proprietary ML models now stored and used on user devices that cannot be trusted.

The Delivery and Protection of On-device Models :

Typically, on-device ML models are trained by app developers or ML service providers on servers with rich computing resources (e.g., GPU clusters and large storage servers). Trained models are shipped with app installation packages. A model can also be downloaded separately after app installation to reduce the app package size. Model inference is performed

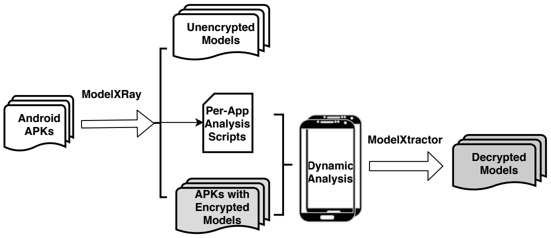


Figure 1: Overview of Static-Dynamic App Analysis Pipeline

by apps on user devices, which relies on model files and ML frameworks (or SDKs). To protect on-device models, some developers encrypt/obfuscate them, or compile them into app code and ship them as stripped binaries [7, 25]. However, such techniques only make it difficult to reverse a model, rather than strictly preventing a model from being stolen or reused.

On-device Machine Learning Frameworks: There are tens of popular ML frameworks, such as Google TensorFlow and TensorFlow Lite [26], Facebook PyTorch and Caffe2 [5], Tencent NCNN [25], and Apple Core ML [8]. Among these frameworks, TensorFlow Lite, Caffe2, NCNN and Core ML are particularly optimized for mobile apps. Different frameworks use different file formats for storing ML models on devices, including ProtoBuf (.pb, .pbtxt), FlatBuffer (.tflite), MessagePack (.model), pickle (.pkl), Thrift (.thrift), etc. To mitigate model reverse engineering and leakage, some companies developed customized or proprietary model formats [46, 54].

On-device Machine Learning Solution Providers: For cost efficiency and service quality, app developers often use third-party ML solutions, rather than training their own models or maintaining in-house ML development teams. The popular providers of ML solutions and services include Face++ [13] and SenseTime [28], which sell offline SDKs (including on-device models) that offer facial recognition, voice recognition, liveness detection, image processing, Optical Character Recognition (OCR), and other ML functionalities. By purchasing a license, app developers can include such SDKs in their apps and use the ML functionalities as black-boxes. ML solution providers are more motivated to protect their models because model leakage may severely damage their business [28].

3 Analysis Overview

On-device ML is quickly being adopted by apps, while its security implications on model/app owners remain largely unknown. Especially, the threats of model thefts and possible ways to protect models have not been sufficiently studied. This paper aims to shed lights on this issue by conducting a large-scale study and providing quantified answers to three questions: How widely is model protection used in apps? (§4) How robust are existing model protection techniques? (§5) How much can (stolen) models cost? (§6)

To answer these questions, we built a static-dynamic app analysis pipeline. We note that this pipeline and the analysis techniques are kept simple intentionally and are not part of the research contributions of this work. The goal of our study is to understand how easy or realistic it is to leak or steal ML models from mobile apps, rather than demonstrating novel or sophisticated app analysis and reverse-engineering techniques. Our analysis pipeline represents what a knowledgeable yet not extremely skilled attacker can already achieve when trying to steal ML models from existing apps. Therefore, our analysis result gives the lower bound of (or a conservative estimate on) how severe the model leak problem currently is.

The workflow of our analysis is depicted in Figure1. Apps first go through the static analyzer, ModelXRay, which detects the use of on-device ML and examines the model protection, if any, adopted by the app. For apps with encrypted models, the pipeline automatically generates the analysis scripts and send them to the dynamic analyzer, ModelXtractor, which performs a non-sophisticated form of in-memory extraction of model representations. ModelXtractor represents a realistic attacker who attempts to steal the ML models from an app installed on her own phone. Models extracted this way are in plaintext formats, even though they exist in encrypted forms in the device storage or the app packages. Our evaluation of ModelXRay and ModelXtractor (§4.3 and §5.3) shows that they are highly accurate for our use, despite the simple analysis techniques. We report our findings and insights drawn from the large-scale analysis results produced by ModelXRay and ModelXtractor in §4.4 and §5.4, respectively.

We identified three possible financial impacts of model leakages: the Research & Development(R&D) cost, the financial loss to competition, and the financial loss to model evasion. We built an estimation framework and used public data from the Internet to estimate the financial loss of model leakage (§6).

4 Q1: How Widely Is Model Protection Used in Apps?

4.1 Android App Collection

We collect apps from three Android app markets: Google Play, Tencent My App, and 360 Mobile Assistant. They are the leading Android app stores in the US and China [29]. We download the apps labeled *TRENDING* and *NEW* across all 55 categories from Google Play (12,711), and all recently updated apps from Tencent My App (2,192) and 360 Mobile Assistant (31,850).

4.2 Methodology of ModelXRay

ModelXRay statically detects if an app uses on-device ML and whether or not its models are protected or encrypted. ModelXRay is simple by design and adopts a best-effort detection strategy that errs on the side of soundness (*i.e.,* low false positives), which is sufficient for our purpose of analyzing model leakage.

The workflow of ModelXRay is shown in Figure 2. For a

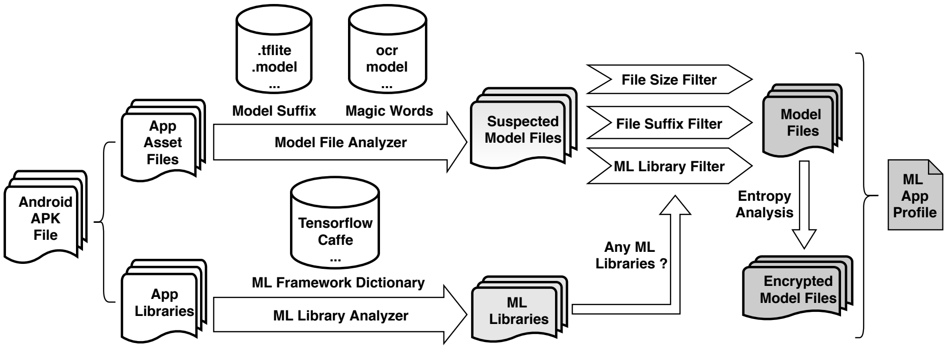


Figure 2: Identify Encrypted Models with ModelXRay

ModelXRay extracts an app's asset files and libraries from the APK file, analyzes the native libraries and asset files to identify ML frameworks, SDK libraries and model files. Then it applies model filters combining file sizes, file suffixes and ML libraries to reduce false positives and use entropy analysis to identify encrypted models.

given app, ModelXRay disassembles the APK file and extracts the app asset files and the native libraries. Next, it identifies the ML libraries/frameworks and the model files as follows:

ML Frameworks and SDK Libraries: On-device model inference always use native ML libraries for performance reasons. Inspired by Xu's work [54], we use keyword searching in binaries for identifying native ML libraries. ModelXRay supports a configurable dictionary that maps keywords to corresponding ML frameworks, making it easy to include new ML frameworks or evaluate the accuracy of keywords(listed in Appendix A1). Further, ModelXRay supports generic keywords, such as "NeuralNetwork","LSTM", "CNN", and "RNN" to discover unpopular ML frameworks. However, these generic keywords may cause false positives. We evaluate and verify the results in §4.3.

ML Model Files: To identify model files, previous work [54] rely on file suffix match to find models that follow the common naming schemes. We find, however, many model files are arbitrarily named. Therefore, We use a hybrid approach combining file suffix match and path keyword match (*e.g.*, `../models/arbitrary.name` can be a model file). We address false positives by using three filters: whether the file size is big enough (more than 8 KB); whether it has a file suffix that is unlikely for ML models (*e.g.*,`model.jpg`); whether the app has ML libraries.

Encrypted Model Files: We use the standard entropy test to infer if a model file is encrypted or not. High entropy in a file is typically resulted from encryption or compression [11]. For compressed files, we rule them out by checking file types and magic numbers. We use 7.99 as the entropy threshold for encryption in the range of [0,8], which is the average entropy of the sampled encrypted model files (see §4.3). Previous work [54] treats models that cannot be parsed by ML framework as encrypted models, which is not suitable in our analysis and has high false positives for several reasons, such as the lack of a proper parser, customized model formats, aggregated models, *etc.*

ML App Profiles: As the output, ModelXRay generates a profile for each app analyzed. A profile comprises of two parts: ML models and SDK libraries. For ML models, it records file names, sizes, MD5 hash and entropy. In particular, the MD5 hashes help us identify shared/reused models among different apps (as discussed in §4.4).

