Preventing Neural Network Model Exfiltration in Machine Learning Hardware Accelerators

Mihailo Isakov, Lake Bu, Hai Cheng, and Michel A. Kinsy
Adaptive and Secure Computing Systems Laboratory
Department of Electrical and Computer Engineering, Boston University
Email: {mihailo, bulake, chenghai, mkinsy}@bu.edu

***Abstract*—Machine learning (ML) models are often trained using private datasets that are very expensive to collect, or highly sensitive, using large amounts of computing power. The models are commonly exposed either through online APIs, or used in hardware devices deployed in the field or given to the end users. This provides an incentive for adversaries to steal these ML models as a proxy for gathering datasets. While API-based model exfiltration has been studied before, the theft and protection of machine learning models on hardware devices have not been explored as of now. In this work, we examine this important aspect of the design and deployment of ML models. We illustrate how an attacker may acquire either the model or the model architecture through memory probing, side-channels, or crafted input attacks, and propose (1) power-efficient obfuscation as an alternative to encryption, and (2) timing side-channel countermeasures.**

***Index Terms*—Neural network, model exfiltration, hardware security, model theft, memory probing, side-channels, inference.**

I. Introduction

The number of parameters in machine learning models is rapidly increasing and larger datasets are being used to attain higher accuracies [1]. As a result, these models require more training time and compute resources. Altogether, some of the factors contributing to the large upfront cost of deploying machine learning models are:

Datasets: The datasets required for training DNNs (Deep Neural Networks) are very large, and are both resource and time consuming to gather. Datasets are often private, and can be very expensive to buy, if at all available for purchase. They are often the main business advantage of incumbent companies with large data collection teams. An entrant in a certain field must invest not only considerable resources to acquire a similar dataset, but must also wait for some period of time to collect the dataset. Therefore, any model trained on a large, proprietary or hard to collect dataset, i.e., an expensive dataset, should also be considered expensive.

Compute power: The computation required to train modern ML algorithms is growing exponentially. OpenAI has highlighted that the number of petaflops reported in ML papers is doubling every three months [1]. The energy consumption alone makes these models expensive, not accounting for the cost of the training hardware.

Metaparameter search: The final neural network topology and metaparameters used in these systems are not obvious during the development of these models. Researchers and developers typically have to run many searches to narrow down the neural network parameters, e.g., the number and types of layers, the connectivity of these layers, learning rate, regularization functions, etc. The time to arrive at an efficient architecture may be far greater than the time to train it.

The high cost of developing and training these models makes them highly attractive targets for attackers to steal. In [2], the authors illustrate several attacker motivations: (i) avoiding the query charges of models locked behind APIs (Application Programming Interface). In other words, an attacker may steal a model in order to avoid having to pay to use it in the model's native environment behind an API. Extracting the model through the API may initially be expensive, but can pay off during the lifetime of the model. (ii) Learning the training dataset: attackers who have access to the model or the API can craft queries that reveal the training data or some properties of the training dataset [3]. This attack highly depends on the machine learning algorithm, as some algorithms, e.g., SVMs (Support Vector Machine), tend to retain more information about the original training dataset, while others, e.g., decision trees, remember very little. Nonetheless, having access to the model can expose information about biases or the distribution of the underlying training set. (iii) Evasion and adversarial attacks: attackers who have access to the model can find ways to trick the model into producing incorrect outputs, possibly evading detection by the model as in the case of spam filters and malware detection [4]. Adversarial attacks find input samples that are correctly classified by humans or domain experts, but will cause the model to produce a random or specific output. Adversarial examples are easier to find if the attacker has access to model weights. This gives attackers the motivation to first exfiltrate the model, and then find adversarial examples.

Beyond the deployment scenarios mentioned above when the ML model is behind an API, the 'non-API' models are also susceptible to various attacks. Broadly speaking, there are three main classes of 'non-API' model deployment contexts. (1) The model is ran on a private network for internal use by the organization that created it, e.g., stock market prediction models. (2) The results of the ML model are indirectly accessed online, as in the case of most recommender systems. Users do not directly interact with the API of the model, rather there is an interactive translation layer between the user's actions and the results from the model. This deployment modality is still vulnerable to API attacks, yet the interface may appear obfuscated. (3) The ML model is loaded onto a device or directly implemented in hardware, e.g., voice assistants, self-driving cars, etc.

