Understanding and Defending against the Security Threats on Mobile and IoT Devices with Hardware-backed Trusted Computing Primitives

PhD Thesis Proposal

Zhichuang Sun

Northeastern University
Khoury College of Computer Sciences

January 7, 2021

Abstract

Mobile devices are becoming an essential part of our daily life and IoT devices are also being rapidly deployed in our home, factories and infrastructures. These omnipresent devices raise widely security and privacy concerns while bringing in great convenience. For example, the attacks on IoT devices are evolving and becoming more and more complex and destructive. The newly emerging technologies deployed on mobile devices like AI, also opens new attack surface for the attackers, *e.g.,* the privacy of machine learning models on mobile devices.

In this thesis, I present two novel systems and one large-scale study to understand and combat the newly emerging attacks on both IoT devices and mobile devices. The security of these two systems are built on hardware-backed trusted computing primitives to provide strong security guarantee. First, I present OAT, an attestation system that captures the integrity of both control flow and critical data involved in an operation execution to detect advanced attacks on IoT devices. Second, I present a large-scale study of insufficient model protection in mobile apps based on 46,753 trending Android apps collected from the US and Chinese app markets. Alarmingly, this study shows that 41% of ML apps do not protect their models at all. For those apps that use model protection or encryption, we are able to extract the models from 66% of them via unsophisticated dynamic analysis techniques. Third, to protect on-device ML models, I propose ShadowNet, a secure and efficient model inference system built for mobile devices. The security of ShadowNet is rooted in the TEE, which can protect the model privacy even when the OS is compromised. ShadowNet exploits a novel idea to outsource the heavy part of the model inference to the untrusted world (including GPU) for acceleration without leaking the model weights.

Contents

1 Introduction 3
- 1.1 Problem Statement . . . 3
- 1.2 Thesis Statement . . . 3
- 1.3 Approaches Overview . . . 4

2 Background and Related Work 4
- 2.1 Attacks on IoT Devices and Backends . . . 4
- 2.2 ARM TrustZone . . . 5
- 2.3 Remote Attestation . . . 5
- 2.4 On-device Machine Learning . . . 5
- 2.5 Secure Machine Learning . . . 6

3 Attesting Operation Integrity of IoT Devices 6
- 3.1 Example: A Vulnerable Robotic Arm . . . 7
- 3.2 Methodology . . . 8
  - 3.2.1 Operation Execution Integrity (OEI) . . . 8
  - 3.2.2 OAT System . . . 9
- 3.3 Evaluation . . . 10

4 A Large-scale Study on Insufficient Machine Learning Model Protection in Mobile Apps 12
- 4.1 Analysis Overview . . . 12
- 4.2 Q1: How Widely Is Model Protection Used in Apps? . . . 13
  - 4.2.1 Methodology . . . 13
  - 4.2.2 Findings and Insights . . . 14
- 4.3 Q2: How Robust Are Existing Model Protection Techniques? . . . 17
  - 4.3.1 Methodology . . . 17
  - 4.3.2 Findings and Insights . . . 18
- 4.4 Q3: What Impacts can (Stolen) Models Incur? . . . 19
  - 4.4.1 Financial Impact . . . 19
  - 4.4.2 Security Impact . . . 20

5 Research Plan 21
- 5.1 ShadowNet: a Secure and Efficient Model Inference Scheme . . . 21
- 5.2 Formal Security Analysis of the Scheme . . . 22
- 5.3 System Design Challenges . . . 22
- 5.4 Milestones . . . 22

1 Introduction

1.1 Problem Statement

Mobile and IoT devices devices are omnipresent. Our shopping, social networking, entertainment, work, and payment are made much more convenient with mobile phones. IoT devices are also being rapidly deployed in home automation, industrial automation, smart cities, agriculture and so on, making our home more smart and our industry more efficient. However, with great convenience also comes great concerns about the security and privacy of our digital assets. First, the attacks on IoT devices are evolving and become more and more complex and destructive. Second, with the newly emerging technologies like AI being deployed on mobile devices, new attack surface are being opened up for the attackers, *e.g.,* the privacy of machine learning models on our mobile phones.

Advanced attacks on IoT devices: Internet-of-Things (IoT) and Cyber-Physical Systems (CPS) are being rapidly deployed in smart homes, automated factories, intelligent cities, and more. As a result, embedded devices, playing the central roles as sensors, actuators, or edge-computing nodes in IoT systems, are becoming attractive targets for cyber attacks. Unlike computers, attacks on embedded devices can cause not only software failures or data breaches but also physical damage. Moreover, a compromised device can trick or manipulate the IoT backend (e.g., remote controllers or in-cloud services): hijacking operations and forging data.

The early attacks on IoT devices usually abuse the misconfiguration of IoT devices, like default passwords [81], which can be mitigated through education on the users and security-aware default configuration. Nowadays, more powerful attacks that exploit the implementation bug of wireless protocol have been demonstrated practical and destructive [71]. Return-oriented programming (ROP) and data-only attacks are also proven to be easy to launch on embedded devices, as demonstrated on vulnerable industrial robot controllers [69].

Unfortunately, today's IoT backends cannot protect themselves from manipulations by compromised IoT devices. This is due to the lack of a technique for remotely verifying if an operation performed by an IoT device has been disrupted, or any critical data has been corrupted while being processed on the device. As a result, IoT backends are forced to blindly trust remote devices for faithfully performing assigned operations and providing genuine data. Our work aims to make this trust verifiable, and therefore, prevent compromised IoT devices from deceiving or manipulating the IoT backend.

Model privacy on mobile devices: With large and continuous R&D investment in security from smart phone vendors like Apple and Google as well as many app developers, we now feel comfortable to use our mobile devices for security-critical services like payments and online banking. However, attackers never stop to find new battle grounds especially among the newly emerging technologies. For example, with the rapid development of machine learning technology, 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, 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. 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 R&D investment of 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. Third, a leaked model facilitates malicious actors to find adversarial inputs to bypass or confuse the ML systems, which can lead to not only reputation damages to the vendor but also critical failures in their products (*e.g.,* fingerprint recognition bypass).

1.2 Thesis Statement

Driven by the great security and privacy concern about the omnipresent mobile and IoT devices, on which our life has growing dependence, my research aims to : (1) secure the IoT devices in the face of evolving attacking techniques; (2) advance the understanding of the newly emerging attack surface on mobile devices, namely the privacy problem of on-device ML models, and (3) design novel systems to protect the model privacy on mobile devices.

For IoT devices, we need to improve our attack detection capabilities. In addition to detecting traditional attacks that change the firmware, we should also develop new techniques to capture advanced attacks that can manipulate the device behavior without modifying the firmware, *e.g.,* control-flow hijacking and data-only attacks.

For mobile devices, we need to systematically study the new problem of ML model privacy, figure out how widely the problem is and what impacts it can incur, and understand the challenges of protecting on-device ML models. Besides, we should also come up with secure and efficient ways to run model inference on mobile devices without breaking existing ML applications.

1.3 Approaches Overview

In this thesis, I advance the state-of-the-art research on the security of IoT devices and mobile devices with novel security system based on hardware-backed trusted computing primitives and systematic study of the ML model privacy problem on mobile devices.

First, I present OAT [74], which is the first attestation method that captures both control-flow and data-only attacks on embedded devices. The root of security is built on top of the TEE. Using this method, IoT backends can now verify if a remote device is trustworthy when it claims it has performed an operation, sent in a service request, or transported back data from the field. In addition, unlike traditional attestation methods, which only output a binary result, our method allows verifiers to reconstruct attack execution traces for postmortem analysis.

Second, I present the first large-scale study of ML model protection and theft on mobile devices based on 46,753 trending Android apps collected from the US and the Chinese app markets [75]. Our study aims to shed light on the less 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 techniques?**
- **Q3: What impacts can (stolen) models incur?**

Finally, I propose ShadowNet as part of my thesis, which is a secure and efficient model inference system built for mobile devices. The security of ShadowNet is also rooted in the TEE. ShadowNet avoids the naive approach of moving the whole model inference into the TEE which can easily exhaust the limited resource of the TEE and lose access to the untrusted hardware accelerators. Instead, ShadowNet exploits a novel idea based on linear transformation to outsource the heavy linear layers of the model to the untrusted world (including GPU) for acceleration without leaking the model weights.

2 Background and Related Work

2.1 Attacks on IoT Devices and Backends

Embedded devices, essential for IoT, have been increasingly targeted by powerful attacks. For instance, hackers have managed to subvert different kinds of smart home gadgets, including connected lights [71], locks [52], etc. In industrial systems, robot controllers [69] and PLCs (Programmable Logic Controller) [41] were exploited to perform unintended or harmful operations. The same goes for connected cars [62, 54], drones [49], and medical devices [42, 70]. In addition, large-scale IoT deployments were compromised to form botnets via password cracking [81], and recently, vulnerability exploits [60].

