When NAS Meets Watermarking: Ownership Verification of DNN Models via Cache Side Channels

Lou Xiaoxuan  
Nanyang Technological University

Guo Shangwei  
Chongqing University

Zhang Tianwei  
Nanyang Technological University

Zhang Yinqian  
Southern University of Science and Technology

Liu Yang  
Nanyang Technological University

Abstract

We present a novel watermarking scheme to verify the ownership of DNN models. Existing solutions embedded watermarks into the model parameters, which were proven to be removable and detectable by an adversary to invalidate the protection. In contrast, we propose to implant watermarks into the model architectures. We design new algorithms based on Neural Architecture Search (NAS) to generate watermarked architectures, which are unique enough to represent the ownership, while maintaining high model usability. We further leverage cache side channels to extract and verify watermarks from the black-box models at inference. Theoretical analysis and extensive evaluations show our scheme has negligible impact on the model performance, and exhibits strong robustness against various model transformations.

1 Introduction

Deep Neural Networks (DNNs) have shown tremendous progress to solve artificial intelligence tasks. Novel DNN algorithms and models were introduced to interpret and understand the open world with higher automation and accuracy, such as image classification [29, 39, 62], natural language processing [11, 16], bioinformatics [59], game playing [61]. With the increased complexity and demand of the tasks, it is more costly to generate a state-of-the-art DNN model: design of the model architecture and algorithm requires human efforts and expertise; training a model with satisfactory performance needs a large amount of computation resources and valuable data samples. Hence, commercialization of the deep learning technology makes DNN models the core Intellectual Property (IP) of AI products and applications.

Release of DNN models can incur illegitimate plagiarism, unauthorized distribution or reproduction. Even a model is packed in a black-box manner, a malicious entity can still steal the model details from the inference results [35, 63, 66, 70] or execution behaviors [7, 33, 68]. Such model extraction attacks highlight the importance of IP protection for DNN models. One of the most common IP protection approaches is DNN watermarking, which processes the protected model in a unique and observable way such that the model owner is able to recognize the ownership of his models. Some solutions were proposed to directly embed the watermarks into the model parameters [56, 64]. Other works [2, 12, 71] fine-tune the model parameters to produce unique output for certain input samples for verification. The embedded watermark needs to guarantee satisfactory performance for the protected model.

Unfortunately, those parameter-based watermarking solutions are not practically robust. An adversary can easily defeat them without any knowledge of the adopted watermarks. First, since these schemes modify the parameters to embed watermarks, the adversary can also modify the parameters of a stolen model to remove the watermarks. Past works have designed such watermark removal attacks, which leverage model fine-tuning [13, 48, 60] or input transformation [27] to successfully invalidate existing watermark methods. Second, watermarked models need to give unique behaviors, which inevitably make them detectable by the adversary. Some works [3, 51] introduced attack techniques to detect the verification samples and then manipulate the verification results.

Motivated by the above limitations, we propose a watermarking scheme, which is fundamentally different from prior works. Instead of crafting the parameters for watermarks, we treat the network architecture as the IP of the model. We aim to design a methodology to generate unique network architectures for the owner to train the protected models, which can serve as the evidence of ownership. In this way, the adversary is not able to tamper with the watermarks by refining the parameters. He has no incentive to modify the watermarked architecture, as this will cause him to train a new model from scratch, which has the same cost as training his own model. Two questions need to be answered in order to establish this scheme: (1) how to systematically design network architectures, that are unique for watermarking and maintain high usability for the tasks? (2) how to extract the architecture of the suspicious model, and verify the ownership?

We introduce a set of techniques to address these questions.
For the first question, we leverage Neural Architecture Search (NAS) [75]. NAS is a very popular approach of Automatic Machine Learning (AutoML), which can automatically discover the optimal network architecture for a given task and dataset. A quantity of methods [10, 17, 46, 47, 53, 76] have been proposed to improve the search effectiveness and efficiency, and the searched architectures can significantly outperform the ones hand-crafted by humans. Inspired by this technology, we design a novel NAS algorithm, which fixes certain connections with specific operations in the search space, determined by the owner-specific watermark. Then we search for the rest connections/operations to produce a high-quality network architecture. This architecture is unique enough to represent the ownership of the model (Section 4).

The second question is solved by cache side-channel analysis. Side-channel attacks are a common strategy to recover confidential information from the victim system without direct access permissions. Recently, some works designed novel attacks to steal DNN models [7, 33, 68]. Our scheme applies side-channel analysis for IP protection, rather than confidentiality breach. The model owner can use side-channel techniques to extract the architecture of a black-box model to verify the ownership, even the model is encrypted or isolated. It is difficult to directly extend prior solutions [32, 68] to our scenario, because they are designed only for conventional DNN models, but fail to recover new operations in NAS. We devise a more comprehensive method to analyze and identify the types and hyper-parameters of these new operations from a side-channel pattern. This enables us to precisely extract the watermark from the target model (Section 5).

The integration of the above techniques leads to the design of our novel watermarking scheme. It includes three stages, as shown in Figure 1. At stage 1, the model owner generates a unique watermark and the corresponding marking key \( mk \). At stage 2, he adopts a conventional NAS method with \( mk \) to produce the watermarked architecture and a verification key \( vk \). He further trains a watermarked model from this architecture. Stage 3 is to verify the ownership of a suspicious model: the owner collects the inference execution trace, and identifies any potential watermark based on \( vk \). Compared to prior watermarking solutions, it is more difficult for the adversary to remove our watermarks, which are embedded into the architecture instead of the parameters. Experiments show common model transformations (fine-tuning, pruning) cannot affect the existence of watermarks in the model.

This paper makes the following contributions:

- It is the first to propose watermarks embedded into model architectures, instead of parameters. It creatively utilizes Neural Architecture Search for watermark embedding.
- It presents the first positive use of cache side channels to extract and verify watermarks.
- It gives a comprehensive side-channel analysis about sophisticated DNN operations that are not analyzed before.

2 Background

2.1 Neural Architecture Search

NAS [21, 75] has gained popularity in recent years, due to its capability of building machine learning pipelines with high efficiency and automation. It systematically searches for the optimal network architecture for a given task and dataset. Its effectiveness is mainly determined by two factors: Search space. This defines the scope of neural networks to be designed and optimized. Instead of searching for the entire network, a practical strategy is to decompose the target neural network into multiple cells, and search for the optimal structure of a cell [76]. Then cells with the identified architecture are stacked in predefined ways to construct the final DNN models. Figure 2a shows the typical architecture of a CNN model based on the popular NAS-Bench-201 [18]. It has two types of cells: a normal cell is used to interpret the features and a reduction cell is used to reduce the spatial size. A block is composed of several normal cells, and connected to a reduction cell alternatively to form the model.
A cell is generally represented as a directed acyclic graph, where each edge is associated with an operation selected from a predefined operation set \([53]\). Figure 2b gives a toy cell supernet that contains four computation nodes (gray squares) and a set of three candidate operations (circles). The solid arrows denote the actual connection edges chosen by the NAS method. Such supernet enables the sharing of network parameters and avoids unnecessary repetitive training for selected architectures. This significantly reduces the cost of performance estimation and accelerates the search process, and is widely adopted in recent NAS methods \([8, 14, 17, 47]\).