In context (1), the organization can use conventional security approaches to protect the model against attacks, as they would do for any other type of data. Without loss of generality, one can classify the context (2) vulnerabilities under the API attack surface listed in [2]. In this work, we focus on the third context, where the user has physical control over the device housing the ML model.

II. Related Work

Stealing machine learning models through an API has been proposed in [2]. For a simple neural network, the authors devise an equation-solving algorithm that can reconstruct the original model. Deep neural networks do not have analytical solutions, but authors show that these models can be reconstructed with high accuracy using an approach similar to model distillation [5]. Reverse-engineering networks has been further explored in [6], where the authors show an approach that can extract the network and metaparameters, and help find better adversarial examples. A method for defending against model exfiltration has been proposed in [7], where the authors introduce an algorithm that actively analyzes network inputs, and detects all of the known model extraction attacks. Machine learning model fingerprinting has been suggested as a way to counter IP (Intellectual Property) theft. In [8], the authors propose to 'backdoor' the network, where the network owner adds specific input-output pairs to the training set, and later uses the inputs as a challenge to the network, proving their ownership of it. This approach can be circumvented by fine-tuning or pruning the network. Cloud inference on private data has been explored in CryptoNets [9]. The authors use homomorphic encryption to run calculations on encrypted user data without ever having to decrypt it, but this approach slows down inference by several orders of magnitude. In DeepSecure [10], the authors propose a significantly faster approach based on Yao's Garbled Circuits. In Chameleon [11], the authors allow multiple parties to compute functions without having to reveal their private data.

III. Motivation

There are many reasons why a user may want to run an ML model on a hardware device under their control as opposed to accessing it online:

Privacy: Either for privacy or legal reasons, a user is not able to send their data to the cloud for processing. Hospitals might be legally prohibited from sending patient data off their premises. Baby camera users may care about privacy and want the data processed on the camera itself, with only alerts leaving the device. Processing data in-situ removes the risk of a malicious cloud service or a man-in-the-middle attacker stealing the data.

Power: In many cases, sending data over a GSM (Global System for Mobile) network is more expensive than processing the data in-place. In [12], authors show that the optimal power usage in convolutional neural networks is achieved by running several of the lower convolutional layers on the device before sending the activations over the network.

Latency and network access: Sending data off the device may be too slow in many latency-sensitive applications. With the round-trip time in hundreds of milliseconds for cellular networks, transmitting data to a server will break the time budget of many applications. Also, if the device must guarantee high availability, it may not be able to rely on the network at all times.

Throughput: In data-intensive applications, even when the device has network access, it may not have enough bandwidth to send the data out. The device must then either store the data or immediately process it.

A. Exfiltration of Models in Hardware Devices

The majority of existing work assumes that the neural network models are running in the cloud. In this work we consider hardware devices running proprietary machine learning models. This encompasses all devices that are under the user's control - e.g., personal computers, smartphones, wearables, voice assistants, vehicles, smart home appliances, etc. We do not consider hardware devices that are not under users' control. If the user can access the model through the device's API, but not the device itself, then the processing platform poses lesser risk. Neural network hardware devices introduce a whole new set of attacks, while still being vulnerable to API-based attacks. These attacks include (1) memory probing, where an attacker reads the ML model from non-volatile memory (NVM), (2) side-channel attacks, where the attacker learns about the model by observing device power, memory access patterns, or timing information, and (3) evading hardware API protections like limiting the frequency of inference operations.

B. Trusted Inference Engine (TIE)

In this work, we propose defenses that guard against physical attacks, side-channel attacks, and API attacks. These defenses prevent physical exfiltration of machine learning models from RAM and non-volatile memory, stop the leakage of topological and metaparameter information through side-channels, and make API attacks infeasible. We develop a hardware module called the *Trusted Inference Engine* (TIE). The engine can be built into existing hardware accelerators, and serves to (1) protect non-volatile memory against probing attacks, and (2) prevent API-based extraction by ensuring rate-limiting operations. Furthermore, we present a TIE-based model distribution protocol that allows ML model suppliers to securely distribute their models without the risk of exfiltration.