Meanwhile, advanced attacks quickly emerged. Return-oriented Programming (ROP) was demonstrated to be realistic on RISC [33], and particularly ARM [59], which is the common architecture for today's embedded devices. Data-only attacks [38, 53] are not just applicable but well-suited for embedded devices [82], due to the data-intensive or data-driven nature of IoT.

Due to the poor security of today's embedded devices, IoT backends (e.g., remote IoT controllers and in-cloud services) are recommended to operate under the assumption that IoT devices in the field can be compromised and should not be fully trusted [73]. However, in reality, IoT backends are often helpless when deciding whether or to what extent it should trust an IoT device. They may resort to the existing remote attestation techniques, but these techniques are only effective at detecting the basic attacks (e.g., device or code modification) while leaving advanced attacks undetected (e.g., ROP, data-only attacks, etc.). As a result, IoT backends have no choice but to trust IoT devices and assume they would faithfully execute commands and generate genuine data or requests. This blind and unwarranted trust can subject IoT backends to deceptions and manipulations. For example, a compromised robotic arm can drop a command yet still report a success back to its controller; a compromised industrial syringe can perform an unauthorized chemical injection, or change an authorized injection volume, without the controller's knowledge.

2.2 ARM TrustZone

TrustZone is a hardware feature available on both Cortex-A processors (for mobile and high-end IoT devices) and Cortex-M processors (for low-cost embedded systems). TrustZone renders a so-called "Secure World", an isolated environment with tagged caches, banked registers, and private memory for securely executing a stack of trusted software, including a tiny OS and trusted applications (TA). In parallel runs the so-called "Normal World", which contains the regular/untrusted software stack. Code in the Normal World, called client applications (CA), can invoke TAs in the Secure World. A typical use of TrustZone involves a CA requesting a sensitive service from a corresponding TA, such as signing or securely storing a piece of data. In addition to executing critical code, TrustZone can also be used for mediating peripheral access from the Normal World.

2.3 Remote Attestation

Early works on remote attestation, such as [78][65], were focused on static code integrity, checking if code running on remote devices has been modified. A series of works [27, 68, 40, 58] studied the Root of Trust for remote attestation, relying on either software-based TCB or hardware-based TPM or PUF. Armknecht et al. [28] built a security framework for software attestation.

Other works went beyond static property attestation. Haldar et al. [79] proposed the verification of some high-level semantic properties for Java programs via an instrumented Java virtual machine. ReDAS [57] verified the dynamic system properties. Compared with our work, these previous systems were not designed to verify control-flow or dynamic data integrity. Further, their designs do not consider bare-metal embedded devices or IoT devices. Some recent remote attestation systems addressed other challenges. A tool called DARPA [55] is resilient to physical attacks. SEDA [30] proposed a swarm attestation scheme scalable to a large group of devices. In contrast, we propose a new remote attestation scheme to solve a different and open problem: IoT backend's inability to verify if IoT devices faithfully perform operations without being manipulated by advanced attacks (i.e., control-flow hijacks or data-only attacks). Our attestation centers around OEI, a new security property we formulated for bare-metal embedded devices. OEI is operation-oriented and entails both control-flow and critical data integrity.

A recent work called C-FLAT [25] is closely related to our work. It enabled control-flow attestation for embedded devices. However, it suffers from unverifiable hashes, especially when attested programs have nested loops and branches. This is because verifying a control-flow hash produced by C-FLAT requires the knowledge of all legitimate control-flow hashes, which are impossible to completely pre-compute due to the unbounded number of code paths in regular programs (i.e., the path explosion problem). In comparison, OAT uses a new hybrid scheme for attesting control-flows, which allows deterministic and fast verification. Moreover, OAT verifies Critical Variable Integrity and can detect data-only attacks, which C-FLAT and other previous works cannot.

2.4 On-device Machine Learning

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 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, 18]. However, such techniques only make it difficult to reverse a model, rather than strictly preventing a model from being stolen or reused.

2.5 Secure Machine Learning

Existing research on secure machine learning covers both the end devices and the cloud. Offline Model Guard (OMG) [31] provides a secure model inference framework for mobile device based on SANCTUARY[32], a user space enclave based on Arm TrustZone. OMG allows the model inference framework runs fully inside SANCTUARY enclave to protect model privacy. MLCapsule [51] also deploy the model on the client side to protect the user input from being sent to the untrusted cloud end. At the same time, it runs the model inference inside SGX to prevent the model from being leaked to the client. GPU is not guarded by SANCTUARY and SGX, so both OMG and MLCapsule do not support secure GPU acceleration. DarkneTZ[67] is a secure machine learning framework built on top of Arm TrustZone. It allows a few selected layers to be running inside TEE to protect part of the model. By running heavy linear layers inside TEE, it also poses resource challenges(like memory)on TEE. Comparing with OMG, MLCapsule and DarknetTZ, ShadowNet has a small TCB inside TEE and allows secure outsource of linear layers onto GPU. Graviton [80] proposes TEE extension for GPU hardware, thus allowing GPU tasks like machine learning to be running securely on GPU. It is a promising feature but requires hardware changes on GPU. Secloak [64] partitions GPU into secure world with Arm TrustZone to run GPU tasks securely at high performance penalty.

Research on securing machine learning on the cloud end is an active area. TensorSCONE[61] propose a secure machine learning framework running in the untrusted cloud. TensorSCONE integrates TensorFlow with the secure Linux container technology SCONE[29] guarded by SGX. TF Trusted [20] leverages custom operations to send gRPC messages into the Intel SGX device via Google Asylo[6] where the model is then run by Tensorflow Lite. Running model inference inside TEE faces performance challenges due to limited memory and lack of GPU acceleration. Occlumency [63] provides a suite of heuristic techniques based on Caffe and improves inference speed by 3.6 times. Despite promising, these works do not support GPU acceleration.

YerbaBuena [50] partitions the model into frontnets(like the first layer) and backnets,and execute the frontnets inside SGX to protect the user input from being leaked to the untrusted cloud while running backnets unprotected to leverage hardware acceleration. Slalom [76] splits DNN into linear and nonlinear layers, and outsources linear layers to GPU for acceleration with masked input. It verifies the linear layer's results and computes the nonlinear layers inside SGX. Slalom protects the user input privacy but not the model weights. SecureNets[39] transforms both input and linear layer's weights into matrix, and applies matrix transformation proposed in [72] to hide the non-zero elements, then sends them to the untrusted cloud for acceleration. It is not clear whether SecureNets supports depthwise convolution and convolution with stride. ShadowNet does not require transforming input and weights into matrix, and is compatible with existing linear operations.

There are also secure ML with cryptographic approach. CryptoNets[44] applies Homomorphic Encryption(HE) on neural networks and runs model inference on encrypted data to protect user input privacy. Jiang et. al [56] presents a solution to encrypt a matrix homomorphically and perform arithmetic operations on encrypted matrices. It protects both user data and model. TF Encrypted [19] enables training and prediction over encrypted data via secure multi-party computation and homomorphic encryption. SafetyNets[43] designs a Interactive Protocol(IP) that allows clients to verify the correctness of a class of DNNs running on the untrusted cloud by asking for a short mathematical proof.

To ensure the integrity of model weights, Uchida et. al [77] and Zhang et. al [84] embed watermarks into deep neural model parameters, while training the models. DeepAttest [37] encodes fingerprint in DNN weights to prevent weight modification. These works are incapable of preventing weight leakage.

3 Attesting Operation Integrity of IoT Devices

Due to the wide adoption of IoT/CPS systems, embedded devices (IoT frontends) become increasingly connected and mission-critical, which in turn has attracted advanced attacks (e.g., control-flow hijacks and data-only attacks). Unfortunately, IoT backends (e.g., remote controllers or in-cloud services) are unable to detect if such attacks have happened while receiving data, service requests, or operation status from IoT devices (remotely deployed embedded devices). As a result, currently, IoT backends are forced to blindly trust the IoT devices that they interact with.

To fill this void, we first formulate a new security property for embedded devices, called *"Operation Execution Integrity"* or *OEI*. We then design and build a system, OAT, that enables remote OEI attestation for ARM-based bare-metal embedded devices. Our formulation of OEI captures the integrity of both control flow and critical data involved in an operation execution. Therefore, satisfying OEI entails that an operation execution is free of unexpected control and data manipulations, which existing attestation methods cannot check. Our design of OAT strikes a balance between prover's constraints (embedded devices' limited computing power and storage) and verifier's requirements (complete verifiability and forensic assistance). OAT uses a new control-flow measurement scheme, which enables lightweight and space-efficient collection of measurements (97% space reduction from the trace-based approach). OAT performs the remote control-flow verification through abstract execution, which is fast and deterministic. OAT also features lightweight integrity checking for critical data (74% less instrumentation needed than previous work). Our security analysis shows that OAT allows remote verifiers or IoT backends to detect both control-flow hijacks and data-only attacks that affect the execution of operations on IoT devices. In our evaluation using real embedded programs, OAT incurs a runtime overhead of 2.7%.