**Search strategy.** This defines the approach to seek for the optimal architecture in the search space. Different types of strategies have been designed to enhance the search efficiency and results, based on reinforcement learning \([53, 75, 76]\), evolutionary algorithm \([20, 54]\) or gradient-based optimization \([15, 17, 47]\). Our watermarking scheme is general and independent of the search strategies.

### 2.2 Cache Side Channels

CPU caches are introduced between the CPU cores and main memory to accelerate the memory access. Two micro-architectural features of caches enable an adversarial program to perform side-channel attacks and infer secrets from a victim program, even their logical memory is isolated by the operating system. First, multiple programs can share the same CPU cache, and they have contention on the usage of cache lines. Second, the timing difference between a cache hit (fast) and a cache miss (slow) can reveal the access history of the memory lines. As a result, an adversary can carefully craft interference with the victim program sharing the same cache, and measure the access time to infer the victim’s access trace.

A quantity of techniques have been designed over the past decades to realize cache side-channel attacks. Two representative attacks are PRIME-PROBE and FLUSH-RELOAD. In a PRIME-PROBE attack \([45]\), the adversary first fills up the critical cache sets with its own memory lines. Then the victim executes and potentially evicts the adversary’s data out of the cache. After that, the adversary measures the access time of each memory line loaded previously. A longer access time indicates that the victim used the corresponding cache set. A FLUSH-RELOAD attack \([69]\) requires the adversary to share the critical memory lines with the victim, e.g., via shared library. The adversary first evicts the critical memory lines out of the cache using dedicated instructions (e.g., `clflush`). After a period of time, it reloads these lines into the cache and measures the access time. A shorter time indicates that the memory lines were accessed by the victim.

### 2.3 Threat Model

We consider a model owner searches a network architecture using a NAS method, and then trains a production-level DNN model \(M\). An adversary may obtain an illegal copy of \(M\) and use it for profit without authorization. He may slightly modify the model parameters (e.g., fine-tuning, model compression, transfer learning) to adapt to his scenario and avoid detection. However, it is not feasible for him to change the model architecture (e.g., knowledge distillation \([6, 30]\)), which requires significant cost to retrain the parameters from scratch.

The goal of the model owner is to detect whether a suspicious model \(M'\) is plagiarized from \(M\). He has black-box access to the target model \(M'\), without any knowledge about the model architecture, parameters, training algorithms and hyper-parameters. We further assume the model owner is able to deploy his watermark extraction program on the same physical machine as the target model. Our watermarking scheme can be applied to a couple of scenarios, as described below: **Local scenario.** Trusted Execution Environment (TEE), e.g., Intel SGX \([50]\) and AMD SEV \([36]\), has been widely adopted to protect user-space applications from strong privileged adversaries. It introduces new hardware extensions to provide execution isolation and memory encryption. Researchers also utilized TEE to guard the DNN model privacy and execution integrity \([26, 37, 38, 40, 42, 52]\). However, an adversary can also exploit TEE to hide their malicious activities, such as side-channel attacks \([58]\), rowhammer attacks \([25]\) and malware \([49, 57]\). Similarly, an adversary can hide the stolen model in TEE when distributing it to the public, so the model owner cannot introspect into the DNN model to obtain the evidence of ownership. With our watermarking scheme, the model owner can extract watermarks from the isolated enclaves, as past works have demonstrated successful cache side-channel attacks in TEE processors \([9, 24, 28]\).

**Remote scenario.** This follows the same threat model in \([68]\). An adversary can deploy the plagiarized model in the Machine Learning-as-a-Service (MLaaS) cloud. For instance, Amazon offers the SageMaker \([1]\) service for building machine learning pipelines. Users can upload their DNN models to this platform, which then automatically launches virtual machines in the shared EC2 cloud to host the DNN model and release APIs for inference. To verify the ownership of models using such kind of MLaaS, the model owner can perform co-location attacks to launch a virtual machine on the same server with the DNN model, and then conduct cross-VM side-channel attacks. The feasibility of co-location and cache side-channel attacks in the cloud has been validated in \([4, 5, 22, 55, 65, 67, 73, 74]\).

### 3 Preliminaries

#### 3.1 Definition of A NAS Method

In this paper, we mainly focus on NAS methods using the cell-based search space, as it is the most popular and efficient strategy. Formally, we consider a NAS task, which aims to construct a model architecture containing \(N\) cells:
$\mathcal{A} = \{c_1, \ldots, c_N\}$. The search space of each cell $c_i$ is denoted as $S_i = (B, E, O) : B$ is the number of computation nodes in the cell; $E = \{E_1, \ldots, E_B\}$ is the set of all possible edges between nodes and $E_j$ is the set of edges connected to the $j$-th node ($1 \leq j \leq B$); $O$ is the set of candidate operations on these edges. Hence, the supernet of a cell can be represented as a matrix $M^c$ whose row size is the size of $E$ and column size is the size of $O$. Each element at position $(x, y)$ can be 1 if edge $x$ is attached with operation $y$, or 0 otherwise. Then we combine the search spaces of all cells as $\mathcal{S}$, from which we look for an optimal architecture $\mathcal{A}$. The NAS method we consider is defined as below:

**Definition 3.1. (NAS)** A NAS method is a machine learning algorithm that iteratively searches optimal cell architectures from the search space $\mathcal{S}$ on the proxy dataset $\mathcal{D}$. These cells construct one architecture $\mathcal{A} = \{c_1, \ldots, c_N\}$, i.e.,

$\mathcal{A} = \text{NAS}(\mathcal{S}, \mathcal{D})$.

After the search process, $\mathcal{A}$ is trained from the scratch on the task dataset $\mathcal{D}$ to learn the optimal parameters. The architecture $\mathcal{A}$ and the corresponding parameters give the final DNN model $f = \text{train}(\mathcal{A}, \mathcal{D})$.