IV. Preventing Physical Memory Attacks

The theft of ML/NN models can be carried out in different ways. The model, consisting of weights, metaparameters and a topology, the model transmission process and associated cryptographic keys, and modes of operation such as the number of inference operations carried out by the model, are all vulnerable to attack. Therefore, their protection requires a systematic, composable and comprehensive approach. In this work, we present such an approach rooted in a hardware module, called the Trusted Inference Engine or TIE. Before introducing the TIE module, we first give an overview of the envisioned deployment environment and ecosystem.

A. The TIE Ecosystem

The TIE ecosystem is a three-party framework consisting of a network provider (NP), a user (U), and a hardware device manufacturer (DM) as shown in Figure 1. The security foundation of this ecosystem is based on the trustworthiness of the DM and NP. This means that the DM will manufacture the inference devices providing the assurance that (a) no hardware back-doors or Trojans have been injected and (2) no confidential information about the devices can be leaked to the user. Meanwhile, the user can act maliciously. They can apply different attack vectors on the model provided by the NP or on the hardware device itself. We assume that although the user is capable to apply different attacks, e.g., memory probing or side-channel analysis, they cannot probe the chip internals.

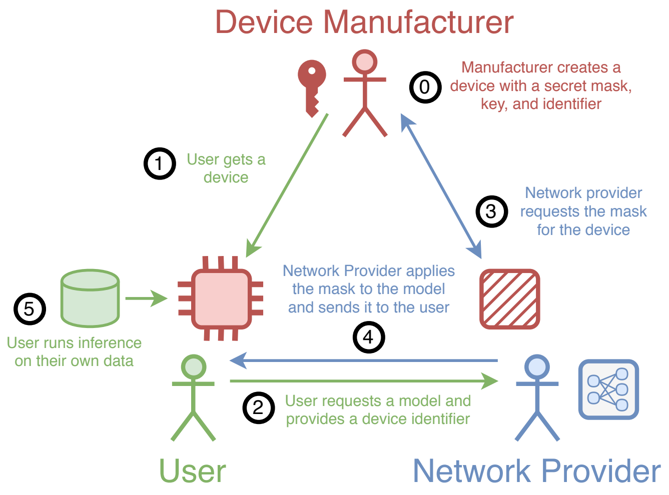
Device Manufacturer
0 Manufacturer creates a device with a secret mask, key, and identifier
1 User gets a device
3 Network provider requests the mask for the device
Network Provider applies the mask to the model and sends it to the user
4
5 User runs inference on their own data
2 User requests a model and provides a device identifier
User
Network Provider

Fig. 1. The three-party ecosystem uses a six-step protocol to implement exfiltration-resistant ML models. The security of the protocol is rooted in hardware anchored by the TIE device to enable private transfers of ML models.

We introduce the following functions to describe the key concepts used in the presentation of the ecosystem's protocol.

- $\mathrm{KeyGen}(Id)$: For a given TIE device - with identity tag Id - the DM uses the $\mathrm{KeyGen}()$ function to generate and record a number of challenge-key pairs $\{(challenge, key)\}$ based on the hardware uniqueness of the TIE chip;
- $\mathrm{DevAuth}(Id)$: A secure device authentication function (i.e., a public key based authentication protocol) is used to verify the identity Id of the device. The returned value of the function is $true$ or $false$;
- $\mathrm{KeyRetrv}(Id, challenge)$: This function is used by the network provider (NP) when deploying a model M on the device with identity Id to retrieve the unique secret key associated with $(Id, challenge)$;
- $\mathrm{Enc}(model, key)$: The returned object of this function is the obfuscated model E_M based on the retrieved key;
- $\mathrm{Dec}(E_M, key)$: In the deployment environment, i.e., the user side, this function is executed to de-obfuscate E_M and return the plain model M. For this operation, the corresponding key held in the TIE module is used. The key cannot be probed from outside the engine and remains oblivious to the user of the model.

