Breaking SafetyCore: Exploring the Risks of On-Device AI Deployment

Guyomard Victor
Skyld AI
Rennes, France
victor.guyomard[at]skyld.io

Mathis Mauvisseau
Skyld AI
Rennes, France
mathis.mauvisseau[at]skyld.io

Paindavoine Marie
Skyld AI
Rennes, France
marie[at]skyld.io

***Abstract*—Due to hardware and software improvements, an increasing number of AI models are deployed on-device. This shift enhances privacy and reduces latency, but also introduces security risks distinct from traditional software. In this article, we examine these risks through the real-world case study of SafetyCore, an Android system service incorporating sensitive image content detection. We demonstrate how the on-device AI model can be extracted and manipulated to bypass detection, effectively rendering the protection ineffective. Our analysis exposes vulnerabilities of on-device AI models and provides a practical demonstration of how adversaries can exploit them.**

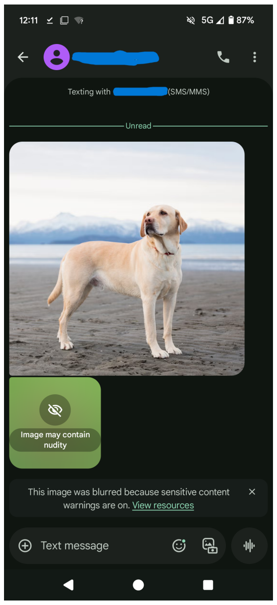
***Index Terms*—Reverse engineering, Model extraction, Adversarial examples**

I. Introduction

Today, an increasing number of AI[1] models are deployed on-device. This trend is driven by the growing computational power of modern hardware, especially with the adoption of specialized components like Neural Processing Unit (NPU)[1]. Moreover, advancements in AI software, such as quantization, have reduced both the size of models and the computational power required to run them, making on-device deployment more feasible and efficient. This form of deployment offers key advantages such as improved data privacy, since data is processed directly on the device, and reduced latency, as inference no longer depends on an internet connection[2]. However, while on-device AI deployment is becoming more popular, its security implications are still often misunderstood, especially when considering how AI differs from traditional software. This gap in understanding can lead to serious vulnerabilities particularly when AI is used in security applications such as content filtering or spam detection.

In this article, we explore the security risks associated with on-device AI deployment through the lens of a real-world case study: the exploitation of the SafetyCore application[3]. SafetyCore is an Android Google system service introduced in November 2024. It provides a service used by other Android applications: The classification of sensitive or problematic content, such as nudity in images. The content detection is performed using an AI based algorithm that is locally embedded on the device for privacy-preserving reasons. This means that the user data is not sent to a remote server, but is kept on

[1]In this article, we use the term AI to specifically refer to deep learning neural networks.



Texting with [redacted] (SMS/MMS)
This image was blurred because sensitive content warnings are on. View resources

Fig. 1. User interface of the Google Messages application with SafetyCore enabled. When nudity is detected by the AI model, the image is blurred and a warning message is displayed to the user.

the device. Currently, SafetyCore is only used by the Google Messages application for content moderation. However, other applications, such as WhatsApp[4], are expected to adopt it in the near future. The AI model used by SafetyCore takes an image as input and predicts whether it contains sensitive content or not. If such content is detected, the image is automatically blurred, and a warning message is displayed to inform the user that the image may contain unwanted characteristics, such as nudity. An example of the application behavior is shown in Figure 1.

SafetyCore relies on the AI model's ability to make accurate predictions based on pixel values in the image. Starting with the extraction of the embedded model, we demonstrate how adversaries can manipulate images to cause misclassifications, thereby rendering the protection mechanism ineffective. The objectives of this article are twofold:

- Explaining the specific risks of deploying AI models on-device, especially for readers without a background in AI.

- Providing a practical guide to extracting and exploiting these models in a real-world setting.

Each aspect of our analysis is illustrated using the SafetyCore attack case study. This attack has been performed on a Google Pixel 6, running Android 15 (build `BP1A.250305.019`) with SafetyCore (`com.google.android.safetycore`) version `1.0.757930370`.