For SDK libraries, we record framework names, the exported symbols, and the strings extracted from the binaries. They contain information about the ML functionalities, such as OCR, face detection, liveness detection. Our analysis pipeline uses such information to generate the statistics on the use of ML libraries (§4.4).

4.3 Accuracy Evaluation of ModelXRay

Accuracy of Identifying ML Apps: To establish the ground truth for this evaluation, we manually selected and verified 219 ML apps as the true positives. We included 219 non-ML apps labeled by [54] as the true negatives. We evaluated ModelXRay on this set of 438 apps. It achieved a false negative rate of 6.8% (missed 30 ML apps) and a false positive rate of 0% (zero non-ML apps is classified as ML apps). We checked the 30 missed ML apps, and found out that they are using unpopular ML Frameworks whose keywords are not in the dictionary. We found two ML apps that ModelXRay correctly detected but are missed by [54], one using ULSFaceTracker, which is an unpopular ML framework and the other using TensorFlow.

To further evaluate the false positive rate, we run ModelXRay on our entire set of 46,753 apps and randomly sampled 100 apps labeled by ModelXRay as ML apps (50 apps from Google Play and 50 apps from Chinese app market). We then manually checked these 100 apps and found 3 apps that are not ML apps (false positive rate of 3%). The manual check was done by examining the library's exposed symbols and functions. This relatively low false positive rate shows ModelXRay's high accuracy in detecting ML apps for our large-scale study.

Accuracy of Identifying Models: We randomly sampled 100 model files identified by ModelXRay from Chinese app markets and Google Play, respectively, and manually verified the results. ModelXRay achieved a true positive rate of 91% and 97%, respectively.

In order to evaluate how widely apps conform to model standard naming conventions, we manually checked 100 ML apps from both Google Play and Chinese app market and found 24 apps that do not follow any clear naming conventions. Some use "*.tfl*" and "*.lite*" instead of the normal "*.tflite*" for TensorFlow Lite models. Some use "*3_class_model*" without a suffix. Some have meaningful but not standard suffixes such as "*.rpnmodel*","*.traineddata*". Other have very generic suffixes such as "*.bin*", "*.dat*", and "*.bundle*". This observation shows that file suffix matching alone can miss a lot of model files. Table 1 shows the top 5 popular model file suffixes used in different app markets. Many of these popular suffixes are not standard. ModelXRay's model detection does not solely depend on model file names.

Table 1: Popular model suffix among different app markets

360 Mobile Assistant	Num.Of.Cases	Google Play	Num.Of.Cases
.bin	1860	*.bin*	318
.model	1540	*.model*	175
.rpnmodel	257	*.pb*	93
.binary	212	*.tflite*	83
.dat	201	*.traineddata*	46

Accuracy of Identifying Encrypted Models: To demonstrate the accuracy of using entropy as indicator of encryption, we sampled 10 models files each from 4 popular encodings: ascii text, protobuffer, flatbuffer, and encrypted format. We calculated their entropy values (Figure 3). Obviously, encrypted model files all have distinguishably high entropy close to 8. The other encodings have varying entropy values but all significant lower than 8. Figure 4 shows the entropy distribution of all model files that we collected from 360 App Assistant app market. It shows that typical entropy values of unencrypted model files stay between 3.5 and 7.5.

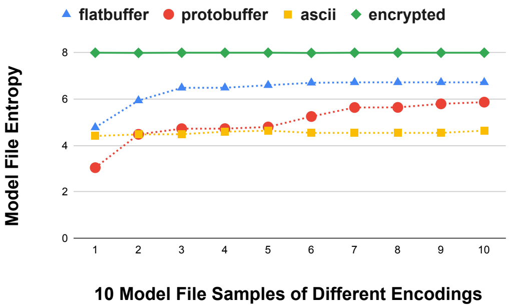


Figure 3: Model File Entropy of 4 Popular Encodings

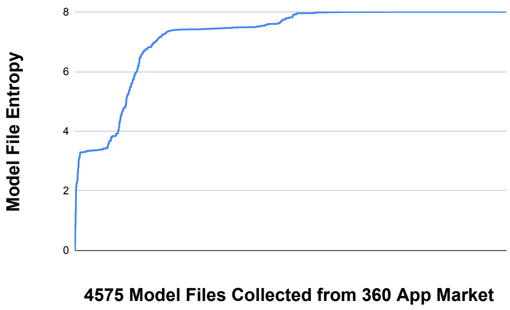


Figure 4: Model File Entropy Distribution of 360 App Market

4.4 Findings and Insights

We now present the results from our analysis as well as our findings and insights, which provide answers to the question "Q1: How widely is model protection used in apps?". We start with the popularity and diversity of on-device ML among our collected apps, which echo the importance of model security and protection. We then compare model protection used in various apps. Especially, we draw observations on how model protection varies across different app markets and different ML frameworks. We also report our findings about the shared encrypted models used in different apps.

Popularity and Diversity of ML Apps: In total, we are able to collect 46,753 Android apps from Google Play, Tencent My App and 360 Mobile Assistant stores. Using ModelXRay, we identify 1,468 apps that use on-device ML and have ML models deployed on devices, which accounts for 3.14% of our entire app collection.

We also measure the popularity of ML apps for each category, based on the intuition that apps from certain categories much more heavily use on-device ML than others. We used the app category information from the three app markets. Table 2 shows the per-category counts for total apps and ML apps (i.e., apps using on-device ML). Our findings are summarized as follows:

On-device ML is gaining popularity in all categories. There are more than 50 ML apps in each of the categories, which suggests the widespread interests among app developers in using on-device ML. Among all the categories, "Business", "Image" and "News" are the top three that see most ML apps. This observation confirms the diversity of apps that make heavy use of on-device ML. It also highlights that a wide range of apps need to protect their ML models and attackers have a wide selection of targets.

More apps from Chinese markets are embracing on-device ML. This is reflected from both the percentage and the absolute number of ML apps: Google Play has 178 (1.40%), Tencent My App has 159 (7.25%), and 360 Mobile Assistant has 1,131 (3.55%).

As we can see from the above findings, Chinese app markets show a significant higher on-device machine learning adoption rate and unique property of per-category popularity, making it a non-negligible dataset for studying on-device machine learning model protection.

Table 2: The number of apps collected across markets.

	Google Play		Tencent My App		360 Mobile Assistant		Total	
Category	All	ML	All	ML	All	ML	All	ML
Business	404	2	99	2	2,450	296	2,953	**300**
News	96	0	102	5	2,450	180	2,648	**185**
Images	349	36	158	23	4,900	156	5,407	**215**
Map	263	4	206	14	2,450	83	2,919	101
Social	438	23	141	17	2,450	79	3,029	119
Shopping	183	5	112	16	2,450	84	2,745	105
Life	1,715	15	193	16	2,450	53	4,358	84
Education	389	3	116	7	2,450	74	2,955	84
Finance	123	6	76	21	2,450	55	2,649	82
Health	317	5	115	3	2,450	42	2,882	50
Other	8,434	79	874	35	4,900	29	14,208	143
Total	12,711	178	2,192	159	31,850	1,131	46,753	1,468

Note: In 360 Mobile Assistant, the number of unique apps is 31,591 (smaller than 32,850) because some apps are multi-categorized. Image category contains 4,900 apps because we merged image and photo related apps.

We measure the diversity of ML apps in terms of ML frameworks and functionalities. We show the top-10 most common functionalities and their distribution across different ML frameworks in Table 3.

On-device ML offers highly diverse functionalities. Almost all common ML functionalities are now offered in the on-device fashion, including OCR, face tracking, hand detection, speech recognition, handwriting recognition, ID card recognition, and bank card recognition, liveness detection, face recognition, iris recognition and so on. This high diversity means that, from the model theft perspective, attackers can easily find targets to steal ML models for any common functionalities.