Protocol II-A: The secure ML model deployment using the TIE ecosystem is as follows:

0. The DM manufactures the TIE chips, and uses the $\mathrm{KeyGen}(Id)$ function to record their challenge-key pairs $\{(challenge, key)\}$;
1. A user U planning to use a certain ML network obtains a TIE chip tagged by Id_i from the DM;
2. U then requests the model $model_x$ from the NP, this request includes the user device identifier Id_i;
3. The NP and DM together use the $\mathrm{DevAuth}(Id_i)$ to authenticate the user device Id_i. If the function returns $true$, the NP acquires a $\{(challenge_i, key_i)\}$ associated with the device from the DM;
4. The NP fetches the model M_x, fuses in it the conditions and terms of use, obfuscates the model - $E_{x\text{-}i} = \mathrm{Enc}(M_x, key_i)$, and sends it with $challenge_i$ to the user. The $E_{x\text{-}i}$ allows U to use the model for some limited k instances - specified in the terms of use;
5. The user loads $E_{x\text{-}i}$ and $challenge_i$ into the TIE chip. The chip internally uses the $\mathrm{KeyRetrv}(Id_i, challenge_i)$ function to retrieve key_i, which is used in the $\mathrm{Dec}()$ function to decode the obfuscated model. It is worth emphasizing that in this phase of the deployment protocol, the user does not know key_i and cannot acquire the output of the $\mathrm{Dec}(E_{x\text{-}i})$ operation. ■

Although PROTOCOL IV-A provides a relatively detailed insight to the deployment ecosystem and its security, there are still some design challenges to be addressed. For example, how can the network provider prevent a user from sharing a model with other users? How can the protocol restrict or enforce certain usage scenarios, e.g., not allow the user to repeatedly apply $E_{x\text{-}i}$ and $challenge_i$ to the TIE chip in order to reconstruct the model? In general, how can the terms of use be enforced as part of the security and trustworthiness of the deployment protocol? We extend the TIE module and the overall architecture to provide support for this functionality.

B. The TIE Architecture

The design objective of the Trusted Inference Engine (TIE) is twofold: (1) it provides a programmable template for network provider defined terms of use - e.g., restrict the number of invocations for a model to a predetermined value k, and (2) it prohibits a user from accessing the native ML model or gaining any side-channel knowledge about the original model or its training conditions. The TIE security is rooted in the hardware. In this work, we introduce two illustrative TIE designs (i) a budget-bounded design and (ii) a performance-oriented design.

Figure 2 shows a depiction of the TIE hardware and its three main components:

- An ML/NN Inference Accelerator;
- A decoder (DEC) to carry out the $\mathrm{Dec}(E, key)$ function;
- A Programmable Usage Controller (PUC) that carries out the $\mathrm{KeyRetrv}(Id,\ challenge)$ function and enforces the model's use conditions.

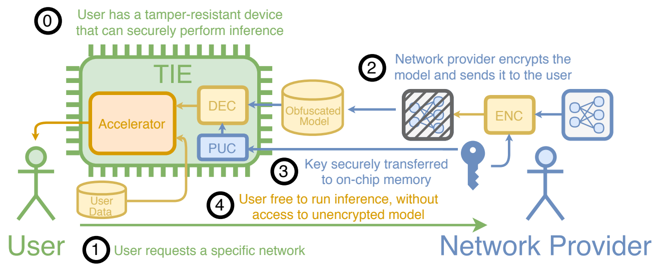
0
User has a tamper-resistant device that can securely perform inference
TIE
Accelerator
DEC
PUC
Obfuscated Model
2
Network provider encrypts the model and sends it to the user
ENC
3
Key securely transferred to on-chip memory
4
User free to run inference, without access to unencrypted model
User Data
User
1
User requests a specific network
Network Provider

Fig. 2. The Trusted Inference Engine (TIE) hardware and its operations.

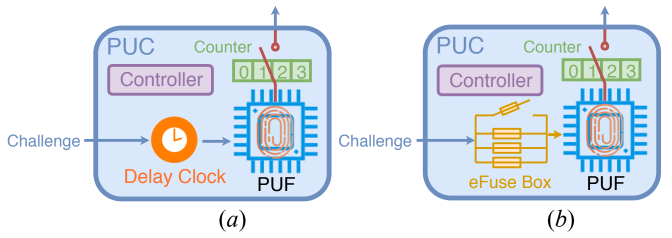

Fig. 3. Design (a) is a budget-bounded approach that uses a delay clock at every boot-up and design (b) is a performance-aware approach that uses eFuse to control the number of inference runs.

1) The Programmable Usage Controller (PUC) Module: The PUC module is one of the key components of the inference engine and the root of its security. It stores and administers the secret keys needed for the model de-obfuscation operation. It also performs the in-deployment model usage policy checking. For example, it can be used to control the total number of times that a user can perform inference with a model or the rate at which new inputs are fed to the model.