In summary, our work makes the following contributions:

- We formulate a new security property, OEI, for IoT backends to attest the integrity of operations executed on remote IoT/embedded devices. It covers both control-flow and critical data integrity.
- We design a hybrid attestation scheme, which uses both hashes and execution traces to achieve complete control-flow verification while keeping the size of control-flow measurements acceptable for embedded devices.
- We present a light-weight variable integrity checking mechanism, which uses selective and value-based checking to keep the overhead low without sacrificing security.
- We design and build OAT to realize OEI attestation on both the prover- and verifier-side. OAT contains the compile-time, run-time, and verification-time components.
- We evaluate OAT on five real-world embedded programs that cover broad use scenarios of IoT devices, demonstrating the practicality of OAT in real-world cases.

3.1 Example: A Vulnerable Robotic Arm

In this example of a vulnerable robotic arm (Listing 1), the function `main_looper` first retrieves an operation command from a remote controller and stores it in `cmd` (Line 11). The looper then reads a peripheral sensor (Line 12), which indicates if the arm is ready to perform a new operation. If ready, the looper finds the operation function specified by `cmd->op` (Line 15) and calls the function with the parameters supplied in `cmd->param` (Line 16). One such function (Line 25) moves the arm to a given position.

We introduce an attacker whose goal is to manipulate the operation of the robotic arm without being detected by the remote controller who uses the existing attestation methods. For simplicity, we assume the attacker has tampered with the sensor and uses it to feed exploit input to the robotic arm. This is realistic given that such external peripherals are difficult to authenticate and protect. The target of the attacker is Function `get_input` (Line 12), which contains a buffer overrun. The vulnerability allows malformatted input to be copied beyond the destination buffer (`peripheral_input`) into the subsequent stack variables (e.g., `cmd`).

By crafting input via the compromised sensor, the attacker can launch either control hijacking or data-only attacks on the robotic arm. To hijack the control, the attacker overwrites both `cmd->param` and `cmd->op` as a result of the buffer overrun exploit, which leads to the execution of an arbitrary operation. To mount a data-only attack, the attacker only needs to change `cmd->param` while leaving `cmd->op` intact, which turns the authorized operation harmful.

Though seemingly simple, such control and data manipulations on embedded devices are realistic and can cause severe consequences in the physical world. More importantly, existing remote attestation methods cannot detect such attacks because most of them are focused on static code integrity and none addresses dynamic data integrity. Therefore, the remote controller is unable to find that the robotic arm is compromised and did not perform the operation as commanded. Moreover, after receiving an (unverifiable) operation-success message from the compromised robotic arm, the controller may command other robotic arms to perform follow-up operations, which can cause further damage. This example illustrates the need for a new form of attestation that allows IoT backends to remotely check the integrity of operations performed by IoT devices.

```c
// Simplified control loop for robotic arms
void main_looper() {
  // Command from remote controller
  cmd_t cmd;
  // Pointer to operation function
  int (*op_func)(int, char*);
  // Input from peripheral sensor
  char peripheral_input[128];
  int st = 0;
  while(1) {
    if (read_command(&cmd)) {
      st = get_input(peripheral_input); //BUGGY!
      if (status_OK(st)) {
        // perform the operation
        op_func = get_op_func(cmd->op);
        (*op_func)(cmd->p_size, cmd->param);
      }
    }
    usleep(LOOPER_SLEEP_TIMER);
  }
  return;
}
...
// The operation that moves the robotic arm
int op_move_arm (int, char*) {...}
...
```

Listing 1: An example of a control loop in a robotic arm

3.2 Methodology

3.2.1 Operation Execution Integrity (OEI)

We propose "Operation Execution Integrity" (or OEI) as a security property for embedded devices. By verifying OEI, a remote verifier can quickly check if an *execution of an operation* suffered from **control-flow hijack** or experienced **critical data corruption**. We formulate OEI with two goals in mind: (i) enabling remote detection of aforementioned attacks; (ii) demonstrating the feasibility of an operation-oriented attestation that can detect to both control and critical data manipulations on embedded devices. Next, we formally define OEI and provide its rationale.

To avoid ambiguity, we informally define an **operation** to be a logically independent task performed by an embedded device. To declare an operation for attestation, programmers need to identify the entry and exit points of the operation in their code. For simplicity of discussion, we assume that every operation is implemented in a way that it has a single pair of entry and exit, $\langle Op_{entry}, Op_{exit}\rangle$ where Op_{entry} dominates Op_{exit} and Op_{exit} post-dominates Op_{entry} in the control flow graph. We do not pose any restriction on what code can an operation include or invoke, except that an operation cannot contain another operation.

Let $P = \{Op_1, Op_2, ...Op_n\}$ be an embedded program, composed of n operations, denoted as Op_i. $CFG(Op_i)$ is the statically constructed CFG (control flow graph) for Op_i (i.e., the CFG's root is Op_i's entry point). Let CV be the set of critical variables in P (i.e., variables affecting the execution of the operations). $D(v)$ and $U(v)$ are the def- and use-sites of a critical variable v: a set of statements where v is *defined* or *used*. $V_b(v, s)$ and $V_a(v, s)$ are the values of variable v immediately *before* and *after* statement s executes.

Definition 1. OEI: for an operation Op_i, its execution satisfies OEI $\iff$ ① the control flow complies with $CFG(Op_i)$ and maintains backward-edge integrity, and ② during the execution, $\forall cv \in CV$, the value of cv reaching each use-site is the same as the value of cv leaving the previous def-site, written as $V_b(cv, u) = V_a(cv, d)$, where $d \in D(cv)$ is the last define of cv prior to the use of cv at $u \in U(cv)$. $\triangle$

OEI entails that the control-flows and critical data involved in an operation must not be tampered with.

Operation-scoped CFI: OEI's control-flow requirement (① in Def. 1) is not a simple adoption of CFI to embedded devices, which would incur impractical time and space overhead on those constrained devices. Our operation-scoped CFI takes advantage of the operation-oriented design of embedded programs. It applies to executions of individual operations, which represent a much smaller attestation scope than a whole program and allow for on-demand attestation. It also implies backward-edge integrity (i.e., returns not hijacked).

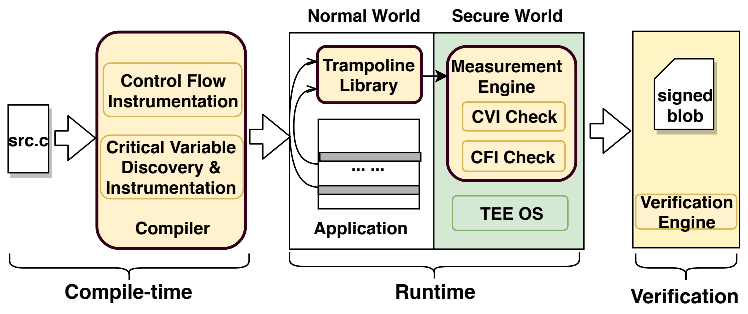


Figure 1: The Workflow of OAT, whose components (colored in yellow) include the compiler, the trampoline library, the measurement engine in the Secure World (shown in green), and the remote verification engine.

Critical Variable Integrity: We call the second requirement of OEI (② in Def. 1) Critical Variable Integrity, or CVI. It dictates that the *values* of critical variables must obey *define-use consistency*. Compared with other data integrity checkers [36, 26, 35] CVI is different in two ways. First, CVI only concerns critical variables, rather than all program data. Second, CVI uses value-based checking, instead of address-based checking, to significantly reduce code instrumentation and runtime overhead. CVI applies to the entire program execution and is not scoped by attested operations. We provide a method to assist developers to automatically identify critical variables.

Secure & Optimized Combination: OEI combines CVI and operation-scoped CFI. These two sub-properties mutually complementary. Without CVI, CFI alone cannot detect data-only attacks or control-flow hijacks that do not violate CFG (e.g., [34]). Without CFI, CVI can be bypassed (e.g., by ROP). On the other hand, this combination yields better performance than independently performing control-flow and data-flow checks. This optimized combination allows for the detection of both control-flow and data-only attacks without enforcing full CFI and DFI. It is suited for embedded devices.

Operation Verifiability: OEI caters to IoT's unique security needs. One defining characteristic of IoT devices is their frequent inter-operations with peers or cloud services. When a device finishes an operation, the operation initiator may wish to verify if the operation was executed as expected without interference. For example, a remote controller needs to verify if a robotic arm has performed a movement as instructed without experiencing any control or data manipulations. OEI attestation answers to such security needs of IoT, which are currently not addressed.

3.2.2 OAT System

We build OAT, a system to realize OEI ATtestation on ARM-based bare-metal embedded devices (i.e., no standalone OS in the Normal World). OAT consists of: (*i*) a customized **compiler** built on LLVM; (*ii*) a **trampoline** library linked to attestation-enabled programs; (*iii*) a runtime **measurement engine**; (*iv*) a remote/offline **verification engine** (Figure 1). For a given embedded program, the compiler identifies the critical variables and instruments the code for measurement collection. At runtime, the measurement engine processes the instrumented control-flow and data events and produces a proof or measurement, which the remote verifier checks and determines if the operation execution meets OEI. OAT relies on ARM TrustZone to provide the trusted execution environment for the measurement engine.