Long tail in the distribution of ML frameworks used in apps Besides the well-known frameworks such as TensorFlow, Caffe2/PyTorch, and Parrots, many other ML frameworks are used for on-device ML, despite their relatively low market share. For instance, as shown in 3, Tencent NCNN [25], Xiaomi Mace [7], Apache MXNet [3], and ULS from Utility Asset Store [27] are used by a fraction of the apps that we collected. Each of them tends to cover only a few ML functionalities. In addition, there could be other unpopular ML frameworks that our analysis may have missed. This long tail in the distribution of ML frameworks poses a challenge to model protection because frameworks use different model formats, model loading/parsing routines, and model inference pipelines.

Model Protection Across App Stores: Figure 5 gives the per-app-market statistics on ML model protection and reuse. Figure 5a shows the per-market numbers of protected apps (*i.e.,* apps using protected/encrypted models) and unprotected apps (*i.e.,* apps using unprotected models).

Overall, only 59% of ML apps protect their models. The rest of the apps (602 in total) simply include the models in plaintext, which can be easily extracted from the app packages or installation directories. This result is alarming and suggests that a large number of app developers are unaware of model theft risks and fail to protect their models. It also shows that, for 41% of the ML apps, stealing their models is as easy as downloading and decompressing their app packages. We urge stakeholders and security researchers to raise their awareness and understanding of model thefts, which is a goal of this work.

Percentages of protected models vary across app markets. When looking closer at each app market, it is obvious to see that Google Play has the lowest percentage of ML apps using protected models (26%) whereas 360 Mobile Assistant has the highest (66%) and Tencent My App follows closely (59%). These percentages indicate that the apps from the Chinese markets are more active in protecting their ML models, possibly due to better security awareness or higher risks [13, 28].

When zooming into apps and focusing on individual models (*i.e.,* some apps use multiple ML models for different functionalities), the percentages of unprotected models (Figure 5b) become even higher. Overall, 4254 out of 6522 models (77%) are unprotected and thus easily extractable and reverse engineered.

Model Protection Across ML Frameworks: We also derive the per-ML-framework statistics on model protection (Figure 6). The frameworks used by a relatively small number apps, including MXNet, Mace, TFLite, and ULS, are grouped into the "Other" category.

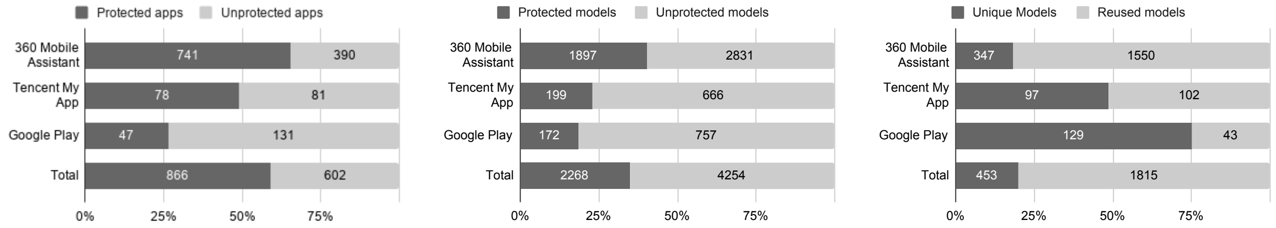
Models on more popular ML frameworks have wider adoption of protection, but still not wide enough. As shows in Figure 6a, more than 79% of the apps using SenseTime (Parrots) have protected models, followed by apps using Caffe (60% of them have protected models). For apps using TensorFlow and NCNN, the number is around 20%. Apps using other frameworks are the least protected against model thefts. This result can be partly explained by the fact that some popular frameworks, such as SenseTime, has first-party or third-party libraries that provide the model encryption feature. However, even for apps using the top-4 ML frameworks, the percentage of ML apps adopting model protection is still low at 59%.

Encrypted Models Reused/Shared among Apps: Our analysis also reveals a common practice used in developing on-device ML apps, which has profound security implications. We found that many encrypted models are reused or shared by different apps. The most widely shared model, namely `SenseID_Motion_Liveness.model`, is found in 81 apps, most of which are unrelated. We confirm that most of these cases are legitimate and resulted from a trend in on-device ML, whereby app developers buy and use ML models and services from third-party providers, such as SenseTime, instead of developing their own ML features. Apps using the same ML functionality provided by the same vendor often contain the same ML model. Moreover, most of these models are encrypted, which reflects professional ML providers' awareness of model theft risks and their efforts to mitigate them.

Table 3: Number of apps using different ML Frameworks with different functionalities.

Functionality	TensorFlow (Google)	*Caffe2/PyTorch (Facebook)	*Parrots (SenseTime)	TFLite (Google)	NCNN (Tencent)	Mace (Xiaomi)	MxNet (Apache)	ULS (Utility Asset Store)	Total
OCR(Optical Character Recognition)	41	186	140	6	37	18	1	11	441
Face Tracking	26	272	216	7	53	6	13	27	620
Speech Recognition	7	32	9	1	11	18	1	9	88
Hand Detection	4	0	0	2	4	0	0	0	10
Handwriting Recognition	8	17	1	0	16	0	0	0	42
Liveness Detection	32	392	349	9	70	7	10	3	872
Face Recognition	17	116	95	6	40	7	10	3	294
Iris Recognition	0	4	0	0	2	0	3	0	9
ID Card Recognition	26	230	147	5	47	18	0	10	483
Bank Card Recognition	11	126	117	2	16	18	0	9	299

Note: 1) One app may use multiple frameworks for different ML functionalities. Therefore, the sum of apps using different functionalities is bigger than the number of total apps. 2) Security critical functionalities are in **bold fonts** and can be used for fraud detection or access control. 3) *Caffe was initially developed by Berkeley, based on which Facebook built Caffe2, which was later merged with PyTorch. The following uses "Caffe" to represent Caffe, Caffe2 and PyTorch.

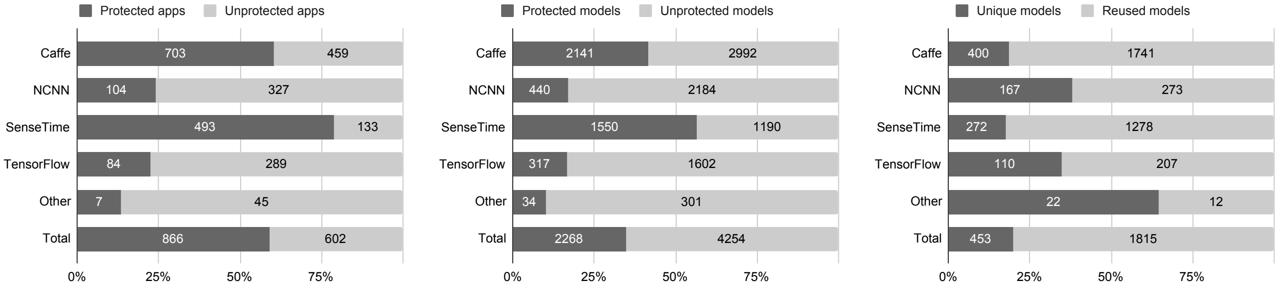


(a) Apps using protected models vs. those using unprotected models

(b) On-device models that are protected/encrypted vs. those not

(c) Unique encrypted models vs. encrypted models reused/shared by multiple apps.

Figure 5: Statistics on ML model protection and reuse, grouped by app markets. The "total" number of unique models is less than the sum of the per-store numbers because some models are not unique from different stores.



(a) Apps using protected models vs. those using unprotected models

(b) On-device models that are protected/encrypted vs. those not

(c) Unique encrypted models vs. encrypted models reused/shared by multiple apps

Figure 6: Statistics on ML model protection and reuse, grouped by ML frameworks. The "total" number is less than the sum of the per-framework numbers because many apps use multiple frameworks for different functionalities.

It is common to see the same encrypted model shared by different apps. For all the encrypted models that we detected from the apps, we calculate their MD5 hashes and identify those models that are used in different and unrelated apps. Figures 5c and 6c show the numbers of unique (or non-shared) models and reused (or shared) models, grouped by app markets and ML frameworks, respectively. Overall, only 22% of all the protected models are unique. 75% of the encrypted models from Google Play are unique whereas only 50% and 19% of the encrypted models on Tencent My App and 360 Mobile Assistant, respectively, are not reused (Figure 5c). When grouped by ML frameworks, 82% of encrypted SenseTime models are shared, the highest among all frameworks (Figure 6c).