We begin this article by examining the specific methods for reverse-engineering an AI model and discuss what makes AI models fundamentally different from traditional software. We then describe the pre-processing and conversion steps necessary to turn an extracted model into a targeted object. Finally, we explore intrinsic vulnerabilities of AI models and demonstrate how they can be exploited through AI based attacks.

II. The Reverse Engineering Challenge

This Section explores what makes AI models fundamentally different from standard code, and why traditional software protections are often insufficient to secure them from reverse-engineering.

A. What is Inside an AI Model?

The goal of an AI model is to perform a given task (prediction, generation) on data never seen before. It is defined by an architecture composed of layers, hyperparameters and learned parameters. Layers represent mathematical operations, often linear transformations such as matrix multiplications. Each layer has hyperparameters, whose values are fixed before training and remain unchanged. In contrast, the learned parameters, such as weights and biases, are updated throughout the training process so that the model can perform its task on new data. An example of a toy AI model is provided in Figure 2. The lifecycle of an AI model can be divided into two phases: training and inference. Training is the step where the learned parameters are updated, i.e. the network learns to adapt itself to new data. The inference step is when the model is used to perform some prediction on unseen data. At this stage, the learned parameters are fixed and no longer updated.

B. Why AI Models are Different from Classical Software?

Software is typically made of code that is compiled and deployed. This means that only the hardware instructions are present on the deployment target. **On the other hand, an AI model is usually stored in a file**. This file does not directly contain the hardware instructions, as traditional software does, but rather a serialized version of the algorithm. This serialized file defines the layer operations as well as the learned parameters needed to produce the model output. The specific implementation of those operations is handled by a separate inference engine. To parse and run a model, the inference engine associated with the model is required.

The operations used during inference are implemented by the inference engine. Thus, a model can only use a limited set of standardized layers. **When an AI model is run, the executed operations came from the same finite set of layers, regardless of the specific model architecture or parameters values.** The inference engine is contained in a compiled library and defines how the model is serialized. **A limited set of inference engines are often used to deploy AI models, and most of them are open-source**. This is because layer implementations are highly optimized for specific hardware, making it inefficient and unnecessary for each company developing AI models to re-implement them from scratch for all standard hardwares[5].

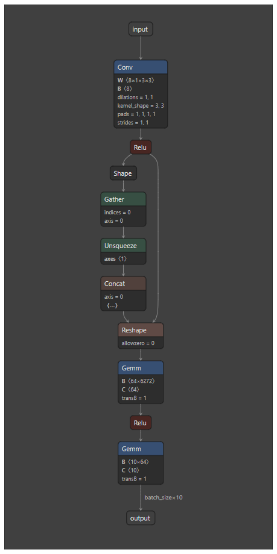


Fig. 2. Example of a toy AI model (in ONNX format). Each node of the graph corresponds to a specific layer. The first node is a convolution layer that contains learned parameters W, b and hyperparameters $dilation, kernelShape, pads, stride$.

While this standardization is beneficial for performance and cross-platform compatibility, it raises significant challenges regarding intellectual property protection.

C. Static Extraction

The first method to reverse engineer a model is to perform a static analysis (i.e., analysis without running the application) of the application to locate and extract AI models. Since serialization depends on the specific library used, it is important to understand the different libraries and their formats. The major inference engines used when deploying an AI model on-device include LiteRT[6] (formerly TensorFlow Lite), ONNX[7], and PyTorch (through TorchScript[8] and ExecuTorch[9] formats). **Each of these engines reads AI models from a file that has distinctive identifiable characteristics**. This makes it relatively easy to locate such files within a software package, enabling static file analysis. Table I lists the different characteristics that can be searched for in the files of an application to locate AI models.

a) The SafetyCore case: In the case of SafetyCore, static analysis of the service's downloaded files revealed a TensorFlow Lite model. The presence of the ASCII-encoded magic value `TFL3` at byte offset 4 led to a quick identification of the model file.

TABLE I
CHARACTERISTICS TO IDENTIFY AI MODELS USED BY MAJOR INFERENCE ENGINES.