We design a hybrid attestation scheme that overcomes two challenges associated with CFI and data integrity verifications. First, remotely attesting CFI is more challenging than performing local CFI enforcement because remote verifiers can only rely on limited after-fact measurements to make the determination whereas local enforcers simply use the readily available unlimited runtime information. Furthermore, remote verifiers cannot always determine whether a hash-based control-flow measurement is legitimate or not because the verifiability is limited to those hashes pre-computed from known/traversed code paths.

Second, checking the integrity of dynamic data is known for its high overhead. Existing checkers [36, 26, 35], mostly address-based, instrument every memory-accessing instruction in a program, and during runtime, check if an instrumented instruction should be allowed to access the referenced address, based on a statically constructed access table. Even when the integrity checking is only applied to selected variables, address-based checkers would still need to instrument and check all memory-accessing instructions to ensure no unintended instructions can write to the addresses of the critical variables.

Our hybrid attestation scheme achieves complete verifiability while maintaining acceptable performance on

Table 1: Runtime overhead measured on 5 real embedded programs

Prog.	Operation Exec. Time			OAT Instrumentation Statistics					Blob	Verification
	w/o OEI (s)	w/ OEI (s)	Overhead (%)	B.Cond	Def-Use	Ret	Icall/Ijmp	Critical Var.	Size (B)	Time (s)
SP	10.19	10.38	1.9%	488	2	1946	1	20	69	5.6
HA	5.28	5.36	1.6%	147	91	33	2	6	44	0.61
RM	10.01	10.13	1.3%	901	100	100	100	7	913	1.74
RC	2.55	2.66	4.5%	14	33	1	1	8	10	0.13
LC	5.33	5.56	4.4%	931	2420	10	10	4	205	1.35
Avg.	N/A	N/A	2.7%	496	529	418	23	9	248	1.89

embedded devices. For CFI attestation, OAT's measurements contain compact control-flow transfer traces for forward-edges and fixed-length hashes for backward-edges. This combination allows remote verifiers to quickly and deterministically reconstruct control flows. It also yields size-optimized measurements. For CVI, OAT performs local verification rather than remote attestation. By doing so, OAT avoids sending a large amount of data (e.g., critical values at def- and use-sites) to remote verifiers. Sending such data would be costly for IoT devices and undesirable when privacy is concerned.

To use OEI attestation, programmers declare the to-be-attested operations in their code by using two intuitive compiler directives: `#oei_attest_begin` and `#oei_attest_end`. They may also annotate critical variables of their choice via a GCC-style attribute. For example, to enable OEI attestation in Listing 1, a programmer only needs to change Line 4, 8, and 25:

```
4  cmd_t __attribute__((annotate("critical"))) cmd;
```

```
8  int __attribute__((annotate("critical"))) peripheral_input = 0;
```

```
25 int op_move_arm (int, char*) {#oei_attest_begin ... #oei_attest_end}
```

For simplicity, our current design requires that a pair of `#oei_attest_begin` and `#oei_attest_end` is used in the same function (i.e., an operation enters and exits in the same call stack frame) and the `#oei_attest_end` always dominates the `#oei_attest_begin` (i.e., an operation always exits). Operations cannot be nested. These requirements are checked by our compiler. Developers are advised to keep the scope of an operation minimal and the logic self-contained, which is not difficult because most embedded programs are already written in an operation-oriented fashion.

As shown in Figure 1, during compilation, the customized compiler instruments a set of control flow transfers inside each to-be-attested operation. The compiler automatically annotates control-dependent variables as critical (e.g., condition variables). It also performs a global data dependency analysis on both automatically and manually annotated critical variables so that their dependencies are also treated as critical and subject to CVI checks. At runtime, the instrumented control flow transfers and critical variable define/use events trigger the corresponding trampolines, which pass the information needed for CFI or CVI verification to the measurement engine in the Secure World protected by TrustZone. Finally, the signed attestation blob (i.e., measurements) is sent to the remote verification engine (e.g., IoT backend) along with the output from the attested operation.

3.3 Evaluation

We selected 5 open-source embedded programs to evaluate the end-to-end overhead of OAT. We could not test more programs because of the non-trivial manual effort required for porting each program to our development board. This requirement is due to embedded programs' device-specific nature. It is not posed by our system. We picked these programs because they represent a reasonable level of variety. Some of them may seem toy-like, which is not intentional but reflects the fact that most embedded programs are simple by design. We note that these programs are by no means standard benchmarks. In fact, there are no standard benchmarks for bare-metal embedded devices available at the time of writing. It is also rare for embedded device vendors to publicly release their firmware in source code form. The 5 selected embedded programs are:

- *Syringe Pump* (`SP`) is a remotely controlled liquid-injection device, often used in healthcare and food processing settings. We apply OEI attestation to the "injection-upon-command" operation.

- *House Alarm System* (`HA`) [45] is an IoT device that, when user-specified conditions are met, takes a picture and triggers an alarm. We apply OEI attestation to its "check-then-alarm" operation.
- *Remote Movement Controller* (`RM`) [47] is an embedded device that allows the physical movement of its host to be controlled remotely. We attest the "receive-execute-command" operation.
- *Rover Controller* (`RC`) [48] controls the motor on a rover. We attest the "receive-execute-command" operation.
- *Light Controller* (`LC`) [46] is a smart lighting controller. We attest the "turn-on/off-upon-command" operation.

Operation Execution Time & Instrumentation Statistics: For each test program, we measured the execution times with and without OEI attestation enabled for the selected operation. The results are shown in the column named "Operation Exec. Time" in Table 1. The sub-column "Overhead" shows the relative delay caused by OAT to operation executions, averaging 2.7%. We observed that the delays are unnoticeable and blend in the much longer end-to-end execution time of the operations.

It is worth noting that the execution delay caused by our attestation may vary significantly as the following two factors change: the length/duration of the attested operation and the frequency of critical variable def-use events. For shorter operations, the attestation overhead tends to be higher percentage-wise. For instance, the operation in `RC` (the shortest among the tested programs) takes 2.55 seconds to finish without attestation and 2.66 seconds with attestation, resulting in the highest relative overhead (4.5%) among all tested programs. However, this does not mean the absolute delay, in this case, is longer than others or unacceptable.

The more frequent the def-use events of critical variables are, the higher the attestation delay becomes. For example, the operations in `HA` and `LC` are similar in lengths. But the attestation overhead on `HA` (1.3%) is lower than the overhead on `LC` (4.4%) partly because `HA` has fewer def-use events of critical variables.

In the "Instrumentation Statistics" column, we show, for each operation execution, the numbers of the instrumented events encountered during the attestation (including conditional branches, def-use checks, returns, and indirect calls/jumps) as well as the number of critical variables selected. These statistics provide some insights into the overhead reported earlier. The instrumented events occur about thousands of times during an operation, which translates roughly to an average per-operation delay of 0.15 seconds.

Measurement Engine Memory Footprint and Runtime Overhead: The measurement engine inside TEE consumes memory mainly for three purposes: the BLAKE-2s HASH calculation, the critical data define and use check, and the forward control-flow trace recording, including taken or not-taken bits and indirect jump/call destination. The BlAKE-2 HASH function only requires less than 2KB for storing the block buffer, 32 Bytes for the IV, 160 Bytes for the sigma array, and some temporary buffers. The critical data check requires a static HASH table of 4KB with 512 slots, and a dynamic pool for critical variables (the size of this pool is proportional to the number of critical variables; in our evaluation, the pool size is less than 2KB). The forward control-flow trace in our evaluation is no more than 2KB. The whole memory footprint of the measurement engine is less than 10KB for the real embedded applications used in our evaluation.

The runtime overhead of the measurement engine comes from three sources (i.e., the three major tasks of the measurement engine): calculating the hash upon function return events, recording critical data define events, and verifying critical data use event. In our evaluation, on average, processing one return event and calculating the new hash takes $0.19\,\mu$s; recording a critical data define event takes $11.04\,\mu$s; verifying a critical data use event takes $2.03\,\mu$s. Obviously, hash calculation is relatively fast whereas critical data event processing requires a longer time mainly because it involves hash table lookup or memory allocation for a new entry.

Value-based Check vs. Addressed-based Check: To show the performance difference between value-based checking (CVI) and address-based checking (e.g., DFI), we measured the number of instrumented instructions needed in both cases for all of the test programs. As shown in Table 2, on average, CVI's instrumentation is 74% less than the instrumentation required by address-based checking (i.e., a 74% reduction). Specifically, CVI's instrumentation is as little as 6.8% of what DFI requires when the program is relatively large (e.g., `RM`). The number only increases to 44.4% when we annotated most of the variables as critical in `RC`.