The security implication of widely shared encrypted models is two folds. First, the protection of the shared models is not only the responsibility of the SDK providers, but also the responsibility of all the app developers, and the app users. Second, it is challenging to attribute a leaked model to a certain app because multiple apps use the same model.

GPU Acceleration Adoption Rate among ML Apps: Table 4 shows the number ML apps and libraries that use GPU for acceleration. 797(54%) ML apps make use of GPU. *The wide adoption of GPU acceleration poses a challenge to the design* *of secure on-device ML.* For instance, the naive idea of performing model inference and other model access operations entirely inside a trusted execution environment (TEE, *e.g.,* TrustZone) is not viable due to the need for GPU acceleration, which cannot be easily or efficiently accessed within the TEE.

Table 4: ML apps and libraries that use GPU acceleration

	360 Mobile Assistant	Tencent My App	Google Play
ML Apps	669	104	24
ML Libraries	212	103	23

5 Q2: How Robust Are Existing Model Protection Techniques?

To answer this question, we build ModelXtractor, a tool simple by design to dynamically recover protected or encrypted models used in on-device ML. Conceptually, ModelXtractor represents a practical and unsophisticated attack, whereby an attacker installs apps on his or her own mobile device and uses the off-the-shelf app instrumentation tools to identify and export ML models loaded in the memory. ModelXtractor mainly targets on-device ML models that are encrypted during transportation and at rest (in storage) but not protected when in use or loaded in memory. For protected models mentioned in §4, ModelXtractor is performed to assess the robustness of the protection.

The workflow of ModelXtractor is depicted in Figure 7. It takes inputs from ModelXRay, including the information about the ML framework(s) and the model(s) used in the app (described in §4). These information helps to target and efficiently instrument an app during runtime, and capture models in plaintext from the memory of the app. We discuss ModelXtractor's code instrumentation strategies in §5.1, our techniques for recognizing in-memory models in §5.2, and how ModelXtractor verifies captured models in §5.3. Our findings, insights, the answer to Q2, and several case studies are presented in §5.4 and §5.5.

5.1 App Instrumentation

ModelXtractor uses app instrumentation to dynamically find the memory buffers where (decrypted) ML is loaded and accessed by the ML frameworks. For each app, ModelXtractor determines which libraries and functions need to be instrumented and when to start and stop each instrumentation, based on the instrumentation strategies (discussed shortly). ModelXtractor automatically generates the code that needs to be inserted at different instrumentation points. It employs the widely used Android instrumentation tool, Frida [10], to perform code injection.

ModelXtractor has a main instrumentation strategy (S0) and four alternative ones (S1-S4). When the default strategy cannot capture the models, the alternatively strategies (S1-S4) will be used.

S0: Capture at Model Deallocation: This is the default strategy since we observe the most convenient time and place to capture an in-memory model is right before the deallocation of the buffer where the model is loaded. This is because (1) memory deallocation APIs (*e.g.,* `free`) are limited in numbers and easy to instrument, and (2) models are completely loaded and decrypted when their buffers are to be freed.

Naive instrumentation of deallocation APIs can lead to dramatic app slowdown. We optimize it by first only activating it after the ML library is loaded, and second, only for buffers greater than the minimum model size (a configurable threshold). To get buffer size, memory allocation APIs (*e.g.,*`malloc`) are instrumented as well. The size information also helps correlate a decrypted model to its encrypted version (discussed in §5.3).

This default instrumentation strategy may fail in the following uncommon scenarios. First, an app is not using native ML libraries, but a JavaScript ML library. Second, an app uses its own or customized memory allocator/deallocator. Third, a model buffer is not freed during our dynamic analysis.

S1: Capture from Heap: This strategy dumps the entire heap region of an app when a ML functionality is in use, in order to identify possible models in it. It is suitable for apps that do not free model buffers timely or at all. It also helps in cases where memory-managed ML libraries are used (*e.g.,* JavaScript) and buffer memory deallocations (done by a garbage collector) are implicit or delayed.

S2: Capture at Model Loading: This strategy instruments ML framework APIs that load models to buffers. We manually collect a list of such APIs (*e.g., loadModel*) for the ML frameworks observed in our analysis. This strategy is suitable for those apps where S0 fails and the ML framework code is not obfuscated.

S3: Capture at Model Decryption: This strategy instruments model decryption APIs (*e.g., aes256_decrypt*) in ML frameworks, which we we collected manually. Similar to S2, it is not applicable to apps that use obfuscated ML framework code.

S4: Capture at Customized Deallocation: Some apps use customized memory deallocators. We manually identify a few such allocators (*e.g., slab_free*), which are instrumented similarly as S0.

5.2 Model Representation and Recognition

The app instrumentation described earlier captures memory buffers that may contain ML models. The next step is to perform model recognition from the buffers. The recognition is based on the knowledge of in-memory model representations, *i.e.,* different ML frameworks use different formats model encoding, discussed in the following.

Protobuf is the most popular model encoding format, used by TensorFlow, Caffe, NCNN, and SenseTime. To detect and extract models in Protobuf from memory buffers, ModelXtractor uses two kinds of signatures: content signatures and

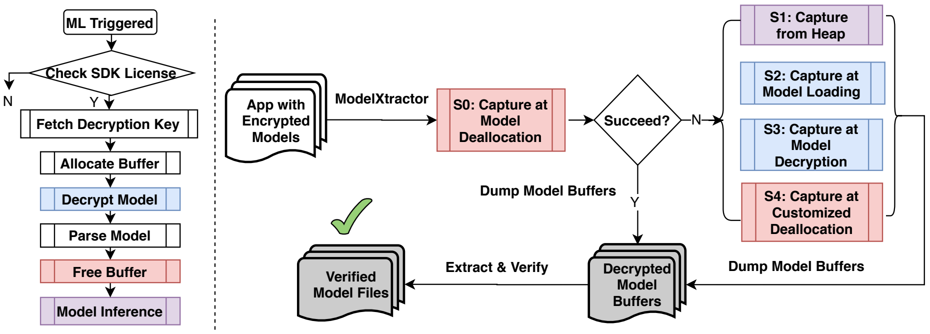


Figure 7: Extraction of (decrypted) models from app memory using ModelXtractor

The left side shows the typical workflow of model loading and decryption in mobile apps. The right side shows the workflow of ModelXtractor. The same color on both sides indicate the same timing of the strategy being used. The "Check SDK License" shows that a model provider will check an app's SDK license before releasing the decryption keys as a way to protect its IP.

encoding signatures. The former is used to identify buffers that contain models and the latter is used to locate the beginning of a model in a buffer.

Model encoded in Protobuf usually contains words descriptive of neural network structures and layers. For example, "conv1" is used for one-dimension convolution layer, and "relu" for the Rectified Linear Unit. Such descriptive words appear in almost every model and are used as the content signatures.

The encoding signatures of Protobuf is derived from its encoding rule [24]. For example, a Protobuf contains multiple *messages*. Every message is a series of key-value pairs, or *fields*. The key of a field is encoded as `(field_number ≪ 3) | wire_type`, where the `field_number` is the ID of the field and `wire_type` specifies the field type.

A typical model in Protobuf starts with a message whose first field defines the model name (*e.g.,*`VGG_CNN_S`). This field usually has a `wire_type` of 2 (*i.e.,* a length-delimited string) and a `field_number` of 0 (*i.e.,* the first field), which means that encoded key for this field is "`0A`". This key is usually the first byte of a Protobuf encoded model. Due to alignment, this key appears at a four-byte aligned address within the buffer. It is used as an encoding signature.

Other model formats and representations have their own content and encoding signature. For example, TFLite models usually include "TFL2" or "TFL3" as version numbers. Some model files are even stored in JSON format, with easily identifiable names for each field. Models from unknown frameworks or of unknown encoding formats are hard to identify from memory. In such cases, we consider the buffer of the same size as the encrypted model to contain the decrypted model. This buffer-model size matching turns out to be fairly reliable in practice. The reason is that, when implementing a decryption routine, programmers almost always allocate a buffer for holding the decrypted content with the same size as the encrypted content. This practice is both convenient (i.e., no need to precisely calculate the buffer size before decryption) and safe (i.e., decrypted content is always shorter than its encrypted counterpart due to the use of IV and padding during encryption). We show how buffer size matching is used in our case studies in §5.5.