### 3.2 Definition of A Watermarking Scheme

A watermarking scheme for NAS enables the ownership verification of DNN models searched from a NAS method. This is formally defined as below:

**Definition 3.2.** A watermarking scheme for NAS is defined as a tuple of probabilistic polynomial time algorithms ($\text{WMGen}$, $\text{Mark}$, $\text{Verify}$), where

- **WMGen**: this function takes the search space of a NAS method as input and outputs a secret marking key $mk$.
- **Mark**: given a NAS method, a proxy dataset $\mathcal{D}$, and $mk$, this function outputs a watermarked DNN model $f$ with the corresponding verification key $vk$.
- **Verify**: this function takes the input of $vk$ and the monitored side-channel trace, and outputs the verification result of the watermark in $[0, 1]$.

A strong watermarking scheme for NAS should have the following properties.

**Effectiveness.** The watermarking scheme needs to guarantee the success of the ownership verification over the watermarked $f$ using the verification key. Formally,

$$\Pr[\text{Verify}(vk, T) = 1] = 1,$$

where $T$ is the monitored side-channel trace from $f$.

**Usability.** Let $f_0$ be the original DNN model without being watermarked. For any data distribution $\mathcal{D}$, the watermarked model $f$ should exhibit competitive performance compared with $f_0$ on the data sampled from $\mathcal{D}$, i.e.,

$$\Pr[f_0(x) = y | (x, y) \sim \mathcal{D}] - \Pr[f(x) = y | (x, y) \sim \mathcal{D}] \leq \varepsilon.$$  (2)

**Robustness.** Since a probabilistic polynomial time adversary may modify $f$ with common transformations, we expect the watermark remains in the model after those changes. Formally, let $T'$ be the side-channel leakage of a model $f'$ transformed from $f$, where $f'$ and $f$ have similar performance. We have

$$\Pr[\text{Verify}(vk, T') = 1] \geq 1 - \delta$$  (3)

**Uniqueness.** A normal user can follow the same NAS method to learn a model from the same proxy dataset. Without the marking key, the probability that this common model contains the same watermark should be smaller than a given threshold $\delta$. Let $T'$ be the side channel leakage of a common model learned with the same dataset and NAS method, we have

$$\Pr[\text{Verify}(vk, T') = 1] \leq \delta.$$  (4)

### 4 Our Watermarking Scheme

Our solution embeds watermarks into the neural architectures, and uses cache side channels to verify the ownership. Figure 1 shows the overview of our scheme, which includes watermark generation (Section 4.1), embedding (Section 4.2) and verification (Section 4.3). We introduce a novel algorithm for each stage, followed by a theoretical analysis (Section 4.4).

#### 4.1 Watermark Generation

A watermark for a NAS model consists of multiple stamps. For a NAS cell, each element in its supernet $M^c$ can be 1 or 0, identified by the search strategy. We select and fix certain elements in $M^c$ to be 1 during the search process. The set of the fixed edge-operation pairs inside a cell is called a stamp. It is formally defined as below:

**Definition 4.1. (Stamp)** A stamp for a cell is a set of edge-operation pairs $\{s_e : s_o\}$, where $s_e$, $s_o$ are the set of the selected edges and the corresponding operations, respectively.

According to Definition 3.1, a NAS model is a composition of cells. Thus, the combination of the stamps of all cells forms a watermark for the model, as defined below:

**Definition 4.2. (Watermark)** Consider a NAS method with a proxy dataset $\mathcal{D}$ and search space $\mathcal{S}$, $\mathcal{A} = \{c_1, \ldots, c_N\}$ represents the neural architecture produced from this method. A watermark for $\mathcal{A}$ is a set of stamps $k_1, \ldots, k_N$, where $k_i$ is the stamp of cell $c_i$.

We introduce a marking key $mk$ as the representation of a watermark. Algorithm 1 illustrates the detailed procedure of constructing a watermark and the corresponding $mk$. For each cell $c_i$, we randomly select $n_e$ edges from $E$. Since NAS normally requires each node $f$ has maximal two inputs from previous nodes, at most two elements can be selected in $E_j$. For each selected edge, we attach a random operation to it.
4.2 Watermark Embedding

To generate a competitive DNN model embedded with the watermark, we apply a NAS method with the marking key $mk$. Due to the fixed connections and operations, this process will have a smaller search space. Experimental evaluations in Section 6 indicate that with an appropriate value of $n_o$, the reduced search space incurs negligible impact on the model performance, which conforms to the conclusion in [47, 76].

Algorithm 2 shows the procedure of embedding the watermark to a NAS model. For each cell $c_i$ in the model, we first identify the fixed stamp edges and operations $\{s_e : s_o\}$ from key $k_i$. Then the cell search space $S_i$ is updated as $(B, E, O)$, where $E$ is the set of connection edges excluding those fixed ones: $E = E - s_e$. The search spaces of all the cells are combined to form the search space $S$ of the model, from which the NAS method is used to find the optimal architecture $\mathfrak{A}$. Finally, the identified $\mathfrak{A}$ is trained on the task dataset $\mathcal{D}$ to produce the watermarked model $f$.

It is worth noting that the marking key $mk$ only describes the abstract information of the watermark embedded in the model architecture $\mathfrak{A}$, which is insufficient for verifying the details of the watermark from the suspicious model. Hence, we introduce a verification key $vk$ for ownership identification. $vk$ is a superset of the marking key $mk$, including some hyper-parameters of the model $f$ (input size and channels). Besides, during the watermark extraction, the model owner can only identify the sequence of operations with distinct patterns from the side-channel trace. The edge information $s_e$ in $mk$ cannot explicitly disclose their positions in the leakage trace. So we replace $s_e$ with the relative orders $s_{id}$ of their operations $s_o$, which help us better describe the watermark.

Algorithm 1: Marking Key Generation (WMGen)

| Input: # of fixed edges $n_z$, search space $S$ |
|-----------------------------------------------|
| Output: marking key $mk$ |
| 1 for $i$ from 1 to $N$ do |
| 2 $s_e$ ← randomly select $n_z$ edges from $E$ such that $|s_e \cap E_i| \leq 2, \forall j \in [1, B]$ |
| 3 $s_o$ ← randomly select $n_o$ operations from $O$ for $s_e$ |
| 4 $k_i = [s_e : s_o]$ |
| 5 return $mk = (k_1, ..., k_N)$ |

To summarize, the marking key $mk$ is used to represent a class of models that are constructed by the NAS cells with the same watermark, while the verification key $vk$ is used to identify the models retrained for specific tasks. A model owner can use just one marking key to watermark different models regardless of the tasks, datasets or NAS methods, and then use the corresponding verification keys for ownership identification.

4.3 Watermark Verification

During verification, we utilize cache side channels to capture an execution trace $T$ by monitoring the inference process of the target model $f$. Details about side-channel extraction can be found in Section 5. Due to the existence of pre-processing and post-processing operations, the latency between two cells is much longer and human-noticeable. So we can easily divide this trace into sequential windows, with each one representing the pattern of a NAS cell. If $T$ does not have observable windows, we claim it is not generated by a NAS method. A leakage window further contains multiple clusters, each of which corresponds to an operation inside the cell.