Space-efficiency of Hybrid Attestation: Our control-flow attestation uses the hybrid scheme consisting both forward traces and backward hashes to achieve not only complete verifiability but also space-efficiency. To quantify the space-efficiency, we compared the sizes of the control-flow traces produced by OAT (R1 in Table 3) and the traces produced by pure trace-based CFI (R2 in Table 3). On average, OAT's traces take only 2.24% of space as needed by control-flow traces (i.e., a 97% reduction). This result shows that our hybrid scheme is much better suited for embedded devices than solely trace-based CFI in terms of space efficiency.

Table 2: Number of Instrumentation Sites: Value-based (R1) and Address-based (R2)

	SP	HA	RM	RC	LC	Avg.
R1	56	37	57	20	41	-
R2	140	388	842	45	131	-
R1 / R2	40%	9.5%	6.8%	44.4%	31.2%	26%

Table 3: Control-flow Trace Size (Bytes): With Return Hash (R1) and Without Return Hash (R2)

	SP	HA	RM	RC	LC	Avg.
R1	69	44	913	10	205	-
R2	42941	3772	13713	585	13725	-
R1 / R2	0.2%	1.1%	6.7%	1.7%	1.5%	2.24%

On the other hand, compared with existing hash-based attestation schemes, OAT's attestation blobs are not of fixed-length and grow as attested operations execute, which may lead to overly large attestation blobs. However, in practice, OAT attestation blobs are reasonably small in size. Based on our experiments, the average blob size is $0.25kb$. The individual blob size for each program is shown in the "Blob Size" column in Table 1. We attribute this optimal result to two design choices: (*i*) the hybrid measurement scheme that uses fixed length hashes for verifying returns (more frequent) and traces for verifying indirect forward control transfers (less frequent); (*ii*) the operation-scoped control-flow attestation, which generates per-operation measurements and is enabled only when an operation is being performed.

4 A Large-scale Study on Insufficient Machine Learning Model Protection in Mobile Apps

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.

We 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? What impacts can (stolen) models incur? 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 and security 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.

4.1 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? (§4.2) How robust are existing model protection techniques? (§4.3) What impacts can (stolen) models incur? (§4.4)

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

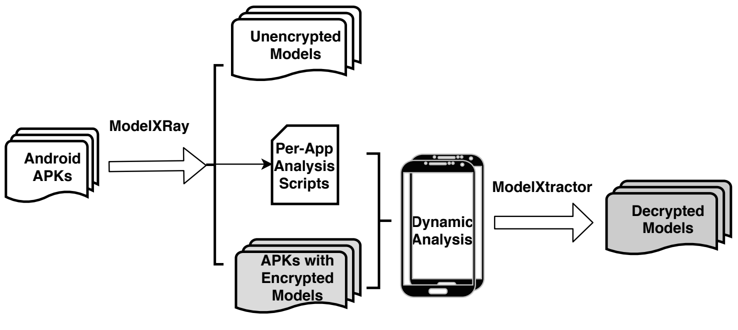


Figure 2: Overview of Static-Dynamic App Analysis Pipeline

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 Figure2. 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. We report our findings and insights drawn from the large-scale analysis results produced by ModelXRay and ModelXtractor in §4.2.2 and §4.3.2, 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 (§4.4).

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

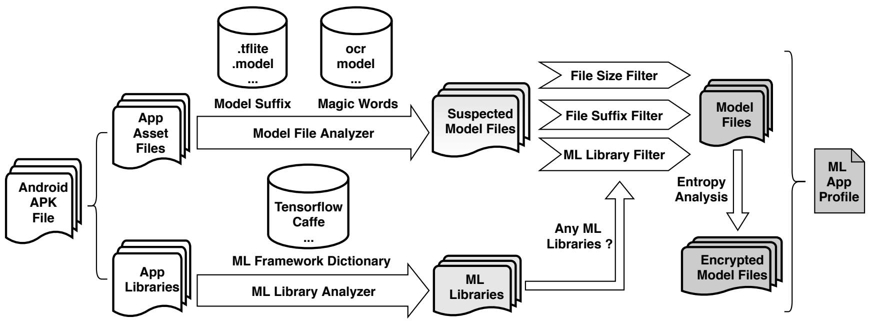


Figure 3: Identify Encrypted Models with ModelXRay

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.

4.2.1 Methodology

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 [24]. 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).

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.

We only consider encrypted models as protected in this study. We are aware that some apps obfuscate the description text in the models. Obfuscation may make it harder for the attacker to understand the model, but does not prevent the attacker from reusing it at all.

The workflow of ModelXRay is shown in Figure 3. For a 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 [83], 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. 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.

ML Model Files: To identify model files, previous work [83] 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 [10]. 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. Previous work[83] 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 §4.2.2).

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 (§4.2.2).

4.2.2 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.

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 4 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 4: 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 5.

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.

Table 5: 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.

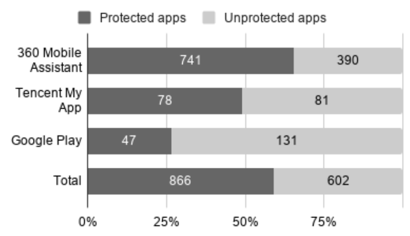
Model Protection Across App Stores: Figure 4 gives the per-app-market statistics on ML model protection and reuse. Figure 4a shows the per-market numbers of protected apps (*i.e.,* apps using protected/encrypted models) and unprotected apps (*i.e.,* apps using unprotected models).

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.

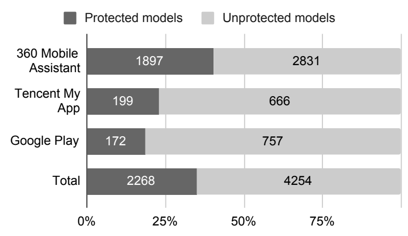
Percentages of protected models vary across app markets. When looking closer at each app market, it is obvious to see that Google Play has the lowest percentage of ML apps using protected models (26%) whereas 360 Mobile

Assistant has the highest (66%) and Tencent My App follows closely (59%). A similar conclusion can be drawn on the unique models (i.e., excluding reused models) found in those apps: 26% models in Chinese apps are protected whereas the percentage of protected models in Google Play apps is 23%. 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 [11, 23].

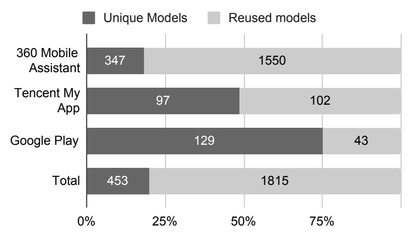
When zooming into apps and focusing on individual models (*i.e.,* some apps use multiple ML models for different functionalities), the percentages of unprotected models (Figure 4b) become even higher. Overall, 4,254 out of 6,522 models (77%) are unprotected and thus easily extractable and reverse engineered.



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

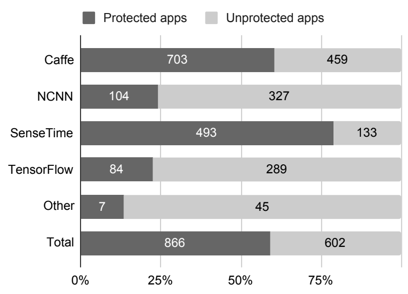


(b) On-device models that are protected/encrypted vs. those not

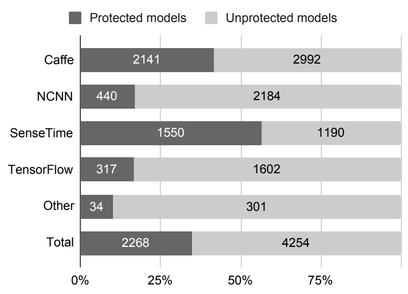


(c) Unique encrypted models vs. encrypted models reused/shared by multiple apps.

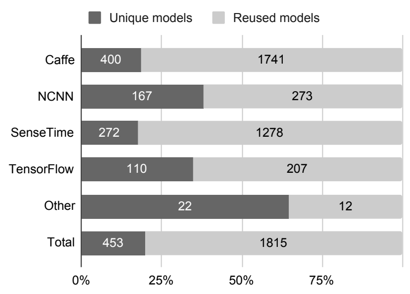
Figure 4: Statistics on ML model protection and reuse, grouped by app markets. The "total" number of unique models is less than the sum of the per-store numbers because some models are not unique from different stores.



(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 5: 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.

Model Protection Across ML Frameworks: We also derive the per-ML-framework statistics on model protection (Figure 5). The frameworks used by a relatively small number apps, including MXNet, Mace, TFLite, and ULS, are grouped into the "Other" category.

Some popular ML frameworks have wider adoption of model protection, but some not. As shows in Figure 5a, 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%.

GPU Acceleration Adoption Rate among ML Apps: Table 6 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 6: 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

4.3 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 §4.2, ModelXtractor is performed to assess the robustness of the protection.