Key retrieval: To decrease the risk of (i) device counterfeiting and (ii) model exfiltration and reuse on other hardware, we propose bounding the model de-obfuscation process to the TIE device. To achieve this tight coupling, we leverage the hardware uniqueness of the TIE hardware using a PUF (Physical Unclonable Function) based key generation approach [13]. Different PUFs, with different security guarantees, can be used to anchor the security of the system based on the deployment environment, i.e., susceptibility to attacks. When a Network Provider (NP) receives a model request from a user with a TIE device, the NP replies with a model obfuscated using that TIE device's identifier Id. The obfuscation operation is performed using one of the TIE PUF responses registered with the Device Manufacturer (DM). It should be highlighted that if a man-in-the-middle attacker attempts a model request without a valid registered Id, the provider will discover it during the process of obtaining a PUF response. In step 3 shown in Figure 2, the corresponding challenge is sent to the user. Here, there is no need to use an expensive eavesdrop-resistant channel. The user also has no knowledge of the response (decoding key), since the $\mathrm{KeyRetrv}(Id, \mathit{challenge})$ function operates inside the TIE hardware that cannot be probed. Therefore, modeling attacks on PUFs do not apply to this scenario. In addition, without owning the device linked to the model, a user cannot successfully decode the obfuscated model. Therefore, the model cannot be operated separately from the intended user's device.

Programmable usage control to protect against passive attacks: As mentioned in section I, with a large enough number of input-output pairs, the underlying ML model can be learned [2]. It has been shown that attackers can reconstruct the entire model with only access to the input and output interfaces of the model. To protect the ML model against these passive attacks, we introduce the concept of a programmable usage control (PUC) hardware sub-module and validate its usefulness through two usage condition policies, i.e., rate-limiting and inference-limiting policies. We propose two usage control designs as illustrative cases - shown in Figure 3. Both designs have a counter that allows the user to run a certain number of ML operations per second. When TIE reaches that upper limit in a specified timespan, it will stop working until the timespan ends. For a budget-bounded design, to prevent a user from simply restarting the device to reset the counter, a clock enforces some predetermined delay at each boot-up before the PUF sub-model can generate a decoding key. This design supports a rate-limiting usage policy, but not an operation-limiting one. In some critical applications, the network designer may want to record the number of inference operations that a TIE device is performing, e.g., by having the device report its inference count to a server. In the performance-oriented design, the one-time read hardware primitive eFuse is used [14]. At each boot-up, after the decoding key is read, an eFuse in the fuse box burns out. The TIE hardware runs this way until the counter reaches a preset limit. When all the eFuses burn out, the TIE hardware is no longer usable. While extreme, the fuses prevent a malicious user from getting around rate and operation counters through a hard device reset. Denoting the safe threshold of the number of runs that a given model can run in a year ($\approx 3.2 \times 10^7$ seconds) as n_s, the length of time required to run the ML model one time as t_m seconds, the upper limit of the counter as c_{max}, and the clock wait time as T for Figure 3 (a) is:

$$T \geq \frac{3.2 \times 10^7}{n_s} - t_m \cdot c_{max}. \tag{1}$$

If the number of eFuses in the fuse box is f, then for Figure 3 (b) we have:

$$f \geq \frac{n_s}{c_{max}}. \tag{2}$$

One can use these two bounds during the design of the TIE hardware. The use of eFuses provides a stronger rate-limiting policy than the delay clock based approach, since the TIE device stops functioning when the user runs out of eFuses.

2) The ENC (Obfuscation) and DEC (De-obfuscation): The ENC and DEC modules in Figure 2 are used to ensure that the ML model is unreadable to the user and any man-in-the-middle attackers. They carry out the $\mathrm{Enc}()$ and $\mathrm{Dec}()$ functions respectively. There are various approaches for obfuscating and de-obfuscating these models. The main selection criterion is based on security-to-cost trade-off. For example, while the encryption-based obfuscation provides higher security, the power consumption is much larger than one using a pseudo-random number generator (PRNG)-based approach, shown in Figure 4. Here, the PRNG obfuscates the model by generating a *one-time-pad* the size of the model, and XOR-ing the pad