5.3 Evaluation of ModelXtractor

Model Verification: ModelXtractor performs a two-step verification to remove falsely extracted models. First, it confirms that the extracted model is valid. Second, it verifies that the extracted model matches the encrypted model. We use publicly available model parsers to verify the validity of extracted model buffers (*e.g.,* protobuf decoder [21] to extract protobuf content, and Netron [19] to show the model structure). When a decoding or parsing error happens, ModelXtractor considers the extracted model invalid and reports a failed model extraction attempt. To confirm that an extracted model indeed corresponds to the encrypted model, ModelXtractor uses the buffer-model size matching described before.

Evaluation on Apps from Google Play: There are 47 ML apps from Google Play that use encryption to protect their models. We applied ModelXtractor on half of the ML apps (randomly selected 23 out of 47). Among the tested 23 apps, we successfully extracted decrypted models from 9 of them. As for the other 14 apps, 2 apps do not use encryption, 1 app does not using ML, and 11 apps do not have their models extracted for the following reasons: apps cannot be instrumented; apps did not trigger the ML function; apps cannot be installed on our test devices.

Evaluation on Apps from Chinese App Markets: There are 819 apps from Chinese app markets found to be using encrypted models, where model reuse is quite common as shown in our static analysis. We carefully selected 59 of these apps prioritizing model popularity and app diversity. Our analyzed apps cover 15 of the top 45 most widely used models (*i.e.,* each is reused more than 10 times) and 8 app categories.

When analyzing the Chinese apps, we encountered some non-technical difficulties of navigating the apps and triggering their ML functionalities. For instance, some apps require phone numbers from certain regions that we could not obtain for user registration. A lot of them are online P2P loan apps or banking apps that require a local bank account to trigger ML functionalities. Out of the 59 apps, we managed to successfully navigate and trigger ML functionalities in 16 apps. We then extracted decrypted models from 9 of them.

Limitation of ModelXtractor: ModelXtractor failed to extract 11 models whose ML functionalities were indeed triggered. This was because of the limitation of our instrumentation strategies discussed in §5.1. We note that these strategies and the design of ModelXtractor are not meant to extract every protected model. Instead, they represent a fairly practical and simple attack, designed only to reveal the insufficient protection of ML models in today's mobile apps.

5.4 Findings and Insights

Results of Dynamic Model Extraction: Table 5 shows the statistics on the 82 analyzed apps, grouped by the ML frameworks they use. Among the 29 apps whose ML functionalities were triggered, we successfully extracted models from 18 of them (66%). Considering the reuse of those extracted encrypted models, the number of apps that are affected by our model extraction is 347 (*i.e.,* 347 apps used the same models and same protection techniques as the 18 apps that we extracted models from). This extraction rate is alarming and shows that a majority of the apps using model protection can still lose their valuable models to an unsophisticated attack. It indicates that *even for app developers and ML providers willing/trying to protect their models, it is hard to do it in a robust way using the file encryption-based techniques*.

Table 6 shows the per-app details about the extracted models. We anonymized the apps for security concerns: many of them are highly downloaded apps or provide security-critical services. Many of the listed apps contain more than one ML models. For simplicity, we only list one representative model for each app.

Most decrypted models in memory are not protected at all. As shown in Table 6, most of the decrypted models (12 of 15) were easily captured using the default strategy (S0) when model buffers are to be freed. This means that the decrypted models may remain in memory for an extended period of time (*i.e.,* decrypted models are not erased before memory deallocation), which creates a large time window for model thefts for leakages. Moreover, this result indicates that apps using encryption to protect models are not doing enough to secure decrypted models loaded in memory, partly due to the lack practical in-memory data protection techniques on mobile platforms.

Popularity and Diversity of Extracted Models: *The extracted models are highly popular and diverse, some very valuable or security-critical.* From Table 6 we can see that 8 of 15 listed apps have been downloaded more than 10 million times. Half of the extracted models belong to commercial ML providers, such as SenseTime, and were purchased by the app developers. Such models being leaked may cause direct financial loss to both app developers and model owners (§6).

As for diversity, the model size ranges from 160KB to 20MB. They span all the popular frameworks, such as TensorFlow, TFLite, Caffe, SenseTime, Baidu, and Face++. The observed model formats include Protobuf, FlatBuffer, JSON, and some proprietary formats used by SenseTime, Face++ and Baidu. In terms of ML functionalities, the models are used for face recognition, face tracking, liveness detection, OCR, ID/card recognition, photo processing, and malware detection. Among them, liveness detection, malware detection, and face recognition are often used for security-critical purposes, such as access control and fraud detection. Leakage of these models may give attackers an advantage to develop model evasion techniques in a white-box fashion.

Potential Risk of Leaking SDK/Model License: *SDK/Model license are poorly protected.* Developers who bought the ML SDK license from model provider usually ship the license along with app package. During analysis, we find the license are used to verify legal use of SDK before model file get decrypted. However, license file are not protected by the developer, which means it is possible to illegally use the SDK by stealing license file directly from those apps that have bought it. Poor protection of license has been observed in both SenseTime ML SDKs and some other SDKs, which actually affects hundreds of different apps.

Table 5: Model extraction statistics.

ML Framework	Unique Models Analyzed	ML Triggered	Models Extracted	Models Missed	Apps Affected
TensorFlow	3	3	3	0	3
Caffe	7	3	1	2	79
SenseTime	55	16	11	5	186
TFLite	3	2	2	0	76
NCNN	9	3	0	3	0
Other	5	3	2	1	88
Total	82	29	18	11	347

Note: 347 is the sum of affected apps per framework after deduplication.

5.5 Interesting Cases of Model Protection

We observe a few cases clearly showing that some model providers use extra protection on their models. Below we discuss these cases and share our insights.

Encrypting Both Code and Model Files: Anyline is a popular OCR SDK provider. It licenses its SDK and models to apps that need OCR functionalities. We analyzed an app that use Anyline. From the app profile generated by ModelXRay, we can tell that the app uses TensorFlow framework with the Anyline models. It places the encrypted models under a directory named "encrypted_models". Initially, ModelXtractor failed to extract the decrypted models using the default strategy (S0). We manually investigated the reason and found that, unlike most ML apps, this app runs ML inference in a customized WebView, where an encrypted JavaScript, dynamically loaded at runtime, performs the model decryption and inference.

Table 6: Overview of Successfully Dumped Models with ModelXtractor

App name	Downloads	Framework	Model Functionality	Size (B)	Format	Reuses	Extraction Strategy
Anonymous App 1	300M	TFLite	Liveness Detection	160K	FlatBuffer	18	Freed Buffer
Anonymous App 2	10M	Caffe	Face Tracking	1.5M	Protobuf	4	Model Loading
Anonymous App 3	27M	SenseTime	Face Tracking	2.3M	Protobuf	77	Freed Buffer
Anonymous App 4	100K	SenseTime	Face Filter	3.6M	Protobuf	3	Freed Buffer
Anonymous App 5	100M	SenseTime	Face Filter	1.4M	Protobuf	2	Freed Buffer
Anonymous App 6	10K	TensorFlow	OCR	892K	Protobuf	2	Memory Dumping
Anonymous App 7	10M	TensorFlow	Photo Process	6.5M	Protobuf	1	Freed Buffer
Anonymous App 8	10K	SenseTime	Face Track	1.2M	Protobuf	5	Freed Buffer
Anonymous App 9	5.8M	Caffe	Face Detect	60K	Protobuf	77	Freed Buffer
Anonymous App 10	10M	Face++	Liveness	468K	Unknown	17	Freed Buffer
Anonymous App 11	100M	SenseTime	Face Detect	1.7M	Protobuf	18	Freed Buffer
Anonymous App 12	492K	Baidu	Face Tracking	2.7M	Unknown	26	Freed Buffer
Anonymous App 13	250K	SenseTime	ID card	1.3M	Unknown	13	Freed Buffer
Anonymous App 14	100M	TFLite	Camera Filter	228K	Json	1	Freed Buffer
Anonymous App 15	5K	TensorFlow	Malware Classification	20M	Protobuf	1	Decryption Buffer

Note: 1) We excluded some apps that dumped the same models as reported above; 2) We anonymized the name of the apps to protect the user's security; 3) Every app has several models for different functionalities, we only list one representative model for each app.