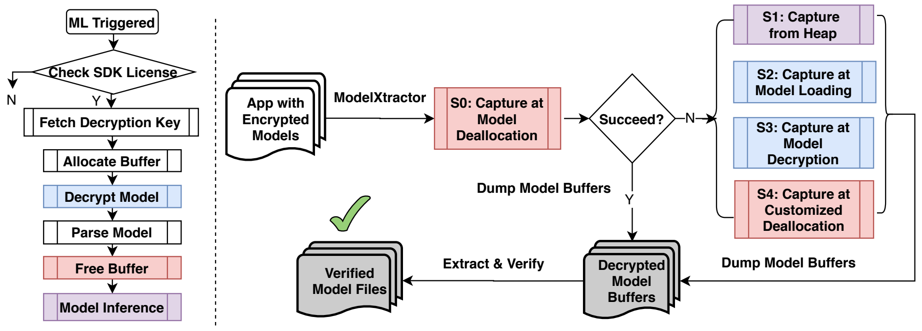


Figure 6: Extraction of (decrypted) models from app memory using ModelXtractor

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.

The workflow of ModelXtractor is depicted in Figure 6. It takes inputs from ModelXRay, including the information about the ML framework(s) and the model(s) used in the app (described in §4.2). 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 §4.3.1. Our findings,insights, and the answer to Q2 are presented in §4.3.2.

4.3.1 Methodology

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 [8], 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.

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 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.

4.3.2 Findings and Insights

Results of Dynamic Model Extraction: Table 7 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 8 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 8, 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 8 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 (§4.4).

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.

Table 7: 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.

Table 8: 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.

4.4 Q3: What Impacts can (Stolen) Models Incur?

ML models are the core intellectual properties of ML solution providers. The impacts of leaked models are wide and profound, including substantial financial impact as well as significant security implications.

4.4.1 Financial Impact

Financial impact mainly applies to two stakeholders: attackers, and vendors (model providers and app companies).

Attackers financially benefit from leaked models. App developers usually have two legitimate ways to get ML models: (1) buying a ML SDK and model license from a ML solution provider, such as SenseTime, Face++, and so on; (2) designing and training their own ML models using open-source or customized frameworks, which usually requires a large amount of computing and human resources. Stealing the models saves the attackers either the license fee paid to the model providers, or the research and development (R&D) cost on the models.

License Fee Savings for Attackers: Often, when vendors license an ML model, the app developer can choose between *online authorization* or *offline authorization*. A license with offline authorization allows a device to use the ML SDK without network connection. A company with such licenses is given unlimited uses on different devices [12]. The down side is that the model provider has no control over the number of devices or which devices to have access to the model SDKs. As a result, the model provider has no idea whether a model has been stolen or not. According to Face++, the annual fee for a license with offline authorization is $50,000 to $200,000 [12]. The saving is large enough to motivate an attacker to steal the models. In our analysis, we found 60 cases of different app companies are reusing model licenses. One of the licenses is even used by potentially 12 different app companies, indicating a high chance of illegal uses.

A license with online authorization can control the usages of the SDKs. Before using the model SDK, a device has to authenticate itself to the model provider with a license key. The model provider can then count the number of authorized devices, and charge the app company per device or per pack of devices. Online authorization offers stronger protection of the model licenses than offline authorization. However, there are still chances that attackers stealthily use a license before it reaches the limit of the current pack. The market price for face landmark SDK is $10,000 for up to 10,000 of online authorizations [12]. Even though the savings are smaller than offline authorized licenses, attackers can still benefit from them financially.

R&D Savings for Attackers: 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 or buying and maintaining storage servers and GPU clusters for training models.

According to Amazon Mechanical Turk [2], the price of labeling an object ranges from $0.012 to $0.84, depending on the type of the object (e.g., image, text, semantic segmentation). Considering the CMU Multi-PIE database as an example, which contains more than 750,000 images [21], the cost of labeling would be at least $9,000. For larger databases, for example, MegaFace with 4.7 million labels [13], or some audio and video datasets [15, 22], the cost of labeling could be even higher. According to LinkedIn statistics [16], the median base salary for machine learning engineers is $145,000 per year. Given a team with five engineers, training and fine-tuning a model for one year, the cost would be $725,000. Based on the pricing of Amazon SageMaker [3], the monthly rate for ML storage is $0.14 per GB, and the hourly rate for the current generation of ml.p3.2xlarge accelerated computing is $4.284. Still considering the CMU Multi-PIE database as an example, with a data size of 305GB, the yearly cost of data storage and training would be $38,040.

Based on the above information, a conservative estimate on the total saving for attackers on model R&D cost could be $772,040. Note, the salary of AI engineers are based on the public information of large AI companies, which can be higher than the average level. The estimation can be adjusted based on the actual salary.

Vendors face financial loss from decreasing competitive edge with stolen models. For vendors whose main business (source of income) depends on ML models, e.g., model providers or app companies, model leakages result into pricing disadvantages, lost of customers and market share.

Pricing Disadvantages for Vendors: As mentioned earlier, the cost of ML models can reach millions of dollars, thereby competitors have strong motivation towards leaked models. Once competitors start adopting leaked models with lower cost, they can offer lower prices to the customers. At the same quality, customers are more willing to choose the cost efficient products. Therefore, vendors who leak their models will lose the pricing competition in the first place.

For model providers, the market is strongly competitive. In our study, we have found some top ML SDK providers, such as SenseTime, Megvii, Baidu, ULSee, Anyline, etc. Take Megvii as an example, according to Owler [14], 10 competitors are closely related to its businesses, such as Cognitec, SenseTime, Kairos, FaceFirst, Cortexica, etc. For app companies, the competition is as much competitive if not more so. In Google Play only, our study found 36 apps using ML SDK for image recognition as the main business. Considering the other two stores, at least 215 apps are competing for this business. *If an ML model cannot be well protected, the market price of the model will be maliciously manipulated and continue to fall.*

Anticipated Falling Market Share for Vendors: The pricing disadvantage caused by leaked models will potentially result into loss of customers and fallen market share, which will both lead to significant revenue loss. Take model provider SenseTime as an example, our study found 8 unique `SenseID_OCR` models, and each is reused by 21 apps on average. Loss of one single app customer will potentially bring a loss of at least $10,000, based on the market price discussed earlier (e.g., $10,000 for up to 10,000 of online authorizations). In fact, SenseTime has more than 700 customers and partners [17], and has a revenue of $750 Million in 2019. For app companies, we also observed unbalanced market share in the 215 apps competing for the business of image recognition. The number of downloads for these apps ranges from ten thousands to one hundred million. *For both model providers and app companies, the decline in market share caused by pricing disadvantage will further lead to financial loss larger than we thought.*

4.4.2 Security Impact

Some ML models are used for security-critical purposes. For example, liveness detection model is used to verify whether it is a real person holding a real ID card. Face, fingerprint and iris recognition models are used to detect and verify the identity of a person. These models bring in great convenience, for example, users do not need to go to a bank or customer service centers to verify their identities. However, breaches of such models bring in security and privacy concerns.

For attackers, a leaked security-critical model makes it easier for them to design and craft adversarial examples. They can then use the examples to either fake different identities, or simply bypass the identity check of the apps [5].

We found more than 100 apps using on-device ML models for banking and loan services. These apps provide personal loan services aiming at quick and convenient loan applications. They use face recognition models to verify the identity of a person by taking a short video, and comparing with the photo on the ID card. The apps then determine the credit limits and rates to loan to the applicants. When the models are leaked, attackers can easily fake identities of other applicants, and apply for loans on their behalf.

In our analysis, we found 872 apps are using liveness detection models, representing 59% of all the apps using on-device ML. We also found security-critical models to be shared among different apps, for example, `SenseID_Motion_Liveness` model in the same MD5 is shared by 81 apps. Leakage of this model from any of the apps makes it easier for the attackers to bypass the detection, and the bypass will be applicable to all the 81 apps.

For end users of these apps, it further raises the concern that attackers with faked identities can access users' personal private information. For example, some apps provide online medical services, such as booking appointments, filling out medical history forms, receiving electrical prescriptions, and laboratory reports from the doctors. They may also use on-device ML models to verify the identities of patients. Bypassing the verification will allow attackers to access personal medical records. In our analysis, we found 6 such apps, which have been downloaded more than 9 million times on 360 Mobile Assistant Store. There download times can be doubled considering other Android stores. One of the face detection model, although encrypted, is shared by 77 different apps. Leakage of the model from any of the apps will potentially expose the personal medical records of mass end users. It is therefore important for vendors to protect the models, especially when they are security-critical. *Vendors should raise the awareness and understand of the potential security implications caused by leaked/stolen models.*

5 Research Plan

In previous sections, I presented the completed work on defending advanced attacks on IoT devices and understanding the newly emerging model privacy problem on mobile devices. However, how to protect the model privacy on mobile devices is still an open research problem.