Inference engine	**Characteristic**
LiteRT / TensorFlow Lite	FlatBuffers file identifier* `TFL3` (ASCII encoded)
ONNX	Protobuf containing the graph. Each layer type starts with `onnx::`
TorchScript	ZIP archive (file signature `PK\x03\x04`) containing: • The directories: – `code` – `data` • The files: – `data.pkl` – `constants.pkl`
ExecuTorch	FlatBuffers file identifier* `ET??` followed by `eh??` (ASCII encoded), where `?` is a digit.

*The FlatBuffers file identifier is a field in the FlatBuffers serialization format.

D. Dynamic Extraction, or Why Encryption Is Not Enough

In some cases, the AI model in plain-text form never touches persistent storage. For instance, when it is downloaded (remote loading) and loaded directly into memory for each inference, or stored only in encrypted form (model encryption).

- **Remote loading** avoids static interception of the model file as it is never stored in persistent storage.
- **Model encryption** effectively hides the file structure, making it impossible to locate using known characteristics of the model while performing a static analysis.

In such situations, dynamic analysis can be used to intercept the serialized model at runtime, capturing it while it is being loaded into the inference engine. This was not the case for SafetyCore, where the model was recovered through static analysis. In practice, encryption and remote loading can be bypassed, and the model extracted using dynamic analysis. While having privileged access on the running device, instrumentation tools such as Frida[10] can be used to hook the model loading functions and exfiltrate the model during execution. **Since most inference engines are open source, it is relatively easy to identify and hook the model loading function, even if it is not directly exposed by the library.**

III. AI MODEL REFINEMENT

After this first reverse-engineering step, an AI model often requires refinement before it can be effectively exploited.

A. Convert to the Right Format

Extracted AI models are often not immediately usable by attackers, because they are deployed in formats that do not support gradient computation (Additional details about gradients are provided in Section IV-A). PyTorch is the most widely used framework for attacking AI models. In contrast, formats such as TFLite and ONNX do not natively allow gradient computation, making them not suitable for direct exploitation. Therefore, converting the extracted model to PyTorch is typically a necessary step. This conversion can often be achieved using available tools, either directly or through a combination of intermediate formats. In Table II is presented common AI model formats and the corresponding tools used to convert them into PyTorch.

Original Format	**Conversion Tool(s)**
ONNX	`onnx2pytorch[11]`
TFLite	(`tf2onnx` + `onnx2pytorch`) via REOM[12]
TorchScript	Natively exploitable
ExecuTorch	Not currently supported*

TABLE II
CONVERSION TOOLS FOR ENABLING PYTORCH BASED ATTACKS ON EXTRACTED AI MODELS.

*Primarily due to the novelty of the framework compared to more mature alternatives such as TFLite.

B. The Quantization Problem

Quantization refers to the process of converting the learned parameters of an AI model from high-precision floating-point (typically *float32*) to lower-precision formats such as 8-bit integers (typically *int8*)[13]. This transformation reduces both memory usage and computational cost, making it particularly suitable for on-device deployment, where hardware resources are limited[13].

The most widely used approach is *affine quantization*. This method relies on two quantization parameters:

- a scale factor $s \in \mathbb{R}^+$.
- a zero point $z \in \mathbb{Z}$.

These parameters are used both to:

- Convert (quantize) the original *float32* parameters to integers.
- Compute operations directly in the quantized domain (integer domain).

Given a real-valued parameter $w \in \mathbb{R}$, its quantized representation $w_q \in \mathbb{Z}_{[\alpha_q,\beta_q]}$ is computed as:

$$w_q = \text{clip}\Big(\text{round}\big(\frac{1}{s}w + z\big), \alpha_q, \beta_q\Big)$$