We analyzed the heap memory dumped by ModelXtractor using the alternative strategy, S1. We searched for potential model buffers and found the TensorFlow model buffers in the memory dump. We verified our findings by decoding the Protobuf model buffers and extract the models' weights.

This interesting case shows that, despite the extra protection and the sophisticated obfuscation, the app can still lose its models to not-so-advanced attacks that can locate and extract decrypted models in app memory.

Encrypting Feature Vectors and Formats: We encountered one malware detection app, which does not encrypt its model file, but instead encrypts the feature vectors. This app uses a Random Forest model for malware classification. It uses TensorFlow and the model is in the Protobuf format. There are more than one thousand features used in this malware classification model, like the APIs used by the App or the Permissions claimed in the Android Manifest files. By encrypting the feature vectors, its developer assumes it is impossible to (re)use the model because the input format and content are unknown to attackers. However, we instrumented the decryption functions and extracted decrypted feature vectors. With this information, an attacker can steal and recover the model as well as the feature vector format, which can lead to model evasions or bypassing the malware detection.

Encrypting Models Multiple Times: Another type of extra model protection we observed was in an app that encrypts its models multiple times. This app is a financial app that offers online P2P loans. It uses two models provided by SenseTime: one for ID card recognition and one for liveness detection to fraud, which are security critical. ModelXtractor captured 6 model buffers successfully, whose sizes range from 200KB to 800KB. However, we only found 2 encrypted model files. When we were trying to map the dumped model buffers to the encrypted files, we found something very interesting. One encrypted model file named *SenseID_Ocr_Idcard_Mobile_1.0.1.model* has a size of 1.3 MB. Among the dumped model buffers, we have one buffer of the same size. It is supposed to be the decrypted buffer of the encrypted file. After analyzing its content, we found that it is actually a tar file containing multiple files, one of which is *align_back.model*. After inspecting the content of *align_back.model*, we found that it is also an encrypted file. We then found another buffer of exactly the same size, 246 KB, which contains a decrypted model. We finally realized that the app encrypts each model individually and then compresses all encrypted models into a tar file, which is encrypted again.

6 Q3: How Much Can (Stolen) Models Cost?

ML models are the core intellectual properties of ML solution providers. The impacts of leaked models are wide and profound. First, designing and training models usually requires a large amount computing and human resources. Second, models leaked to competitors can weaken model owners' competitive edge and adversely affect their pricing strategy. Third, leaking critical models can result in security and privacy breaches. With access to private models, attackers can create adversarial inputs, for example, to bypass the model used in face/fingerprint/iris recognition. All these impacts of leaked models can translate to financial loss.

In this section, we qualitatively estimate the potential financial loss caused by leaked ML models. There are two common sources of models. Type-I: app developers buy an ML SDK and model license from an ML solution provider, such as SenseTime, Face++, and so on. Type-II: app developers design and train their own ML models using open-source or customized frameworks. The following discussion is focused on Type-I models because it is more popular among commercial models and the analysis for Type-II can be derived from Type-I.

6.1 Breakdown of Financial Impact

Research and Development Cost: The R&D cost of ML models comes from three sources: collecting and labeling data for training, hiring AI engineers for designing and fine-tuning models, and computing resources, such as renting Amazon SageMaker or buying and maintaining GPU clusters for training models. With simplification, the R&D cost can be formulated as follows:

$$Cost_{R\&D} = Cost_{Data} + Cost_{HR} + Cost_{Machine} \tag{1}$$

Data labelling can be outsourced and its pricing can be estimated base on public data online. The data collection cost varies case by case, depending on the nature of the data and source. $Cost_{HR}$ can be estimated based on the annual salary paid to AI engineers, multiplied by the duration of model development. Similarly, $Cost_{Machine}$ can be estimated based on the rates of renting storage servers and GPU clusters.

Financial Loss to Competition: The impact of models leaked to competitors can be reflected in potential revenue loss caused by the reduction of customers, market share, or product price due to lost competitive edge. All stakeholders, including ML solution providers and app developers (model license subscribers), are affected. We estimate the competition loss with the following formula:

$$Cost_{Compete} = \sum (R_{StakeholderX} \times H_{CompeteX}) \tag{2}$$

where $H_{CompeteX}$ is the competition impact factor for Stakeholder X, ranging from 0 to 1. $R_{StakeholderX}$ is the revenue of stakeholder X. Intuitively, $Cost_{Compete}$ is the revenue lost to all competitors due to a leaked model. For an SDK model provider, its revenue can be considered as the accumulation of license fee for each customer.

Financial Loss to Model Evasion: On-device ML has been used widely in access control, *e.g.,* face, fingerprint, and iris recognition, which are security critical. The leakage of such critical models can greatly facilitate attackers to craft adversarial inputs to pass the security checks. We estimate the financial loss from model evasion using the following formula:

$$Cost_{Security} = \sum (R_{StakeHolderX} \times H_{SecurityX}) \tag{3}$$

where $H_{SecurityX}$ means the security impact factor on stakeholder X, ranging from 0 to 1.

6.2 Case Study of Financial Loss Estimation

We choose the following apps for this study (anonymized for security reasons): Translate App A, Bank App B, Bank App C, and E-Commerce App D [18]. These apps are chosen for their diversity in model complexity, functionality, popularity and model providers. The popularity of model (reuse rate) ranges from 1 to 68, and the models are provided by different providers like Baidu, Face++, and SenseTime. The model size ranges from 508KB to 3.1MB, they cover functionalities such as OCR, liveness, bank card detection, etc.

Table 7 shows the estimated financial loss of model leakage from the four apps mentioned above. We estimated the amount of resources, *e.g.,* engineering team size and number of servers, based on the five-point Likert scale [43], and the impact factors, *e.g.,* $H_{CompeteX}$, $H_{SecurityX}$, based on the three-point Likert scale. The scale categorization details can be found in the Appendix at Table B1. We use the public pricing of Amazon SageMaker [16] to calculate the cost for data collection, labeling, GPU, and storage. We use Paysa [14] for estimating AI engineer salaries and use Crunchbase [9] for collecting revenue and competition information. We give higher security impact factor when the model is security critical, e.g. liveness detection.

Financial Impact on Model Providers: For model providers, their revenue comes from licensing their models to app developers. They are the one who invest on developing the models. The financial impact on them is largely affected by their R&D cost and their revenue loss due to possible customer loss after model leakage.

Leakage of popular and security critical models may cause much more financial loss to model providers. From Table 7, we can see that the financial loss can be as high as $1,246,000 for SenseTime in the case of E-Commerce App D when a popular model used by 68 apps is stolen. The lowest loss is $464,000, still fairly high, in the case of Translate App A, whose model is only used by one app.

Financial Impact on App Developers: For developers, they do not have direct R&D cost in developing ML models. Their loss comes mainly from the license fee they paid to the model provider. The financial loss to competition comes when their competitors use a leaked model at a lower or zero price. This loss can be bounded and is also reflected in the license fee. So we do not consider an extra financial loss to competition in addition to the license fee for app developer. If the model is security critical, their revenue are further affected due to user's security concern; otherwise, we do not consider the financial loss to security.

Machine learning used in security critical case is a double-edged sword to app developer. From Table 7, we can see that the financial loss can be as high as $18,100,000 in the case of E-Commerce App D, when the model is security critical and the company has high revenue. The model here is used to verify whether it is a real human holding a real ID card. It is a good example showing that when ML is used for security-critical purposes, they can bring in great convenience and security risks at the same time. For example, users do not need to go to customer service centers to verify their identities; however, if the ML models are leaked, the security check may be easily bypassed.