Our previous work [75] on model privacy shows that even though the ML models can be encrypted by the app, it still suffers from unsophisticated runtime attacks. 54% ML apps use GPU acceleration, which means supporting GPU acceleration is very important for existing ML apps. Besides, many ML apps use on-device ML for live video stream analysis including face recognition, liveness detection and so on. With all above consideration, we want to design a secure on-device model inference system that meets the following goals:

- Security rooted in the hardware, so that the models will be secure even when the OS is compromised;
- Reasonable performance overhead, making sure it will not break existing real-time tasks;
- Access to hardware accelerators, which are being developed and used for on-device ML tasks;

TEE provides hardware-level security. Our previous work OAT [74] demonstrated that TEE can be used to build efficient attestation system with hardware-backed trusted computing primitives. However, using mobile TEE for secure model inference has several technical challenges.

First, the mobile TEE, like Arm TrustZone, is designed for small security critical services like managing encryption keys. The memory reserved for the TEE OS is limited. For example, only 14 MB is available for Trusted Applications of OP-TEE OS on Hikey960 Dev Board, while the model size of AlexNet is 242MB. It is simply not feasible to run the high resource-demanding model inference inside the TEE.

Second, the current TEE does not include GPU/NPU into the secure domain, so we will also lose access to the hardware acceleration. Third, the model inference framework will also greatly increase the TCB of TEE, risking the security of the whole system.

To address the above challenges, I propose the following work as part of my doctoral thesis.

5.1 ShadowNet: a Secure and Efficient Model Inference Scheme

We propose ShadowNet: a secure and efficient model inference scheme built on top of mobile TEE. The key observation of ShadowNet is shared with the previous research like Slalom [76] that the linear layers of CNNs occupy the majority of the model parameters and the model inference time. For example, the linear layers of MobileNets occupy around 95% of the model parameters, 99% of the model inference time. The idea of ShadowNet is to apply linear transformation on the weights of the linear layers and outsource them onto the untrusted world, so that we can leverage the hardware acceleration without trusting it. ShadowNet restores the results inside the TEE. The other nonlinear layers are also kept secure inside the TEE.

With this design, ShadowNet's security is rooted in the TEE, meeting the first design goal; ShadowNet does not introduce any heavy cryptographic operations, the performance overhead should not be heavy, meeting the second design goal; ShadowNet is still able to use the hardware acceleration meeting the third design goal. At the same time, ShadowNet solves the technical challenges of mobile TEE by keeping the TCB size small and the TEE memory usage low.

5.2 Formal Security Analysis of the Scheme

In order to apply ShadowNet scheme, we need to formally prove the scheme is secure. In detail, we need to show that ShadowNet meet two security properties: (1) by observing the weights of the transformed linear layers that is outsourced to the untrusted world, the attacker won't be able to infer the original secret weights before transformation. (2) based on the exposed transformed weights, training an equivalent model should not be made significantly easier than training the original model from scratch.

These two security property guarantee that after applying the ShadowNet transformation, the transformed model will not leak the original weights or make it easier for the attacker to train an equivalent model by outsourcing the transformed linear layers onto the untrusted hardware accelerators.

5.3 System Design Challenges

ShadowNet split the model inference between the Rich Execution Environment (REE) and the Trusted Execution Environment (TEE). The transformed linear layers can run on the untrusted hardware accelerators including GPU while the restoration of the transformed results and the model inference of non-linear layers run inside the TEE.

In order to achieve good performance, we need to overcome several system design and engineering challenges. First, the communication overhead between REE and TEE should be optimized as it happens frequently. The parameter passing between REE and TEE should be designed in a secure and efficient way. Second, writing code for TEE also faces the challenges of limited resources, like limited secure memory. The model inference code should be written in a memory efficient way. Third, mobile TEE does not have rich computation library support. For example, several popular computation libraries including Eigen Library [9] and Arm Compute Library [4] only have C++ version and do not support mobile TEE OS like the OP-TEE OS [66].

5.4 Milestones

The plan for completing my research is presented in Table 9.

Table 9: Plan for completion

Task	**Completion Date**
Finish implementation & evaluation of ShadowNet	January 2021
Formalize the security analysis of ShadowNet	February 2021
Finalizing ShadowNet for publication	March 2021
Dissertation defense	April 2021

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] Amazon SageMaker Pricing. https://aws.amazon.com/sagemaker/pricing/.

[4] Arm Compute Library. https://www.arm.com/why-arm/technologies/compute-library.

[5] Artificial Intelligence + GANs can create fake celebrity faces. https://medium.com/datadriveninvestor/artificial-intelligence-gans-can-create-fake-celebrity-faces-44fe80d419f7.

[6] Asylo - An open and flexible framework for enclave applications. https://asylo.dev/.

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

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

[9] Eigen. http://eigen.tuxfamily.org/index.php?title=Main_Page.

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

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

[12] Face++ pricing details - mobile sdk. https://www.faceplusplus.com/pricing-details/#offline.

[13] MegaFace and MF2: Million-Scale Face Recognition. http://megaface.cs.washington.edu/.

[14] Megvii's Competitors, Revenue, Number of Employees, Funding and Acquisitions. https://www.owler.com/company/megvii.

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

[16] Salary for the Machine Learning Engineer. https://www.linkedin.com/salary/machine-learning-engineer-salaries-in-san-francisco-bay-area-at-xnor-ai.

[17] SenseTime has 700+ customers and partners. https://www.forbes.com/sites/bernardmarr/2019/06/17/meet-the-worlds-most-valuable-ai-startup-chinas-sensetime/.

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

[19] TF Encrypted. https://github.com/tf-encrypted/tf-encrypted.

[20] TF Trusted. https://github.com/dropoutlabs/tf-trusted.

[21] The CMU Multi-PIE Face Database. http://www.cs.cmu.edu/afs/cs/project/PIE/MultiPie/Multi-Pie/Home.html.

[22] Video Dataset Overview - Sortable and searchable compilation of video dataset. https://www.di.ens.fr/~miech/datasetviz/.

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

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

[25] Tigist Abera, N. Asokan, Lucas Davi, Jan-Erik Ekberg, Thomas Nyman, Andrew Paverd, Ahmad-Reza Sadeghi, and Gene Tsudik. C-flat: Control-flow attestation for embedded systems software. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 743–754, New York, NY, USA, 2016. ACM.

[26] Periklis Akritidis, Cristian Cadar, Costin Raiciu, Manuel Costa, and Miguel Castro. Preventing memory error exploits with wit. In *Security and Privacy, 2008. SP 2008. IEEE Symposium on*, pages 263–277. IEEE, 2008.

[27] W. A. Arbaugh, D. J. Farber, and J. M. Smith. A secure and reliable bootstrap architecture. In *Proceedings of the 1997 IEEE Symposium on Security and Privacy*, SP '97, pages 65–, Washington, DC, USA, 1997. IEEE Computer Society.

[28] Frederik Armknecht, Ahmad-Reza Sadeghi, Steffen Schulz, and Christian Wachsmann. A security framework for the analysis and design of software attestation. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer &#38; Communications Security*, CCS '13, pages 1–12, New York, NY, USA, 2013. ACM.

[29] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'keeffe, Mark L Stillwell, et al. {SCONE}: Secure linux containers with intel {SGX}. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, pages 689–703, 2016.

[30] N. Asokan, Ferdinand Brasser, Ahmad Ibrahim, Ahmad-Reza Sadeghi, Matthias Schunter, Gene Tsudik, and Christian Wachsmann. Seda: Scalable embedded device attestation. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 964–975, New York, NY, USA, 2015. ACM.

[31] Sebastian P Bayerl, Tommaso Frassetto, Patrick Jauernig, Korbinian Riedhammer, Ahmad-Reza Sadeghi, Thomas Schneider, Emmanuel Stapf, and Christian Weinert. Offline model guard: Secure and private ml on mobile devices. *DATE 2020*, 2020.

[32] Ferdinand Brasser, David Gens, Patrick Jauernig, Ahmad-Reza Sadeghi, and Emmanuel Stapf. Sanctuary: Arming trustzone with user-space enclaves. In *NDSS*, 2019.

[33] Erik Buchanan, Ryan Roemer, Hovav Shacham, and Stefan Savage. When good instructions go bad: Generalizing return-oriented programming to risc. In *Proceedings of the 15th ACM conference on Computer and communications security*, pages 27–38. ACM, 2008.

[34] Nicholas Carlini, Antonio Barresi, Mathias Payer, David Wagner, and Thomas R Gross. Control-flow bending: On the effectiveness of control-flow integrity. In *USENIX Security Symposium*, pages 161–176, 2015.

[35] Scott A Carr and Mathias Payer. Datashield: Configurable data confidentiality and integrity. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, pages 193–204. ACM, 2017.

[36] Miguel Castro, Manuel Costa, and Tim Harris. Securing software by enforcing data-flow integrity. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, pages 147–160, Berkeley, CA, USA, 2006. USENIX Association.

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

[38] Shuo Chen, Jun Xu, Emre Can Sezer, Prachi Gauriar, and Ravishankar K Iyer. Non-control-data attacks are realistic threats. In *USENIX Security Symposium*, volume 14, 2005.