Algorithm 3 describes the process of watermark verification. First the total number of stamp edges $N_v$ is computed from the verification key $vk$. We introduce a variable $N_v$ to denote the number of matched edge-operation pairs, initialized as 0. Then we analyze the cluster patterns from the trace $T$ in sequence. Specifically, for the $i$-th leakage window, we first retrieve its stamp from $k_i$ in $vk$, which contains the computation orders $s_{id}$ of stamp edges and the attached operations $s_o$. Then we use the methodology in Section 5 to analyze each cluster whose order is contained in $s_{id}$, as it corresponds to a possible stamp edge. If the recovered operation of the cluster matches the corresponding stamp operation in $s_o$, $N_v$ is added by 1. After scanning all the clusters in all the cells, we calculate the ratio of edge-operation pairs in the target model matched with the watermark as $N_v/N_v$. We claim the ownership of this model if the ratio is higher than a designated threshold $\tau$. 
We theoretically prove our scheme can satisfy the properties of the watermarking scheme for NAS models. The proof can be summarized as follows.

**Theorem 1.** With Assumptions 1-2, the proposed Algorithms 1-3 form a watermarking scheme that satisfies the properties of effectiveness, usability, robustness, and uniqueness.

### 4.4 Theoretical Analysis

We theoretically prove our scheme can satisfy the properties in Section 3.2. We first assume the search space restricted by the watermark is still large enough for the model owner to find a qualified architecture.

**Assumption 1.** Let $S_0$ be the search spaces before and after restricting a watermark in a NAS method, where $S_0 \supseteq S$. $A_0 \in S_0$ is the optimal architecture for an arbitrary data distribution $D$. $A$ is the optimal architecture in $S$. The model accuracy of $A$ is no smaller that that of $A_0$ by a relaxation of $\frac{\tau}{6}$.

We further assume the existence of an ideal analyzer that can recover the watermark from the given side-channel trace.

**Assumption 2.** Let $m_k$ and $v_k$ be the marking and verification key of a DNN architecture $A = \{c_1, \ldots, c_N\}$. For $m_k$ and $v_k$, and $A$, there is a leakage analyzer $P$ that is capable of recovering all the stamps of $\{c_i\}_{i=1}^N$ from a corresponding cache side-channel trace.

With the above two assumptions, we prove that our proposed algorithms (WMGen, Mark, Verify) form a qualified watermarking scheme for NAS models. The proof can be found in Appendix A.

**Theorem 1.** With Assumptions 1-2, the proposed Algorithms 1-3 form a watermarking scheme that satisfies the properties of effectiveness, usability, robustness, and uniqueness.

### 5 Side Channel Extraction

Given a suspicious model, we aim to extract the embedded watermark using cache side channels. Past works have proposed cache side channel attacks to steal DNN models [32, 68]. However, these attacks are only designed for conventional DNN models and cannot extract NAS models with more sophisticated operations (e.g., separable convolutions, dilated-separable convolutions). Besides, the adversary needs to have the knowledge of the target model’s architecture family (i.e., the type of each layer), which cannot be obtained in our case.

We design an improved methodology over the Cache Telepathy [68] to extract the architecture of NAS models by monitoring the side-channel pattern from the BLAS library\(^1\). In this paper, we take OpenBLAS as an example, which is a mainstream library for many deep learning frameworks (e.g., Tensorflow, PyTorch). Our method is also generalized to other BLAS libraries, such as Intel MKL. We make detailed analysis about the leakage pattern of common operations used in NAS, and describe how to identify the operation type and hyper-parameters.

#### 5.1 Method Overview

State-of-the-art NAS algorithms [18, 46, 47, 76] commonly adopt eight classes of operations: (1) identity, (2) fully connected layer, (3) normal convolution, (4) dilated convolution, (5) separable convolution, (6) dilated-separable convolution, (7) pooling and (8) various activation function. Note that although zeroize is also a common operation in NAS, we do not consider it, as it just indicates a lack of connection between two nodes and is not actually used in the search process.

These operations are commonly implemented in two steps. (1) The high-level deep learning framework converts an operation to a matrix multiplication: $C = \alpha A \times B + \beta C$, where input $A$ is an $m \times k$ matrix and $B$ is a $k \times n$ matrix, output $C$ is an $m \times n$ matrix, and both $\alpha$ and $\beta$ are scalars; (2) The low-level BLAS library performs the matrix multiplication with the GEMM algorithm (Algorithm 4). Constants of $P, Q, R$ and UNROLL are determined by the host machine configuration. Our testbed (Section 6.1) adopts $P = 320, Q = 320, R = 104512$ and $UNROLL = 4$. As $R$ is generally larger than $n$ in NAS models, we assume $loop_1$ is performed only once. More details about GEMM can be found in Appendix B.

We take three steps to recover each operation and its hyper-parameters. First, we monitor the memory accesses to the icopy and oncopy functions in Algorithm 4, and count the

\(^1\) For RNN models, we monitor the high-level deep learning framework, as the BLAS library does not leak information about the model.
number of iterations \( \text{iter}_n \) for each loop \( n \). Figure 3 illustrates the leakage patterns of four representative operations with a sampling interval of 2000 CPU cycles. Different operations have distinct patterns of side-channel leakage. By observing such patterns, we can identify the type of the operation.

Second, we utilize the technique in [68] to derive the range of the matrix dimension \( (m,n,k) \) from \( \text{iter}_n \), based on the equations: \( \text{iter}_1 = 1 \), \( \text{iter}_2 = \lfloor k/Q \rfloor \), \( \text{iter}_3 = \lceil (m-P)/P \rceil \) and \( \text{iter}_4 = \lceil n/3\text{UNROLL} \rceil \). Note that the final two iterations of each loop are actually assigned with two equal-size blocks, rather than blocks of size \( m \) (or \( n,k \)). This does not make big differences on the derivation. Then we deduce the possible values of matrix dimension from the range, based on the constraints of NAS models.

Third, we derive the hyper-parameters of each operation based on the matrix dimension. The relationships between the hyper-parameters of various operations and the dimensions of the transformed matrices are summarized in Table 1. We present both the general calculations of the hyper-parameters as well as the ones specifically for NAS models.

Below we give detailed descriptions of the above three-step analysis for each operation.