7 Discussion

Research Insights: *White-box Adversarial Machine Learning.* Previous research on adversarial machine learning has been focused on black-box threat models, assuming the model files are inaccessible. Our research shows that an attacker can easily extract the protected private models. As a result, more research

Table 7: Financial Loss Estimation Examples

App Name	Model Size (B)	Reuse Times	SDK Information	Functionality	Model Provider			App Developer			Total
					R&D	Competition	Evasion	License	Competition	Evasion	
Translate App A	2.7M	1	Baidu	OCR	$454,000	$10,000	-	$100,000	-	-	$564,000
Bank App B	508K	2	Face++	Liveness	$228,000	$20,000	$40,000	$100,000	-	$ 3,600,000	$3,988,000
Bank App C	3.1M	15	SenseTime	Bank card	$424,000	$75,000	-	$100,000	-	-	$599,000
E-Commerce App D	644K	68	SenseTime	Liveness	$226,000	$340,000	$680,000	$100,000	-	$18,000,000	$19,346,000

Note: We anonymized the real name of the apps, only show their categories.

on defending adversarial machine learning under white-box threat model is much needed to improve the resiliency of those models used in security critical applications.

Model Plagiarism Detection. As machine learning models are not well protected, attackers, instead of training their own model, can steal their competitor's model and reuse it. As a result, model plagiarism detection is needed to prevent this type of attack. It is challenging because the attacker can retrain their model based on the stolen one, making it looks very different. We need research to detect model plagiarism and provide forensic tools for illegal model reuse analysis.

Limitations: Since the goal of this paper is to show that even simple tools can extract on-device ML models in a large scale, ModelXRay and ModelXtractor are limited by the straightforward design of keyword matching. We acknowledge that the scale of model extraction can be further improved by leveraging program slicing and partial execution [34,44], and Android app de-obfuscation [32,53]. Further, model encoding and content features are limited to well-known ML SDKs having documentation available, thereby we believe an extended knowledge base can further include special model encoding formats.

We note that our financial loss analysis is subjective and limited by the asymmetric information of R&D cost and company revenue. The approach is used to emphasize the point that costs can be very high. A more comprehensive study can be carried out by stakeholders having real data of model leakage cases.

8 Related Work

Motivated by hardware acceleration and efficiency improvement of deep neural networks [40], on-device model inference becomes a new trend [54]. This work empirically evaluates model security on mobile devices. It interacts with three lines of research: machine learning model extraction, adversarial machine learning, and proprietary model protection.

Prior work on machine learning model extraction focuses on learning-based techniques targeting ML-as-a-service. Tramer et. al proposes stealing machine learning models via prediction APIs [50], since ML-as-a-service may accept partial feature vectors as inputs and include confidence values with predictions. Then, Wang et. al [52] extend the attacks by stealing hyperparameters. Other work includes stealing the functionality of the models [36,42], querying the gradient to reconstruct the models [41], exploratory attacks to reverse engineer the classifiers [45], and side channel attacks to recover the models [31]. Our work is orthogonal to these study by targeting on-device model inference, assuming the attackers having physical access to the mobile devices running model inference.

Model extraction paves the road for adversarial machine learning. Prior work [35,39] fooling the models or bypassing the check is mostly under the black-box threat model. Once ML models become white-box, attackers can easily craft adversarial examples to deceive the learning systems. Our study shows white-box adversarial machine learning is a real threat to on-device ML models.

To secure machine learning training and inference, research effort has been spent to prevent model information leakage and unauthorized uses. Graviton [51] is proposed to make GPU a trusted execution environment with minimal hardware changes incurred. Slalom [49] uses hardware enclaves, e.g. SGX, during model inference and applies homomorphic encryption to outsource the computation of linear layers to GPU securely. Privado [48] uses SGX to mitigate side channel attacks of input inference. TensorScone [38] also uses SGX to protect model inference but does not include GPU. So far, research in this area focuses on cloud end security. Our work shows that model security on mobile device is worrisome and research on hardware backed model protection for mobile devices is in need. To protect the model as an intellectual property, watermark technique has been used to detect illegitimate model uses [30,55]. Moreover, fingerprinting has been used to protect model integrity. Chen et al. encodes fingerprint [33] in DNN weights so that the models can be attested to make sure it is not tampered or modified. Our research supports it with the finding that model plagiarism is a realistic problem especially for mobile platforms.

9 Conclusion

We carry out a large scale security analysis of machine learning model protection on 46,753 Android apps from both the Chinese and the US app markets. Our analysis shows that on-device machine learning is gaining popularity in every category of mobile apps, however, 41% of them are not protecting their models. For those are, many suffer from weak protection mechanisms, such as using the same encrypted model for multiple apps, and even the encrypted models can be easily recovered with our unsophisticated analysis. Our financial loss analysis shows that model leakage can cost as high as millions of dollars, indicating that attackers both technically can and financially are motivated to steal models. We call for research into robust model protection.

References

[1] A brief guide to mobile AI chips. https://www.theverge.com/2017/10/19/16502538/mobile-ai-chips-apple-google-huawei-qualcomm.

[2] Amazon SageMaker Ground Truth pricing. https://aws.amazon.com/sagemaker/groundtruth/pricing/.

[3] Apache MXNet | A flexible and efficient library for deep learning. https://mxnet.apache.org/.

[4] Boston Consulting Group's Analysis on AI and Banking Operation. http://image-src.bcg.com/Images/BCG_Banking_Operations_Oct%202017_CHN_tcm55-173580.pdf.

[5] Caffe2 -a lightweight, modular, and scalable deep learning framework. https://github.com/facebookarchive/caffe2.

[6] Case Study: How To Build A High Performance Data Science Team. https://www.business-science.io/business/2018/09/18/data-science-team.html.

[7] Converting model to C++ code. https://mace.readthedocs.io/en/latest/user_guide/advanced_usage.html.

[8] Core ML | Apple Developer Documentation. https://developer.apple.com/documentation/coreml.

[9] Crunchbase Pro | Industry Leading Company Data. https://www.crunchbase.com/.

[10] Dynamic instrumentation toolkit for developers, reverse-engineers, and security researchers. https://frida.re/.

[11] Entropy(information theory). https://en.wikipedia.org/wiki/Entropy_(information_theory)#Entropy_as_information_content.

[12] Equifax Data Breach Settlement. https://www.ftc.gov/enforcement/cases-proceedings/refunds/equifax-data-breach-settlement.

[13] Face++ - Cognitive Services. https://www.faceplusplus.com/.

[14] Google Machine Learning Engineer Salaries. https://www.paysa.com/salaries/google--machine-learning-engineer.

[15] Kairos:60 Facial Recognition Databases. https://www.kairos.com/blog/60-facial-recognition-databases.

[16] Machine Learning with Amazon SageMaker - Amazon SageMaker. https://docs.aws.amazon.com/en_pv/sagemaker/latest/dg/how-it-works-mlconcepts.html.

[17] Market Share—a Key to Profitability. https://hbr.org/1975/01/market-share-a-key-to-profitability.

[18] NetEase Kaola ranked China's No.1 cross-border import retail e-commerce platform for sixth consecutive time. https://www.prnewswire.com/news-releases/netease-kaola-ranked-chinas-no1-cross-border-import-retail-e-comm
html.

[19] Netron. https://lutzroeder.github.io/netron/.

[20] OCR & Handwriting Datasets for Machine Learning . https://lionbridge.ai/datasets/15-best-ocr-handwriting-datasets/.

[21] Online Protobuf Decoder. https://protogen.marcgravell.com/decode.

[22] Over 1.5 TB's of Labeled Audio Datasets. https://towardsdatascience.com/a-data-lakes-worth-of-audio-datasets-b45b88cd4ad.

[23] PCI Compliance Guide. https://www.pcicomplianceguide.org/faq/.

[24] Protocol Buffers Encoding Rule. https://developers.google.com/protocol-buffers/docs/encoding#simple.

[25] Strip visible string in ncnn. https://github.com/Tencent/ncnn/wiki.

[26] TensorFlow. https://www.tensorflow.org/.

[27] Unity Asset Store - The Best Assets for Game Making. https://assetstore.unity.com/?category=tools%2Fai&orderBy=1.

[28] SenseTime. https://www.sensetime.com/, 2019.

[29] The AppInChina App Store Index. https://www.appinchina.co/market/app-stores/, 2019.

[30] 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 ({USENIX} Security 18)*, pages 1615–1631, 2018.

[31] Lejla Batina, Shivam Bhasin, Dirmanto Jap, and Stjepan Picek. Csi neural network: Using side-channels to recover your artificial neural network information. *arXiv preprint arXiv:1810.09076*, 2018.