[39] Xuhui Chen, Jinlong Ji, Lixing Yu, Changqing Luo, and Pan Li. Securenets: Secure inference of deep neural networks on an untrusted cloud. In *Asian Conference on Machine Learning*, pages 646–661, 2018.

[40] Karim Eldefrawy, Aurélien Francillon, Daniele Perito, and Gene Tsudik. SMART: Secure and Minimal Architecture for (Establishing a Dynamic) Root of Trust. In *NDSS 2012, 19th Annual Network and Distributed System Security Symposium, February 5-8, San Diego, USA*, San Diego, UNITED STATES, 02 2012.

[41] Nicolas Falliere, Liam O Murchu, and Eric Chien. W32. stuxnet dossier. *White paper, Symantec Corp., Security Response*, 5(6), 2011.

[42] FDA. Cybersecurity vulnerabilities identified in implantable cardiac pacemaker, August 2017.

[43] Zahra Ghodsi, Tianyu Gu, and Siddharth Garg. Safetynets: Verifiable execution of deep neural networks on an untrusted cloud. In *Advances in Neural Information Processing Systems*, pages 4672–4681, 2017.

[44] Ran Gilad-Bachrach, Nathan Dowlin, Kim Laine, Kristin Lauter, Michael Naehrig, and John Wernsing. Cryptonets: Applying neural networks to encrypted data with high throughput and accuracy. In *International Conference on Machine Learning*, pages 201–210, 2016.

[45] Github. House Alarm System. https://github.com/ddrazir/alarm4pi.

[46] Github. Light Controller. https://github.com/Barro/light-controller.

[47] Github. Remote Movement Controller. https://github.com/bskari/pi-rc/tree/pi2.

[48] Github. Rover Controller. http://github.com/Gwaltrip/RoverPi/tree/master/tcpRover.

[49] Andy Greenberg. Hacker says he can hijack a $35 k police drone a mile away, 2016.

[50] Zhongshu Gu, Heqing Huang, Jialong Zhang, Dong Su, Ankita Lamba, Dimitrios Pendarakis, and Ian Molloy. Yerbabuena: Securing deep learning inference data via enclave-based ternary model partitioning. *arXiv preprint arXiv:1807.00969*, 2018.

[51] Lucjan Hanzlik, Yang Zhang, Kathrin Grosse, Ahmed Salem, Max Augustin, Michael Backes, and Mario Fritz. Mlcapsule: Guarded offline deployment of machine learning as a service. *arXiv preprint arXiv:1808.00590*, 2018.

[52] Grant Ho, Derek Leung, Pratyush Mishra, Ashkan Hosseini, Dawn Song, and David Wagner. Smart locks: Lessons for securing commodity internet of things devices. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, pages 461–472. ACM, 2016.

[53] Hong Hu, Shweta Shinde, Sendroiu Adrian, Zheng Leong Chua, Prateek Saxena, and Zhenkai Liang. Data-oriented programming: On the expressiveness of non-control data attacks. In *Security and Privacy (SP), 2016 IEEE Symposium on*, pages 969–986. IEEE, 2016.

[54] Troy Hunt. Controlling vehicle features of nissan leafs across the globe via vulnerable apis. https://www.troyhunt.com/controlling-vehicle-features-of-nissan/, February 2016.

[55] Ahmad Ibrahim, Ahmad-Reza Sadeghi, Gene Tsudik, and Shaza Zeitouni. Darpa: Device attestation resilient to physical attacks. In *Proceedings of the 9th ACM Conference on Security &#38; Privacy in Wireless and Mobile Networks*, WiSec '16, pages 171–182, New York, NY, USA, 2016. ACM.

[56] Xiaoqian Jiang, Miran Kim, Kristin Lauter, and Yongsoo Song. Secure outsourced matrix computation and application to neural networks. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 1209–1222, 2018.

[57] Chongkyung Kil, Emre C. Sezer, Ahmed M. Azab, Peng Ning, and Xiaolan Zhang. Remote attestation to dynamic system properties: Towards providing complete system integrity evidence. *Dependable Systems & Networks, 2009.*, 2009.

[58] Joonho Kong, Farinaz Koushanfar, Praveen K. Pendyala, Ahmad-Reza Sadeghi, and Christian Wachsmann. PUFatt: Embedded Platform Attestation Based on Novel Processor-Based PUFs. In *DAC*, page 6. ACM, 2014.

[59] Tim Kornau. *Return oriented programming for the ARM architecture.* PhD thesis, Master's thesis, Ruhr-Universität Bochum, 2010.

[60] Brian Krebs. Reaper: Calm before the iot security storm. https://krebsonsecurity.com/2017/10/reaper-calm-before-the-iot-security-storm/, October 2017.

[61] 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.

[62] Keen Security Lab. New car hacking research: Tesla motors. http://keenlab.tencent.com/en/2017/07/27/New-Car-Hacking-Research-2017-Remote-Attack-Tesla-Motors-Again/, 2017.

[63] Taegyeong Lee, Zhiqi Lin, Saumay Pushp, Caihua Li, Yunxin Liu, Youngki Lee, Fengyuan Xu, Chenren Xu, Lintao Zhang, and Junehwa Song. Occlumency: Privacy-preserving remote deep-learning inference using sgx. In *The 25th Annual International Conference on Mobile Computing and Networking*, pages 1–17, 2019.

[64] Matthew Lentz, Rijurekha Sen, Peter Druschel, and Bobby Bhattacharjee. Secloak: Arm trustzone-based mobile peripheral control. In *Proceedings of the 16th Annual International Conference on Mobile Systems, Applications, and Services*, pages 1–13, 2018.

[65] Yanlin Li, Jonathan M. McCune, and Adrian Perrig. VIPER: verifying the integrity of peripherals' firmware. In *Proceedings of the 18th ACM Conference on Computer and Communications Security, CCS 2011, Chicago, Illinois, USA, October 17-21, 2011*, pages 3–16, 2011.

[66] Linaro. OP-TEE. https://www.op-tee.org.

[67] Fan Mo, Ali Shahin Shamsabadi, Kleomenis Katevas, Soteris Demetriou, Ilias Leontiadis, Andrea Cavallaro, and Hamed Haddadi. Darknetz: Towards model privacy at the edge using trusted execution environments. In *ACM MobiSys 2020*.

[68] Bryan Parno, Jonathan M. McCune, and Adrian Perrig. Bootstrapping trust in commodity computers. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, SP '10, pages 414–429, Washington, DC, USA, 2010. IEEE Computer Society.

[69] D. Quarta, M. Pogliani, M. Polino, F. Maggi, A. M. Zanchettin, and S. Zanero. An experimental security analysis of an industrial robot controller. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 268–286, May 2017.

[70] Rapid7. Multiple vulnerabilities in animas onetouch ping insulin pump. https://blog.rapid7.com/2016/10/04/r7-2016-07-multiple-vulnerabilities-in-animas-onetouch-ping-insulin-pump/, October 2016.

[71] Eyal Ronen, Adi Shamir, Achi-Or Weingarten, and Colin O'Flynn. Iot goes nuclear: Creating a zigbee chain reaction. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 195–212. IEEE, 2017.

[72] Sergio Salinas, Changqing Luo, Weixian Liao, and Pan Li. Efficient secure outsourcing of large-scale quadratic programs. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, pages 281–292, 2016.

[73] Christos Stergiou, Kostas E Psannis, Byung-Gyu Kim, and Brij Gupta. Secure integration of iot and cloud computing. *Future Generation Computer Systems*, 78:964–975, 2018.

[74] Zhichuang Sun, Bo Feng, Long Lu, and Somesh Jha. Oat: Attesting operation integrity of embedded devices. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1433–1449. IEEE, 2020.

[75] Zhichuang Sun, Ruimin Sun, and Long Lu. Mind your weight (s): A large-scale study on insufficient machine learning model protection in mobile apps. *arXiv preprint arXiv:2002.07687*, 2020.

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

[77] Yusuke Uchida, Yuki Nagai, Shigeyuki Sakazawa, and Shin'ichi Satoh. Embedding watermarks into deep neural networks. In *Proceedings of the 2017 ACM on International Conference on Multimedia Retrieval*, pages 269–277. ACM, 2017.

[78] Arvind Seshadri Adrian Perrig Leendert van Doorn Pradeep Khosla. Using software-based attestation for verifying embedded systems in cars. *S&P, Oakland*, 2004.

[79] Deepak Chandra Vivek Haldar and Michael Franz. Semantic remote attestation – a virtual machine directed approach to trusted computing. *VM*, 2004.

[80] 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.

[81] Wikipedia. Mirai(malware). https://en.wikipedia.org/wiki/Mirai_(malware), 2020.

[82] Jacob Wurm, Khoa Hoang, Orlando Arias, Ahmad-Reza Sadeghi, and Yier Jin. Security analysis on consumer and industrial iot devices. In *Design Automation Conference (ASP-DAC), 2016 21st Asia and South Pacific*, pages 519–524. IEEE, 2016.

[83] 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.

[84] 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.