Since the model no longer operates on differentiable *float32* parameters, standard gradient-based techniques cannot be directly applied to a quantized model. However, it is important to note that **quantization does not act as a security mechanism**. The combination of quantized parameters and their associated scale and zero point is sufficient to reconstruct an approximation of the original parameters. For $w \in \mathbb{R}$ we have:

$$w \approx \text{dequantize}(w_q, s, z) = s \cdot (w_q - z) \qquad (1)$$

Using this equation, an attacker can construct a proxy model, i.e., a model that approximates the behavior of the original one. This proxy model is fully differentiable and can be attacked using standard gradient-based methods.

a) The SafetyCore case: In the case of the SafetyCore application, the target model was provided in the TFLite format. As shown in Table II, REOM[12] allows the conversion of a TFLite model to PyTorch by leveraging a combination of two intermediate conversion tools. Additionally, REOM integrates a quantization module that applies Equation 1 to recover *float32* parameters from *int8* parameters, enabling the construction of the proxy model. The tool successfully generated a *float32* proxy model that could be subjected to further attacks[2].

IV. Exploiting AI Models

After transforming a model into a usable artifact, we analyze the vulnerabilities of AI systems and the unique security challenges they pose. We then present how to exploit these vulnerabilities through adversarial examples. Finally, we discuss additional attacks that are relevant once an AI model is extracted.

A. Intrinsic Vulnerability of AI Models

The intrinsic vulnerability of AI relies on three intricate problems:

a) Gradient manipulation: A neural network is a highly complex function that takes an input and, using a set of parameters, produces an output. These parameters must be learned in order for the model to generate meaningful results. To learn these parameters, we define an auxiliary function, the loss function, which tells us how much the model is wrong in its prediction. For instance, in an image classification task, the loss function could quantify how inaccurately the model distinguishes between images of cats and dogs. The goal is typically to minimize this loss function. Training the model involves updating its parameters to reduce the loss, using a dataset of input/output pairs (known as the training set). This process is often performed using an algorithm called gradient descent, which iteratively adjusts the parameters in the direction that reduces the loss. The gradient is a mathematical object that indicates how to change the parameters to minimize the loss.

During the training phase, the gradients are computed with respect to the model parameters to minimize the loss. **Once the model is trained, however, an attacker can instead compute gradients with respect to the input, this time to *maximize* the loss i.e. make the model's prediction as wrong as possible. In this setting, the gradient reveals how the input should be perturbed to mislead the model.** Because neural networks are highly complex and operate in high-dimensional input spaces, these perturbations can be crafted so that they remain imperceptible to humans, making them particularly dangerous.

[2]For our proxy model, we did not add additional layers to simulate quantization errors, as we observed no significant differences between the quantized and the reconstructed model.

b) The black-box problem: Despite their remarkable performance, AI models are black-boxes in the sense that the decision-making process is not understandable by humans[14]. In other words, given a particular input, we often cannot understand why the model produces a specific output[15]. Although the field of explainable AI (XAI) has made significant progress, it remains difficult to predict how a model will behave on unseen or slightly altered inputs. This inherent opacity creates "gray areas" of unpredictable or unintuitive behavior that are exploitable. **These unpredictable behaviors are not easily identifiable or interpretable by human observers, making them ideal entry points for adversarial manipulation.**

c) Features correlation: AI models typically rely on statistical correlations in the training data rather than causal relationships. This distinction is critical: a model might learn that "A" often co-occurs with "B," without grasping whether "A" causes "B." **This reliance on correlation rather than causation contributes to unexpected and or unintelligible model behavior that can bypass human judgment.**

B. Exploiting These Vulnerabilities

The vulnerabilities presented above can be exploited in multiple ways across the AI lifecycle. In this Section, we focus on concrete attack strategies that target on-device AI models when having access to the architecture and parameters.

a) Inferring the loss function: Many attacks rely on gradient computations, which not only require knowledge of the model's architecture and parameters but also defining a suitable loss function. The choice of this loss function is driven by the identification of the task the model is solving, for example a binary classification task. **A common attack strategy is to identify the loss that was used for model training, and use the opposite for attacking (in order to maximize it). While this information is typically unavailable after deployment, attackers can often infer it through careful analysis of the architecture and model behavior.**

This process typically involves the following steps:

1) **Architecture probing**: By analyzing the metadata (e.g., input/output shapes, presence of specific layers such as convolutions or residual blocks, and even embedded strings in the model file), one can make educated guesses about the model architecture and its intended task. For some model formats, such as TFLite or ONNX, the overall architecture can be visualized using a visualization tool like Netron[3].
2) **I/O probing**: By feeding the model with various sample inputs and observing the output responses, one can understand the appropriate input format and the semantics of the outputs (classification scores, images, heatmaps etc...).
3) **Output layer inspection**: The final activation function often reveals the nature of the learning problem. A softmax activation suggests a classification task, and a

[3]http://netron.app

sigmoid (logistic) activation a multi-label classification task. A linear output usually indicates regression.

b) The SafetyCore case: In the case of the SafetyCore, the input tensor shape is $1 \times 224 \times 224 \times 3$, which strongly suggest image data. The architecture includes residual connections common in ResNet architectures[16] for image classification. The output shape is 1×4, and when probing the model with explicit versus non-explicit images, we observe that explicit content causes some output values to exceed 0.5, while benign content remains below this threshold. This, along with the presence of a sigmoid activation function before the output, suggests that the model is solving a multi-label classification problem, and was trained using a binary cross-entropy loss. This inferred loss enables the attacker to compute gradients for further manipulations.

1) Adversarial Examples: **The philosophy behind adversarial examples involves defining a desired criterion on the model's output through a loss function, and then using model gradients to find an input modification that satisfy this criterion.** This attack relies mostly on the properties of high dimensional spaces and the nature of the functions learned by deep learning models. In these spaces, a tiny perturbation, when applied in a specific direction (using gradient), can often cause a significant change in the model's output.

Generally, the input change is sought to be imperceptible and to maximize the model's output change[17]. For classifiers this could mean altering a picture of a panda so that the model confidently predict a gibbon[17]. There are two main types of adversarial attacks:

- **Untargeted attacks:** The goal is to mislead the model, regardless of what that incorrect output is.
- **Targeted attacks:** The goal is to produce a predetermined output. The attacker doesn't just want the model to be wrong, he wants it to produce a particular output.

For generating these examples, a diverse range of attack algorithms has been developed, from the fast gradient sign method (FGSM)[17] to more iterative and powerful methods like Projected Gradient Descent (PGD)[18], each with different trade-offs in terms of computational cost and effectiveness[19].

It is important to note that the effectiveness of adversarial examples depends on the data modality of the input. They are particularly effective on continuous data, such as images and audio, where small perturbations can be applied directly to the input using gradient-based methods[18]. In contrast, generating adversarial examples for discrete data (like text) is more challenging, as it is difficult to generate discrete changes in a meaningful way using gradients[20]. However, many text models include both discrete and continuous components. In such cases, it is often possible to generate adversarial perturbations in the continuous space and then extrapolate them back to the discrete space[20].

Adversarial examples are extremely powerful in practice. Because of the high dimensionality of the input space, it is not feasible to simply "patch" a given adversarial example by specifically instructing the model to ignore it during training. Doing so leaves the model vulnerable to countless regenerated variants that came from the same region of the input space. A common defense strategy is adversarial training[21], which involves incorporating multiple adversarial examples in the model training. While it offers a potential defense, it remains difficult to fully mitigate the threat, as this approach often involves a trade-off with model performance[21].

a) The Safetycore case: For SafetyCore, we implemented a **Projected Gradient Descent (PGD)** attack, a widely used method known for its effectiveness in white-box setting (e.g when access to the model parameters).

This allows two types of attacks:

1) **False Positive (Enable Blurring):** You can start from a non-explicit image and generate a small, imperceptible perturbations to make the model predicting it as explicit. As a result, SafetyCore will apply blurring to an image that should not be blurred.
2) **False Negative (Bypass Blurring)** Yo can also start from an explicit image. This image will contains a small perturbation that prevent the model from recognizing it as explicit. Consequently, SafetyCore will not apply any blurring to it, effectively bypassing the protection.