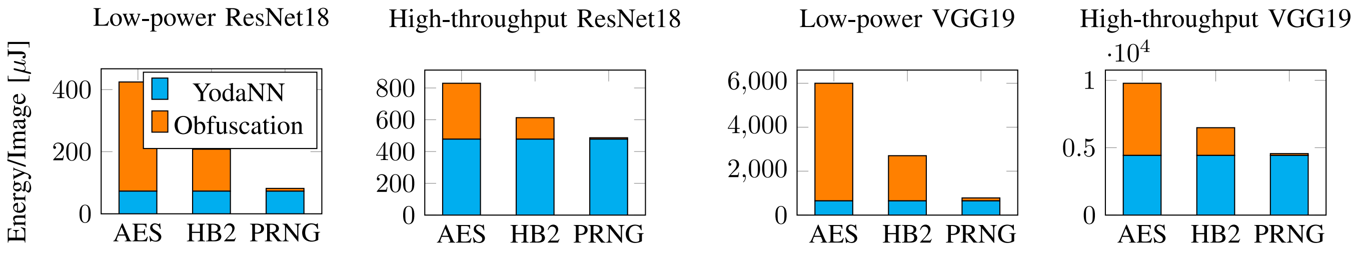

Fig. 4. YodaNN accelerator [15] and encryption power usage breakdown in low-power and high-throughput regimes on ResNet18 and VGG19 models. For both ResNet18 and VGG-19 running on YodaNN, the PRNG's power consumption is almost negligible. In case of AES and Hummingbird2 (HB2), decrypting models can be more energy-intensive than processing them.

with the model. Since the model does not change, repeated transfers do not reveal any information. Three obfuscation approaches compared in this work are: AES, Hummingbird 2 (HD2), and PRNG. Cryptographic encryption schemes could be used in performance-oriented designs, and the cryptographic secure pseudo-random generator (CSPRNG) based techniques in budget-bounded designs. In the former case, the PUF can be used to generate and retrieve the encryption keys, and the latter case to seed the CSPRNG.

V. Preventing API-based Attacks in Hardware

Two key works targeting exfiltration of machine learning models through APIs are [2], [6]. In [2] the authors show that exfiltrating simple models is trivial. However, for deep neural networks, attackers need to gather orders of magnitude more input-output pairs than the model has parameters. We propose several defenses against API-based attacks:

Reducing model output size: In [2], the authors show that if the classification APIs do not report confidence values, but only the most likely category of an input, then the model are more resilient to exfiltration. There are two limitations with this approach: (1) it can only be applied on classification problems, and not on other techniques like regression methods, and (2) applications like image segmentation have a model output as large as the model input. For example, in pix2pix [16], the output of the neural network is a per-pixel labeled version of the input. Even if only the highest class or 'color' is reported, the model still produces a very large output and reveals a lot of information. Therefore, limiting the output information of a model can mitigate some of the model exfiltrating risks.

General-purpose processing in TIE: In a similar vein, not directly exposing the input and output interfaces of the model will reduce its attack surface. For this purpose, we propose expanding the TIE hardware to enclose as many modules as feasible that are directly consuming the output of the model. For example, in a self-driving car, instead of executing just the computer vision algorithms on the TIE-enabled processor, mapping and planning algorithms could also be run on the TIE, and the only observable or accessible outputs are steering decisions. To obfuscate memory accesses related to read and write operations associated with the other modules, one could use an oblivious RAM [17] (ORAM) structure to interact with the TIE.

Programmable runtime usage control - rate-limiting case: To illustrate how one may implement the runtime usage policies, we extend the TIE architecture to include rate-limiters. These circuits prevent a certain operation, e.g., inference, from being run more than a specific number of times per second. While simple devices may use their own throughput as a rate-limiter, e.g., a voice assistant may not be able to physically process inference operations faster than it already does, devices with hard latency requirements may come equipped with far more computing power than is necessary. The network provider can then specify the inference rate based on (1) the application latency requirements, and (2) the trade-off between the risk of model exfiltration and the quality of service to be provided to the end user. The hardware rate-limiters monitor and enforce the specified usage terms.

Inference-limiting: The network provider may want to not only limit the inference rate, but rather the total number of inference operations run with a certain model. As mentioned in section IV, a TIE device may have a hardware kill-switch preventing overuse of certain operations, e.g., inference.