### 5.2 Recovery of NAS Operations

**Fully connected (FC) layer.** This operation can be transformed to the multiplication of a learnable weight matrix \( \Theta \) \((m \times k)\) and an input matrix \( \text{in} \) \((k \times n)\), to generate the output matrix \( \text{out} \) \((m \times n)\). \( m \) denotes the number of neurons in the layer; \( k \) denotes the size of the input vector; and \( n \) reveals the batch size of the input vectors. Hence, with the possible values of \((m,n,k)\) derived from the iteration counts of \( \text{icopy} \) and \( \text{oncopy} \), hyper-parameters (e.g., neurons number, input size) of the FC layer can be recovered. The number of FC layers in the model can also be recovered by counting the number of matrix multiplications. Figure 3(a) shows the leakage pattern of a classifier with two FC layers, where the first layer has 1024 neurons and the second layer has 100 neurons. From this pattern, it is easy to identify these two FC layers. The first layer takes as input a batch of 12 vectors of size 512, and it has \( m=1024, n=12, k=512 \). We can infer the range of \((m,n,k)\) based on the number of iterations in each loop: \( \text{iter}_2 = 2 \) (i.e.,

![Figure 3: Side-channel patterns of four operations in NAS.](image)

![Figure 4: Implementing a convolution as matrix multiplication](image)
shows the leakage pattern of a normal convolution \((H = W_i = 32, D_i = 33, R_i = 3, P_i = 2 \text{ and } D_{i+1} = 132)\), which has \(\text{iter}_2 = 1, \text{iter}_4 = 11 \text{ and } \text{iter}_3 = 3\). Note that the first two red nodes (interval 409 and 410) can be treated as one iteration, as they occur in a very short period and generated by side-channel noise. In the NAS scenario, since the normal convolution is generally used at the preprocessing stage, while the FC layer is adopted as the classifier at the end, they can be distinguished based on their locations.

**Dilated convolution.** This operation is a variant of the normal convolution, which inflates the kernel by inserting spaces between each kernel element. We use the dilated space \(d\) to denote the number of spaces inserted between two adjacent elements in the kernel. The conversion from the hyper-parameters of a dilated convolution to the matrix dimension is similar with the normal convolution. The only difference is the row size \(m\) of the input matrix \(in_i\), i.e., the number of patches. Due to the inserted spaces in the kernel, although the kernel size is still \(R^2_i\), the actual size covered by the dilated kernel becomes \(R^2_i + 1\), where \(R^2_i = R_i \times d\). This changes the number of patches to \((W_i - R^2_i + P_i + 1)(H_i - R^2_i + P_i + 1)\). As a dilated convolution is normally implemented as a dilated separable convolution in practical NAS methods [17, 47], the leakage pattern of the operation will be discussed with the dilated separable convolution.

**Separable convolution.** According to [68], the number of consecutive matrix multiplications with the same pattern reveals the batch number of a normal convolution. However, we find this does not hold in the separable convolution, or precisely, the depth-wise separable convolution used in NAS. This is because the separable convolution decomposes a convolution into multiple separate operations, which can incur the same conclusion that the number of the same patterns equals to the number of input channels.

A separable convolution aims to achieve more efficient computation with less complexity by separating the filters. Figure 5 shows a two-step procedure of a separable convolution. The first step uses \(D_i\) filters (Filters 1) to transform the input to an intermediate tensor, where each filter only convolves one input channel to generate one output channel. It can be regarded as \(D_i\) normal convolutions, with the input channel size of 1 and the filter size of \(R^2_i \times 1\). These computations are further transformed to \(D_i\) consecutive matrix multiplications with the same pattern, which is similar as a normal convolution with the batch size of \(D_i\). But the separable convolution has much shorter intervals between two matrix multiplications, as they are parts of the whole convolution, rather than independent operations. In the second step, a normal convolution with \(D_{i+1}\) filters is applied to the intermediate tensor to generate the final output.

In summary, the leakage pattern of the separable convolution is fairly distinguishable, which contains \(D_i\) consecutive clusters and one individual cluster at the end. Note that in a NAS model, the separable convolution is always applied twice [17, 44, 47, 54, 76] to improve the performance, which makes its leakage pattern more recognizable. Figure 3(c) shows the trace of a separable convolution \((H = W_i = 32, D_i = 12, R_i = 3 \text{ and } D_{i+1} = 33)\). There are clearly two parts following the same pattern, corresponding to the two occurrences of the operation. Each part contains 12 consecutive same-pattern clusters to reveal \(D_i = 12\), and an individual cluster denoting the last \(1 \times 1\) convolution.

**Dilated separable (DS) convolution.** This operation is the practical implementation of a dilated convolution in NAS. The DS convolution only introduces a new variable, the dilated space \(d\), from the separable convolution. Hence, this operation has similar matrix transformation and leakage pattern as the separable convolution, except for two differences. First, \(R_i\) is changed to \(R^2_i + 1\) in calculating the number of patches \(m = (W_i - R^2_i + P_i + 1)(H_i - R^2_i + P_i + 1)\) in Step One. Second, a DS convolution needs much shorter execution time. Figure 3(d) shows the leakage pattern of a DS convolution with the same hyper-parameters as a separable convolution depicted in Figure 3(c), except that the dilated space \(d = 1\). It is easy to see the performance advantage of the DS convolution (8400 intervals) over the separable convolution (10000 intervals) under the same configurations. The reason is that the input matrix in a DS convolution contains.

### Table 1: Mapping between operation hyper-parameters and matrix dimensions.

| Operations | Parameters | Value |
|------------|------------|-------|
| Fully Connected | \(C_i\): # of layers | \# of matrix muls |
| | \(C_o\): # of neurons | \(row(0)) |
| Operations | \(D_{i+1}\): Number of Filters | \(R_i\): Kernel Size | \(P_i\): Padding | Stride | \(d\): Dilated Space |
| Normal Conv | \(\mathit{col}(F)\) | \sqrt{\mathit{col}(m)} | \(\mathit{diff}((\mathit{row}(\mathit{in}_i)), \mathit{row}(\mathit{out}_{i+1}))\) | \(\mathit{NAS} \cdot R_i - 1\) (non-dilated) | \(\sqrt{\mathit{row}(\mathit{m})}\) | 0 |
| Dilated Conv | \(\mathit{col}(F)\) | \(\sqrt{\mathit{col}(m)}\) | \(\mathit{NAS} \cdot R_i - 1\) (dilated), where \(R_i' = R_i + d(R_i - 1)\) | \(\sqrt{\mathit{row}(\mathit{m})}\) | \(\mathit{NAS} = 1\) (normal cells) | \(\mathit{d}\) |
| Separable Conv | Filters \(1\): # of same matrix muls | Filters \(2\): \sqrt{\mathit{row}(\mathit{F})}\) | Filters \(3\): \(\mathit{col}(F)\) | Filters \(2\): \(\mathit{col}(F)\) | Filters \(2\): \(\mathit{col}(F)\) |
| Dil-Sep Conv | \(\mathit{row}(\mathit{F})\) | \(\mathit{row}(\mathit{F})\) | \(\mathit{row}(\mathit{F})\) | \(\mathit{row}(\mathit{F})\) | \(\mathit{row}(\mathit{F})\) | \(\mathit{row}(\mathit{F})\) |

**Figure 5:** Procedure of separable convolutions.
more padding zeroes to reduce the computation complexity. Besides, the DS convolution does not need to be performed twice, which also helps us distinguish it from a separable one.