In Listing 1 a PyTorch implementation of the attack script is provided. This script is intended to be simple and as generic as possible and can be reused on other models as long as the input data is continuous, and a loss function can be chosen. The most important parameters of this script are ϵ and num_iter.

- ϵ control the maximum change per pixel that is expected for the perturbed input. Higher values mean higher pixels variations and then more visible perturbations. You can gradually increase ϵ until you find a sample with the expected model output.
- num_iter control the number of iteration (number of gradient steps) that are taken during the attack. A higher number of iterations allows finding a more effective adversarial example in terms of loss function maximization.

Before running the attack script, the input images are resized to $1 \times 224 \times 224 \times 3$ to match the model's input dimensions. The resulting adversarial examples have the same size and are saved in a .png format. It is important to use an image format that does not apply compression in order to preserve the adversarial perturbation (avoiding JPEG, which would remove part of the added noise).

In Figure 3, we show three benign images that have been perturbed using PGD (Enable Blurring case). When passed through the originally extracted model, these adversarial images are misclassified as explicit content. In Figure 3, we also present a screenshot from the Google Messages app showing how these images appear to users: all are blurred with a warning about potential explicit content. You can imagine the same scenario with explicit images that will not be blurred in the application (Bypass Blurring). For ethical reasons, this case is not illustrated in the article. Executing an "enable

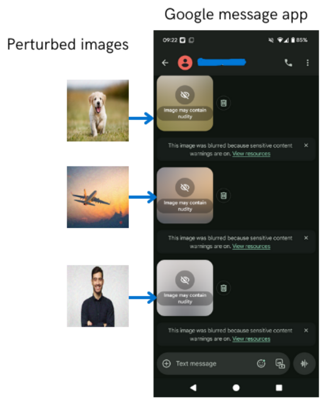

Google message app
Perturbed images
09:22
85%
[redacted]
Image may contain nudity
This image was blurred because sensitive content warnings are on. View resources
Image may contain nudity
This image was blurred because sensitive content warnings are on. View resources
Image may contain nudity
This image was blurred because sensitive content warnings are on. View resources
Text message

Fig. 3. Adversarial attack on SafetyCore. On the left, are provided the perturbed images obtained after a PGD attack, and on the right what appears in the Google Message application when sent to an Android device.

blurring attack" or a "bypass blurring attack" follows the same methodology, allowing the adversary to choose either at will.

The implications of this attack are severe: since the same AI model is shared across all Android devices, it is possible to fully bypass the filtering capabilities, regardless of the input. This attacks takes less than 30 lines of code.

2) Additional Relevant Attacks: Even if it is very powerful, adversarial examples generation is not the only relevant attack that can be performed after extracting an AI model. In this Section we presented two additional types of methods that can be applied.

a) Model inversion: Model inversion attacks aim to reconstruct representations of the training data by using the AI model itself[22]. Typically, these attacks exploit model gradients to iteratively optimize an input that maximizes the model's confidence[22]. This technique is most commonly applied to classification tasks, where the goal is to generate inputs that are representative of a target class. The consequences can be severe, often resulting in the leakage of private data for example, reconstructing faces that were used to train a facial recognition model. Additionally, model inversion can provide insights into the task the model was trained on. For instance, if the reconstructed inputs resemble airplanes and the model architecture indicates a classification task, one can reasonably infer that the model is likely classifying different types of planes from images.

In the context of SafetyCore, we attempted a model inversion attack. However, the optimization process quickly collapsed. This failure can be attributed to two main factors. First, the training data for each class is probably highly diverse, making it difficult for the model to converge toward a shared representation. Second, the task is a multi-class classification problem, which further complicates the inversion process, as each class may share overlapping features with others.

b) Backdoor Attacks: Backdoor attacks involve poisoning the training data with specially crafted samples that cause the model to learn spurious correlations between a trigger pattern and a specific output[23]. When the attacker later inputs a sample containing the same trigger, the model exhibits the intended behavior[23].

This type of attack exploits two vulnerabilities described in Section IV-A:

- The model is learning arbitrary associations (like small patch for images) that have no causal relation to the task.
- The black-box nature of AI means that such malicious correlations are difficult to interpret or detect after training.