VI. Preventing Architecture and Metaparameter Leakage through Probing and Side-Channels

Exfiltrating models through APIs is significantly harder if the attacker does not know the architecture of said models [2]. An attacker with physical control of the device can use different side-channels with varying invasiveness to discover the topology and metaparameters of the network running on the device. They may monitor the power, memory addresses loaded from DRAM, timings of these accesses, etc.

To steal the model architecture, the attacker might probe the memory bus. In section IV we discussed how the network weights can be obfuscated to prevent model theft. Likewise, the network configuration can be stored with the model and obfuscated alongside it. A similar invasive approach may measure which addresses are accessed from DRAM. To hide addresses we may use ORAM, however, counterintuitively, ML models do not necessarily need to hide addresses. Given enough on-chip memory, models are typically stored and accessed in a strictly ascending order. This reveals no information about the network topology or metaparameters except the size of the model.

Memory timing attacks are a simple way of obtaining a model architecture, and are simpler than probing addresses and data. We illustrate the need for obfuscating timing side-channels on a simple example. In Figure 5, we show the timing of DRAM accesses for a VGG19 network running inference

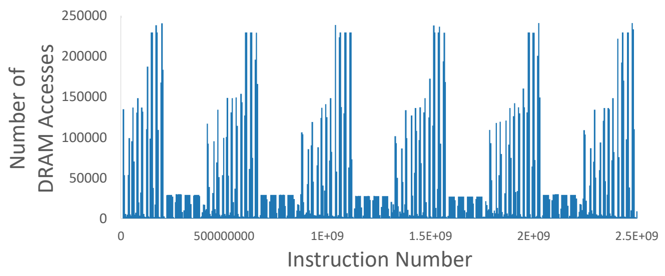

Fig. 5. Memory access timings of six VGG19 network inference operations.

on images from the ImageNet dataset, collected using Intel Pin and a cache simulator. By observing the area of the peaks, an attacker can guess the sizes of the layers. By observing the time between the peaks, the attacker may guess the required compute power for each layer and deduce information about the layer, i.e., the size of convolution kernels, and feature size, etc.

Probing attacks on non-volatile memory can be ignored, as they can only reveal the size of the model, same as with probing DRAM addresses. Other side-channels like power attacks are architecture- and model-dependent, i.e., architectures that have data-dependent power profiles [18] or make use of sparsity are more at risk, and the network designers need to take this into account when creating a secure ML model. Similarly, models with data-dependent inference latencies can reveal information about their internals. Here we provide three approaches for obfuscating memory access timings. Other side-channels will require similar consideration.

On-chip model storage: DNNs are typically orders of magnitude larger than the cache size of most chips. Quantization or sparsity can reduce the model size enough for networks to fit into on-chip memory [19]. On-chip networks still need to be loaded when powering on the device, but this operation does not reveal any information except the network size.

Prefetching: while the device may not have enough on-chip memory to store the whole model, a smaller amount of memory may allow it to prefetch the data required in the future. This allows the device to 'smooth out' memory accesses over time and maintain a constant DRAM throughput. For a model of m bytes which is processed in time t, the device needs to maintain a memory bandwidth of m/t. The buffer then needs to be large enough to store the data accumulated during the periods when the model has a bandwidth of less than m/t.

Artificial memory accesses: a simple way of defending against timing attacks is by creating artificial DRAM accesses [20]. By recording the maximum required memory throughput during a single inference operation, we create fake accesses that maintain that fixed throughput when the application is not accessing enough data. Since the addresses are accessed in an ascending order, the fake addresses need to be present in DRAM. This puts gaps in the stored model, and requires a larger amount of DRAM.

VII. Conclusion

In this paper, we introduced a previously unexplored type of attack - exfiltration of neural network models running on hardware devices. Specifically, we show two new attack vectors: (1) extracting models by probing memory, and (2) learning model architectures using side-channels. We proposed a set of defenses against varying threat levels: memory obfuscation protecting raw memory, and hardware modifications preventing timing side-channels. We discussed how these defenses affect device power and usability.

References

[1] D. Amodei and D. Hernandez, "AI and Compute," https://blog.openai.com/ai-and-compute/, 2018, [Online; accessed 13-July-2018].

[2] F. Tramèr, F. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart, "Stealing machine learning models via prediction apis," *CoRR*, vol. abs/1609.02943, 2016.