**Skip connect.** The operation is also called *identity* in the NAS search space, which just sends \( \text{out} \) to \( \text{in} \) without any processing. This operation cannot be directly detected from the side-channel leakage trace, as it does not invoke any GEMM computations. While [68] argues the skip can be identified as it causes a longer latency due to the introduction of an extra merge operation, it is not feasible in a NAS model. This is because in a cell, each node has an add operation of two inputs and the skip operation does not invoke any extra operations. So there is no obvious difference between the latency of skip and the normal inter-GEMM intervals. Our experiments show that while the skip connect cannot be distinguished in a CNN model, it can still be identified in an RNN model. More details can be found in Section 6.

**Pooling.** We assume the width and height of the pooling operation is the same, which is default in all practical implementations. Given that pooling can reduce the size of the input matrix \( \text{in} \), from the last output matrix \( \text{out}_{-1} \), the size of the pooling layer can be obtained by performing square root over the quotient of the number of rows in \( \text{out}_{-1} \) and \( \text{in} \). In general, pooling and non-unit striding cannot be distinguished as they both reduce the matrix size. However, in a NAS model, non-unit striding is only used in reduction cells which can double the channels. This information can be used for identification. Pooling cannot be directly detected as it does not invoke any matrix multiplications in GEMM. But it can introduce much longer latency (nearly \( 1.5 \times \) of the normal inter-GEMM latency) for other computations. Hence, we can identify this operation by monitoring the matrix size and execution intervals. While monitoring the BLAS library can only tell the existence of the pooling operation, the type can be revealed by monitoring the corresponding pooling functions in the deep learning framework.

**Activation function.** An activation function is normally attached with each convolution operation. Different from CNN models, an RNN model searched by NAS only consists of activation functions, e.g., \( \text{relu}, \text{sigmod}, \text{tanh} \). As they do not perform any complex matrix multiplications, their footprints cannot be found in the low-level BLAS library. Hence, we turn to monitor the deep learning framework for identification.

### 6 Evaluation

#### 6.1 Experimental Setup

**Testbed.** Our approach is general for different types of deep learning frameworks and libraries. Without loss of generality, we adopt the latest version of Pytorch (1.7.0) and OpenBLAS (0.3.13), deployed in Ubuntu 18.04 with a kernel version of 4.15.0. Evaluations are performed on a workstation of Dell Precision T5810 (6-core Intel Xeon E5 processor, 32GB DDR4 memory). The processor has core-private 32KB L1 caches, 256KB L2 caches and a shared 15MB last level cache.

**NAS implementation.** Our scheme is independent of the search strategy, and can be applied to all cell-based NAS methods. We mainly focus on the CNN tasks, and select a state-of-the-art NAS method GDAS [17], which can produce qualified network designs within five GPU hours. We follow the default configurations to perform NAS [17, 76]: the search space of a CNN cell contains eight candidate operations: identity, zeroize, \( 3 \times 3 \) and \( 5 \times 5 \) separable convolutions, \( 3 \times 3 \) and \( 5 \times 5 \) dilated separable convolutions, \( 3 \times 3 \) average pooling, \( 3 \times 3 \) max pooling. The discovered cells are then stacked to construct DNN models. We adopt CIFAR10 as the proxy dataset to search the architecture, and train CNN models over different datasets, e.g., CIFAR10, CIFAR100, ImageNet. Technical details about cell search and model training can be found in Appendix C. We also consider watermarking RNN models (Section 6.6). We choose the DARTS [47] method, which can generate models within six GPU hours. The search space of a RNN cell contains the operations of \( \text{tanh}, \text{relu}, \text{sigmoid} \) activations, identity and zeroize. We use the PTB dataset to search and train RNN models.

**Side channel extraction.** For CNN models, we monitor the \( \text{icopy} \) and \( \text{oncopy} \) functions in OpenBLAS. For RNN models, we monitor the activation functions (\( \text{tanh}, \text{relu} \) and \( \text{sigmoid} \)) in Pytorch, since executions in OpenBLAS do not leak information about the models. We adopt the \( \text{FLUSH+RELOAD} \) side-channel technique, but other methods can achieve our goal as well. We inspect the cache lines storing these functions at a granularity of 2000 CPU cycles to obtain accurate information. Details about the monitored code locations can be found in Table 4 in Appendix D.

#### 6.2 Effectiveness

##### 6.2.1 Key Generation

A NAS method generally considers two types of cells. So we set the same stamp for each type. Then the marking key can be denoted as \( mk = (k_s, k_r) \), where \( k_s = \{s_{en} : s_{on} \} \) and \( k_r = \{s_{er} : s_{or} \} \) represent the stamps embedded to the normal cells and reduction cells, respectively. Each cell has four computation nodes (\( B = 4 \)), so the number of stamp edges satisfies \( 0 \leq n_s \leq 8 \). We choose \( n_s = 4 \) for both of the two cells, indicating four connection edges in each cell are randomly fixed and attached with random operations. We follow Algorithm 1 to generate one example of \( mk \), as shown in Table 2.

| \( k_s \) | \( s_{en} \) | \( s_{on} \) | \( s_{er} \) | \( s_{or} \) |
|--------|-----------|-----------|-----------|-----------|
| \( k_r \) | \( c_{1, 1} \rightarrow \text{B1} \) | \( \text{B2} \rightarrow \text{B3} \) | \( c_{2, 2} \rightarrow \text{B3} \) | \( \text{B2} \rightarrow \text{B3} \) |
| \( s_{en} \) | \( \text{sep} \) | \( \text{avg_pool} \) | \( \text{dil_conv} \) | \( \text{dil_conv} \) |
| \( s_{on} \) | \( \text{dil_conv} \) | \( \text{sep} \) | \( \text{avg_pool} \) | \( \text{sep} \) |

Table 2: Values of \( mk \). \( \text{sep}_\text{conv}(r) \) is a \( r \times r \) separate convolution; \( \text{dil}_\text{conv}(r) \) is a \( r \times r \) dilated separate convolution.
6.2.2 Watermark Embedding

We follow Algorithm 2 to embed the watermark determined by \( mk \) to the DNN model during the search process. Figure 6 shows the architectures of two cells searched by GDAS, where stamps are marked as red edges, and the relative order of each operation is annotated with a number. These two cells are further stacked to construct a complete DNN model (Figure 2a), including three normal blocks (each contains six normal cells) connected by two reduction cells. The pre-processing layer is a normal convolution that extends the number of channels from 3 to 33. The number of filters is doubled in the reduction cells, so the channel sizes (i.e., filter number) of the three normal blocks are 33, 66 and 132. We train the searched architecture over CIFAR10 for 300 epochs to achieve a 3.52% error rate on the validation dataset. This is just slightly higher than the baseline (3.32%), where all connections participate in the search process. This shows the usability of our watermarking scheme. Finally, we generate the verification key \( vK \) from the computation orders \( s_{id} \) and hyper-parameters of the stamp operations.