In the case of SafetyCore, this type of attack may be unnecessary, as adversarial examples can achieve the same effect without requiring the model to be retrained. Moreover, there is no indication that the submitted images will be used for future training. In fact, since the model runs entirely on-device, retraining seems unlikely. However, unlike adversarial examples, which may not transfer to new model versions, backdoor examples are more likely to persist. Indeed, as backdoors are difficult to detect in training data, such examples could remain in future training sets, potentially preserving the backdoor across versions.

```python
import torch
from torch import nn

def pgd_attack(
    model, inputs, targets, epsilon=0.01, alpha=0.005, num_iter=100,
    loss_fn=None, random_start=True, clip_min=0.0, clip_max=1.0):
    """
    Projected Gradient Descent (PGD) attack on a PyTorch model.

    Parameters:
    -----------
    model : torch.nn.Module
        The neural network to attack.
    inputs : torch.Tensor
        Original input images or continuous data to perturb.
        TODO: Replace by your own input
    targets : torch.Tensor
        Ground truth labels corresponding to the inputs.
    epsilon : float
        Maximum perturbation allowed (L-infinity norm).
    alpha : float
        Step size for each iteration.
    num_iter : int
        Number of attack iterations.
    loss_fn : callable, optional
        Loss function to maximize (defaults to BCELoss if None).
        TODO: Replace by your own loss function
    random_start : bool
        If True, start from a random point within the epsilon-ball around the input.
    clip_min : float
        Minimum allowed value for perturbed inputs.
    clip_max : float
        Maximum allowed value for perturbed inputs.

    Returns:
    --------
    torch.Tensor
```

```
        Adversarially perturbed inputs.
    """
    model.eval()
    original_inputs = inputs.clone().detach()

    if random_start:
        # Start from a random point within the epsilon-ball
        adv_inputs = original_inputs + torch.empty_like(inputs).uniform_(-epsilon, epsilon)
        adv_inputs = torch.clamp(adv_inputs, clip_min, clip_max)
    else:
        adv_inputs = original_inputs.clone().detach()

    if loss_fn is None:
        # Loss function used for the attack
        # TODO: Replace by your own loss function
        loss_fn = nn.BCELoss()

    for _ in range(num_iter):
        adv_inputs.requires_grad_(True)
        outputs = model(adv_inputs)
        loss = loss_fn(outputs, targets)

        model.zero_grad()
        loss.backward()
        grad_sign = adv_inputs.grad.detach().sign()

        adv_inputs = adv_inputs + alpha * grad_sign
        # Project back to the epsilon-ball and clip to valid range
        perturbation = torch.clamp(adv_inputs - original_inputs, min=-epsilon, max=epsilon)
        adv_inputs = torch.clamp(original_inputs + perturbation, clip_min, clip_max).detach()

    return adv_inputs
```
Listing 1. Small generic Python code for generating adversarial examples with PGD

V. Conclusion

Security should serve as a cornerstone for building trust in AI systems. In this paper, we explored the risks of deploying AI models on-device through the lens of the SafetyCore application. Our work demonstrates that once adversaries gain access to the model, it can be compromised with relative ease and rendered ineffective, raising important concerns for the security of on-device AI based applications. While on-device AI enables countless use cases, its specific security challenges are still overlooked, even by major players in the field as illustrated by the SafetyCore example. This work acts as a foundation for understanding and running attacks on on-device AI models, and can be extended to a wide range of applications. In future work, we plan to extend our methodology to more data modalities and use-cases.

References

[1] Vivienne Sze, Yu-Hsin Chen, Tien-Ju Yang, and Joel S. Emer. Efficient processing of deep neural networks: A tutorial and survey. *Proceedings of the IEEE*, 105(12): 2295–2329, 2017. doi:10.1109/JPROC.2017.2761740.

[2] Kimberly Jane. Edge ai devices for multimodal document intelligence: Designing low-latency, privacy-preserving systems for on-device fraud prevention. 03 2025.

[3] Google. Safetycore (android system), 2025. URL https://play.google.com/store/apps/details?id=com.google.android.safetycore&hl=fr.