[32] Benjamin Bichsel, Veselin Raychev, Petar Tsankov, and Martin Vechev. Statistical deobfuscation of android applications. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 343–355, 2016.

[33] Huili Chen, Cheng Fu, Bita Darvish Rouhani, Jishen Zhao, and Farinaz Koushanfar. DeepAttest: An End-to-End Attestation Framework for Deep Neural Networks. 2019.

[34] Yi Chen, Wei You, Yeonjoon Lee, Kai Chen, XiaoFeng Wang, and Wei Zou. Mass discovery of android traffic imprints through instantiated partial execution. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 815–828, 2017.

[35] Ling Huang, Anthony D Joseph, Blaine Nelson, Benjamin IP Rubinstein, and J Doug Tygar. Adversarial machine learning. In *Proceedings of the 4th ACM workshop on Security and artificial intelligence*, pages 43–58. ACM, 2011.

[36] Matthew Jagielski, Nicholas Carlini, David Berthelot, Alex Kurakin, and Nicolas Papernot. High-fidelity extraction of neural network models. *arXiv preprint arXiv:1909.01838*, 2019.

[37] Andrej Karpathy, George Toderici, Sanketh Shetty, Thomas Leung, Rahul Sukthankar, and Li Fei-Fei. Large-scale video classification with convolutional neural networks. In *Proceedings of the IEEE conference on Computer Vision and Pattern Recognition*, pages 1725–1732, 2014.

[38] Roland Kunkel, Do Le Quoc, Franz Gregor, Sergei Arnautov, Pramod Bhatotia, and Christof Fetzer. TensorSCONE: A Secure TensorFlow Framework using Intel SGX. *arXiv preprint arXiv:1902.04413*, 2019.

[39] Alexey Kurakin, Ian Goodfellow, and Samy Bengio. Adversarial machine learning at scale. *arXiv preprint arXiv:1611.01236*, 2016.

[40] Juhyun Lee, Nikolay Chirkov, Ekaterina Ignasheva, Yury Pisarchyk, Mogan Shieh, Fabio Riccardi, Raman Sarokin, Andrei Kulik, and Matthias Grundmann. On-Device Neural Net Inference with Mobile GPUs. https://arxiv.org/abs/1907.01989, 2019.

[41] Smitha Milli, Ludwig Schmidt, Anca D Dragan, and Moritz Hardt. Model reconstruction from model explanations. *arXiv preprint arXiv:1807.05185*, 2018.

[42] Tribhuvanesh Orekondy, Bernt Schiele, and Mario Fritz. Knockoff nets: Stealing functionality of black-box models. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 4954–4963, 2019.

[43] Sujatha Perera, Graeme Harrison, and M Poole. Customer-focused manufacturing strategy and the use of operations-based non-financial performance measures: a research note. *Accounting, Organizations and Society*, 22(6):557–572, 1997.

[44] Siegfried Rasthofer, Steven Arzt, Marc Miltenberger, and Eric Bodden. Harvesting runtime values in android applications that feature anti-analysis techniques. In *NDSS*, 2016.

[45] Tegjyot Singh Sethi and Mehmed Kantardzic. Data driven exploratory attacks on black box classifiers in adversarial domains. *Neurocomputing*, 289:129–143, 2018.

[46] Mark Slee, Aditya Agarwal, and Marc Kwiatkowski. Thrift: Scalable Cross-Language Services Implementation. Technical report.

[47] Richard Stiennon. Categorizing Data Breach Severity with a Breach Level Index. Technical report.

[48] Shruti Tople, Karan Grover, Shweta Shinde, Ranjita Bhagwan, and Ramachandran Ramjee. Privado: Practical and secure DNN inference. *arXiv preprint arXiv:1810.00602*, 2018.

[49] Florian Tramer and Dan Boneh. Slalom: Fast, verifiable and private execution of neural networks in trusted hardware. *arXiv preprint arXiv:1806.03287*, 2018.

[50] Florian Tramèr, Fan Zhang, Ari Juels, Michael K Reiter, and Thomas Ristenpart. Stealing machine learning models via prediction apis. In *25th {USENIX} Security Symposium ({USENIX} Security 16)*, pages 601–618, 2016.

[51] Stavros Volos, Kapil Vaswani, and Rodrigo Bruno. Graviton: Trusted execution environments on GPUs. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, pages 681–696, 2018.

[52] Binghui Wang and Neil Zhenqiang Gong. Stealing hyperparameters in machine learning. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 36–52. IEEE, 2018.

[53] Michelle Y Wong and David Lie. Tackling runtime-based obfuscation in android with {TIRO}. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*, pages 1247–1262, 2018.

[54] Mengwei Xu, Jiawei Liu, Yuanqiang Liu, Felix Xiaozhu Lin, Yunxin Liu, and Xuanzhe Liu. A First Look at Deep Learning Apps on Smartphones. *The World Wide Web Conference on - WWW '19*, (May):2125–2136, 2019.

[55] 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. ACM, 2018.

Appendix A Keywords for Different ML Frameworks

Table A1: ML Framework Keywords

Framework	Magic Words	Framework	Magic Words
TensorFlow	tensorflow	Caffe	caffe
MXnet	mxnet	NCNN	ncnn
Mace	libmace, mace_input	SenseTime	sensetime, st_mobile
ULS	ulstracker, ulsface	Other	neuralnetwork, lstm, cnn, rnn

Note: "TensorFlow Lite" and "TensorFlow" are merged into one framework, we handle "TensorFlow Lite" separately with magic word "tflite" in the "TensorFlow" category.

Appendix B Financial Loss Estimation Details

We calculate the loss factors based on a set of model metadata, listed in Table B1. The size of storage and number of labels depend on the type of input dataset of the model. We categorize the types to text, image, audio, and video. The input dataset size and number of labels within each category are estimated base on the mean of well researched public datasets [15, 20, 22, 37]. The number of engineers and GPUs needed are estimated based on our model development experience, and the team structure suggested by Business Science [6]. We consider model sizes of small, medium-small, medium, medium-large, and large, as less than 1MB, 1MB to 5MB, 5MB to 10MB, 10MB to 20MB, and bigger than 20MB, respectively. These numbers are chosen based on our observation that on-device model file size usually ranges from 4KB to 25MB. For the percentage of company's revenue that will be affected by the application of AI, we assume 1% will be affected based on BCG's estimation on banking apps [4]. $H_{competition}$ is in range [0,1] and increases with the number of competitors. We chose low, medium, and high to represent $H_{competition}$, and the corresponding numbers 1%, 5% and 10% are chosen to represent the decrease of market share because of competition [17]. Given that one company usually has fewer than 10 competitors [9], low, medium, and high are categorized as the numbers of competitors fewer than 2, 3 to 5, and more than 5, respectively. $H_{security}$ is represented by minor, moderate, and critical [23, 47], and the corresponding numbers are chosen based on the fee of Equifax data breach settlement (10% revenue) [12]. We consider model with functionalities such as face detect, hand detect, OCR, bank card and ID card recognition, handwriting recognition as minor, liveness detection, IRIS and face recognition as critical.

To simplify the calculation, we chose the following setting for our estimation: $186,000 yearly salary per engineer [14], $0.04 per label [2], $10,000 per GPU and $500 per storage machine for one year rental.

Table B1: Model Metadata and Financial Loss Estimation

Metadata	Loss factor	Description of loss value estimation
Data type	data storage	Text: 984MB, Image: 160GB, Audio: 280GB, Video: 1TB
	# of labels	Text: 800,000, Image: 750,000, Audio: 1,000,000, Video: 1,000,000
Model size	# of engineers	Small: 1, Medium-small: 2, Medium: 3, Medium-large: 4, Large: 5
	# of GPUs	Small: 1, Medium-small: 2, Medium: 3, Medium-large: 4, Large: 5
App name	Revenue	Query Crunchbase with app name
	$H_{competition}$	Low: 1%, Medium: 5%, High: 10%
SDK information	License fee	Query SDK provider website
	Reuse times	From our calculation
	$H_{competition}$	Low: 1%, Medium: 5%, High: 10%
Functionality	$H_{security}$	Minor: 1%, Moderate: 5%, Critical: 10%