6.2.3 Watermark Extraction and Verification

Given a suspicious model, we launch a spy process to monitor the function activities in OpenBLAS during inference, and collect the side-channel trace. We conduct the following steps to analyze this trace.

First, we check whether the pattern of the whole trace matches the macro-architecture of a NAS model, i.e., the trace has three blocks, each of which contains six similar leakage windows, and divided by two different leakage windows. An example of a leakage trace can be found in Appendix E.

Second, we focus on the internal structure of each cell. Here we only demonstrate the pattern of the first leakage window (i.e., the first normal cell) as an example (Figure 7). Other cells can be analyzed in the same way. From this figure, we can observe four large clusters, which can be easily identified as four (dilated-)separable convolutions. Figure 8a shows the measured execution time of the four GEMM operations. Clusters \( \{2, 6\} \) and \( \{7\} \) are separable convolutions with 33 input channels, since they all contain \( D = 33 \) consecutive sub-clusters\(^2\). The shorter cluster \( \{4\} \) denotes a dilated separable convolution, based on the patterns profiled in Section 5.2. The three small clusters at the beginning of the trace are identified as three normal convolutions used for preprocessing the input.

Third, we further identify other non-convolution operations from the leakage trace. Figure 8b shows the inter-GEMM latency for different clusters in the cell. We observe the latency of \( \{5\} \) and \( \{8\} \) is much larger, indicating they are pooling operations. Particularly, the latency of \( \{8\} \) contains two parts: pooling and interval between two cells. The other three inter-GEMM latency have similar values, even two of them actually contain a \( skip \) operation. This confirms our conclusion about the \( skip connect \) in Section 5.2, and a simple way to address it is ignoring the \( skip \) when constructing the verification key. We can modify the \( s_{id} \) from \( \{3, 4, 5, 7\} \) to \( \{2, 3, 5\} \) to remove two \( skip \) operations, which can still lead to correct verification results. The analysis of the reduction cell is similar.

The above analysis discloses the types of stamp operations, channel size (number of filters) and stride size (cell type), which can give fair verification results. To be more confident, we further recover the remaining hyper-parameters (in particular, the kernel size) based on their matrix dimensions \((m, n, k)\),

\(^2\)The value of \( D \) can be identified if we zoom in Figure 7, which is not shown in this paper due to page limit.
Figure 8: Execution time of the operations in a cell

Figure 9: Extracted values of the matrix parameters \((m, n, k)\).

Figure 10: Top-1 validation accuracy on CIFAR10 (top) and CIFAR100 (bottom) for watermarked models.

6.3 Usability

This property requires the watermarked model has competitive performance with the original one. To evaluate this property, we vary the number of stamp edges \(n_s\) from 1 to 8 to search watermarked architectures. Then we train the models over CIFAR10 and CIFAR100, and measure the top-1 validation accuracy. Figure 10 shows the average results over five experiments versus the training epochs.

We observe that models with different stamp sizes have quite distinct performance at epoch 50. Then they gradually converge along with the training process, and finally reach a similar accuracy at epoch 300. For CIFAR10, the accuracy of the original model is 96.53%, while the watermarked model with the worst performance \((n_s = 7)\) gives an accuracy of 96.09%. Similarly for CIFAR100, the baseline accuracy and worst accuracy \((n_s = 3)\) are 81.07% and 80.05%. We also check this property on ImageNet. Since training an ImageNet model is quite time-consuming (about 12 GPU days), we only measure the accuracies of the baseline model and two watermarked models \((n_s = 4 \text{ and } 8)\), which are also roughly the same (73.97%, 72.73% and 72.51%). This confirms our watermarking scheme does not affect the usability of the model.

6.4 Robustness

A watermarked model should be robust against any model transformations that aim to remove the watermarks while maintaining the model usability. Prior solutions watermark the parameters of the target model, which are proven to be vulnerable against model fine-tuning or image transformations [13, 27]. In contrast, our scheme modifies the network architecture instead of parameters. So it is robust against common model transformations.

First, we consider four types of model fine-tuning evaluated in [2], including Fine-Tune Last Layer (FTLL), Fine-Tune All Layers (FTAL), Re-Train Last Layer (RTLL), Re-Train All Layers (RTAL). These transformations only focus on tun-
Given a watermarked model, we expect that benign users have very low probability to obtain the same architecture following the original NAS method. This is to guarantee small false positives of watermark verification.

To evaluate this property, we repeat the GDAS method on CIFAR10 for 100 times with different random seeds to generate 100 architecture pairs for the normal and reduction cells. We find our stamps have no collision with these 100 normal models. Specifically, Figure 12 shows the distribution of the operations on eight connection edges in the two cells. We observe that most edges have some preferable operations, and there are some operations never attached to certain edges. This is more obvious in the architecture of the reduction cell. So if a stamp selects such operation-edge pairs, the probability of model collision is even lower than random selection. Besides, the collision probability is decreased when the stamp size \( n_s \) is larger. A stamp size of 4 with random edge-operation selection can already achieve strong uniqueness.

### 6.6 Watermarking RNN Models

In addition to CNN models, our scheme can watermark RNN models as well. A NAS RNN model is commonly stacked by a series of recurrent cells with a searched architecture. In a recurrent cell, each computation node only takes one input from the previous nodes, which is processed by one function in the candidate operation set. Then all the intermediate nodes are averaged to generate the cell output \( h_i \). Figure 13 shows an example of a recurrent cell searched on the PTB dataset with DARTS. Two inputs (the input data \( x_i \) and hidden state of the last layer \( h_{i-1} \)) are added and passed to a \( 	ext{tanh} \) function for the initial node \( n_0 \) [47, 53].