[4] Stan Kaminsky. Whatsapp integration, 2025. URL https://www.kaspersky.fr/blog/what-are-android-safetycore-and-key-verifier/22653/.

[5] Vivienne Sze, Yu-Hsin Chen, Tien-Ju Yang, and Joel Emer. Efficient processing of deep neural networks: A tutorial and survey, 2017. URL https://arxiv.org/abs/1703.09039.

[6] Google. Litert inference engine, 2025. URL https://ai.google.dev/edge/litert?hl=fr.

[7] Microsoft. Onnx inference engine, 2025. URL https://onnx.ai/.

[8] Meta. Torchscript inference engine, 2025. URL https://docs.pytorch.org/docs/stable/jit.html.

[9] Meta. Executorch inference engine, 2025. URL https://docs.pytorch.org/executorch/stable/index.html.

[10] Frida. https://frida.re/. Dynamic instrumentation toolkit.

[11] ENOT developers, Igor Kalgin, Arseny Yanchenko, Pyoter Ivanov, and Alexander Goncharenko. onnx2torch. https://enot.ai/, 2021. Version: x.y.z.

[12] Mingyi Zhou, Xiang Gao, Jing Wu, Kui Liu, Hailong Sun, and Li Li. Investigating white-box attacks for on-device models, 2024. URL https://arxiv.org/abs/2402.05493.

[13] Markus Nagel, Marios Fournarakis, Rana Ali Amjad, Yelysei Bondarenko, Mart van Baalen, and Tijmen Blankevoort. A white paper on neural network quantization, 2021. URL https://arxiv.org/abs/2106.08295.

[14] Zachary C. Lipton. The mythos of model interpretability, 2017. URL https://arxiv.org/abs/1606.03490.

[15] Victor Guyomard, Françoise Fessant, Tassadit Bouadi, and Thomas Guyet. Post-hoc counterfactual generation with supervised autoencoder. pages 105–114, 2021.

[16] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778, 2016. doi:10.1109/CVPR.2016.90.

[17] Ian J. Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and harnessing adversarial examples, 2015. URL https://arxiv.org/abs/1412.6572.

[18] Aleksander Madry, Aleksandar Makelov, Ludwig Schmidt, Dimitris Tsipras, and Adrian Vladu. Towards deep learning models resistant to adversarial attacks, 2019. URL https://arxiv.org/abs/1706.06083.

[19] Joana C. Costa, Tiago Roxo, Hugo Proença, and Pedro Ricardo Morais Inácio. How deep learning sees the world: A survey on adversarial attacks & defenses. *IEEE Access*, 12:61113–61136, 2024. ISSN 2169-3536. doi:10.1109/access.2024.3395118. URL http://dx.doi.org/10.1109/ACCESS.2024.3395118.

[20] Javid Ebrahimi, Anyi Rao, Daniel Lowd, and Dejing Dou. Hotflip: White-box adversarial examples for text classification, 2018. URL https://arxiv.org/abs/1712.

06751.

[21] Weimin Zhao, Sanaa Alwidian, and Qusay H. Mahmoud. Adversarial training methods for deep learning: A systematic review. *Algorithms*, 15(8), 2022. ISSN 1999-4893. doi:10.3390/a15080283. URL https://www.mdpi.com/1999-4893/15/8/283.

[22] Matt Fredrikson, Somesh Jha, and Thomas Ristenpart. Model inversion attacks that exploit confidence information and basic countermeasures. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, page 1322–1333, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450338325. doi:10.1145/2810103.2813677. URL https://doi.org/10.1145/2810103.2813677.

[23] Yang Bai, Gaojie Xing, Hongyan Wu, Zhihong Rao, Chuan Ma, Shiping Wang, Xiaolei Liu, Yimin Zhou, Jiajia Tang, Kaijun Huang, and Jiale Kang. Backdoor attack and defense on deep learning: A survey. *IEEE Transactions on Computational Social Systems*, 12(1): 404–434, 2025. doi:10.1109/TCSS.2024.3482723.