[3] N. Carlini, C. Liu, J. Kos, Ú. Erlingsson, and D. Song, "The secret sharer: Measuring unintended neural network memorization & extracting secrets," *CoRR*, vol. abs/1802.08232, 2018.

[4] B. Kolosnjaji, A. Demontis, B. Biggio, D. Maiorca, G. Giacinto, C. Eckert, and F. Roli, "Adversarial malware binaries: Evading deep learning for malware detection in executables," *CoRR*, vol. abs/1803.04173, 2018.

[5] G. Hinton, O. Vinyals, and J. Dean, "Distilling the Knowledge in a Neural Network," 2015.

[6] S. J. Oh, M. Augustin, M. Fritz, and B. Schiele, "Towards reverse-engineering black-box neural networks," in *International Conference on Learning Representations*, 2018.

[7] M. Juuti, S. Szyller, A. Dmitrenko, S. Marchal, and N. Asokan, "PRADA: protecting against DNN model stealing attacks," *CoRR*, vol. abs/1805.02628, 2018.

[8] Y. Adi, C. Baum, M. Cisse, B. Pinkas, and J. Keshet, "Turning your weakness into a strength: Watermarking deep neural networks by backdooring," in *27th USENIX Security Symposium (USENIX Security 18)*. Baltimore, MD: USENIX Association, 2018, pp. 1615–1631.

[9] P. Xie, M. Bilenko, T. Finley, R. Gilad-Bachrach, K. E. Lauter, and M. Naehrig, "Crypto-nets: Neural networks over encrypted data," *CoRR*, vol. abs/1412.6181, 2014.

[10] B. D. Rouhani, M. S. Riazi, and F. Koushanfar, "Deepsecure: Scalable provably-secure deep learning," *CoRR*, vol. abs/1705.08963, 2017.

[11] M. S. Riazi, C. Weinert, O. Tkachenko, E. M. Songhori, T. Schneider, and F. Koushanfar, "Chameleon: A hybrid secure computation framework for machine learning applications," *CoRR*, vol. abs/1801.03239, 2018.

[12] B. Mcdanel, "Distributed Deep Neural Networks over the Cloud , the Edge and End Devices."

[13] L. Bu, C. Hai, and M. A. Kinsy, "Adaptive and dynamic device authentication using lorenz chaotic systems," *61st IEEE International Midwest Symposium on Circuits and Systems*, 2018.

[14] N. Robson, J. Safran, C. Kothandaraman, A. Cestero, X. Chen, R. Rajeevakumar, A. Leslie, D. Moy, T. Kirihata, and S. Iyer, "Electrically programmable fuse (efuse): From memory redundancy to autonomic chips," in *Custom Integrated Circuits Conference, 2007. CICC'07. IEEE*. IEEE, 2007, pp. 799–804.

[15] R. Andri, L. Cavigelli, D. Rossi, and L. Benini, "Yodann: An architecture for ultralow power binary-weight cnn acceleration," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 1, pp. 48–60, 2018.

[16] P. Isola, J. Zhu, T. Zhou, and A. A. Efros, "Image-to-image translation with conditional adversarial networks," *CoRR*, vol. abs/1611.07004, 2016.

[17] E. Stefanov, M. van Dijk, E. Shi, C. Fletcher, L. Ren, X. Yu, and S. Devadas, "Path oram: An extremely simple oblivious ram protocol," in *Proceedings of the 2013 ACM SIGSAC Conference on Computer &#38; Communications Security*, ser. CCS '13. New York, NY, USA: ACM, 2013, pp. 299–310.

[18] J. Albericio, P. Judd, T. Hetherington, T. Aamodt, N. E. Jerger, and A. Moshovos, "Cnvlutin: Ineffectual-neuron-free deep neural network computing," *SIGARCH Comput. Archit. News*, vol. 44, no. 3, pp. 1–13, June 2016.

[19] M. Isakov, A. Ehret, and M. A. Kinsy, "Closnets: Batchless dnn training with on-chip a priori sparse neural topologies," 2018.

[20] M. A. Kinsy, D. Kava, A. Ehret, and M. Mark, "Sphinx: A secure architecture based on binary code diversification and execution obfuscation," *CoRR*, vol. abs/1802.04259, 2018.