Our scheme randomly selects \( n_s = 4 \) edge-operation pairs as the stamp (red lines in Figure 13). Since the search space for a recurrent cell only contains activation functions, which cannot be observed from the trace of GEMM computations, we monitor the function activities in the PyTorch framework instead. Figure 14 shows the leakage trace of a NAS RNN model. We can observe a recurrent cell contains 9 separate clusters: the first one denotes the \( 	ext{tanh} \) operation at the cell input and the remaining eight clusters are the operations attached to the computation nodes. The input to each node is processed by the \( 	ext{sigmoid} \) functions, followed by the searched NAS operations. Compared with the leakage trace of a CNN model, a RNN model has a much simpler and observable
We observe the watermark can increase the perplexity of the watermarks into model parameters directly could affect their performance (e.g., the fifth cluster in Figure 14). The embedded watermark has relatively larger impact on the performance of the searched RNN model, as the recurrent cell is much simpler and more sensitive from the initial state. Table 3 shows the perplexity of the watermarked models over the validation and test datasets, with different stamp sizes $n_s$. We observe the watermark can increase the perplexity of the RNN model. However, such performance is still satisfactory compared to the original model. Rouhan et al. [56] found that implanting a distinguishable blank area that shortens the cluster length operates the RNN leakage trace than the CNN trace, as it results in a distinguishable blank area that shortens the cluster length (e.g., the fifth cluster in Figure 14).

Figure 14: The side-channel trace of a recurrent cell.

Table 3: Perplexity of watermarked RNN models with various $n_s$.

| Perplexity | Original | Stamp Size $n_s$ |
|------------|----------|-----------------|
|            | 60.33    | 60.27           |
| Valid      | 59.71    | 59.99           |
| Test       | 60.27    | 60.03           |
|            | 62.23    | 62.93           |

We showed a carefully-crafted network architecture can be utilized as the ownership evidence, which exhibits stronger resilience against model transformations than previous parameter-based watermarks. Namba et al. [51] and Li et al. [43] generated watermark samples that are almost indistinguishable from normal samples to avoid detection by adversaries. Different from the above works, this paper proposes a totally new watermarking scheme. Instead of modifying the parameters, our approach makes the architecture design as Intellectual Property, and adopts cache side channels for architecture verification. This strategy can defeat all the watermark removal attacks via parameter transformations.

### 7.2 DNN Model Extraction via Side Channels

**Cache side channels.** One popular class of model extraction attacks is based on cache side channels, which monitors the cache accesses of the inference program. Hong et al. [32] recovered the architecture attributes by observing the invocations of critical functions in the deep learning frameworks (e.g., Pytorch, TensorFlow). Similar technique is also applied to NAS models [31]. However, these attacks are very coarse-grained. They can only identify convolutions without the specific types and hyper-parameters. Yan et al. [68] proposed Cache Telepathy, which monitors the GEMM calls in the low-level BLAS library. The number of GEMM calls can greatly narrow down the range of DNN hyper-parameters and then reveal the model architecture. Our method extends this technique to NAS models. Our improved solution can recover more sophisticated operations without the prior knowledge of the architecture family, which cannot be achieved in [68].

**Other side channels.** Some works leveraged other side channels to extract DNN models. Batina et al. [7] extracted a functionally equivalent model by monitoring the electromagnetic signals of a microprocessor hosting the inference program. Duddu et al. [19] found that models with different depths have different execution time, which can be used as a timing channel to leak the network details. Memory side-channels were discovered to infer the network structure of DNN models on GPUs [33] and DNN accelerators [34]. Future work will apply those techniques to our watermarking scheme.

### 7 Conclusion

In this work, we proposed a new direction of DNN model watermarking. We showed a carefully-crafted network architecture can be utilized as the ownership evidence, which exhibits stronger resilience against model transformations than previous parameter-based watermarks. We leveraged the Neural Architecture Search technique to produce watermarked architecture, and cache side channels to extract the black-box models for ownership verification. Evaluations indicate our scheme can provide great effectiveness, usability, robustness, and uniqueness, making it a promising and practical option for IP protection of AI products.
References

[1] Amazon SageMaker - Machine Learning. https://aws.amazon.com/sagemaker/, 2021.

[2] Adi, Y., Baum, C., Cisse, M., Pinkas, B., and Keshet, J. Turning your weakness into a strength: Watermarking deep neural networks by backdooring. In USENIX Security Symposium (2018), pp. 1615–1631.

[3] Aiken, W., Kim, H., and Woo, S. Neural network laundering: Removing black-box backdoor watermarks from deep neural networks. arXiv preprint arXiv:2004.11368 (2020).

[4] Atiya, A. O. F., Qian, Z., Krishnamurthy, S. V., La Porta, T., McDaniel, P., and Marvel, L. Malicious co-residency on the cloud: Attacks and defense. In IEEE Conference on Computer Communications (2017), pp. 1–9.

[5] Atiya, A. O. F., Qian, Z., Krishnamurthy, S. V., La Porta, T., McDaniel, P., and Marvel, L. M. Catch me if you can: A closer look at malicious co-residency on the cloud. IEEE/ACM Transactions on Networking 27, 2 (2019), 560–590.

[6] Ba, L. J., and Caruana, R. Do deep nets really need to be deep? arXiv preprint arXiv:1312.6184 (2013).

[7] Batina, L., Bhasin, S., Jap, D., and Picek, S. CSI NN: Reverse engineering of neural network architectures through electromagnetic side channel. In USENIX Security Symposium (2019), pp. 515–532.

[8] Bender, G., Kindermans, P.-J., Zoph, B., Vasudevan, V., and Le, Q. Understanding and simplifying one-shot architecture search. In International Conference on Machine Learning (2018), pp. 550–559.

[9] Brassier, F., Müller, U., Dmitrienko, A., Kostiainen, K., Capkun, S., and Sadeghi, A.-R. Software grand exposure: SGX cache attacks are practical. In USENIX Workshop on Offensive Technologies (2017).

[10] Brock, A., Lim, T., Ritchie, J. M., and Weston, N. SMASH: One-shot model architecture search through hypernetworks. arXiv preprint arXiv:1708.05344 (2017).

[11] Brown, T. B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., et al. Language models are few-shot learners. arXiv preprint arXiv:2005.14165 (2020).

[12] Chen, K., Guo, S., Zhang, T., Li, S., and Liu, Y. Temporal watermarks for deep reinforcement learning models. In International Conference on Autonomous Agents and Multiagent Systems (2021).

[13] Chen, X., Wang, W., Bender, C., Ding, Y., Jia, R., Li, B., and Song, D. REFIT: A unified watermark removal framework for deep learning systems with limited data. arXiv preprint arXiv:1911.07205 (2019).

[14] Chu, X., Zhang, B., Xu, R., and Li, J. Fairnas: Rethinking evaluation fairness of weight sharing neural architecture search. arXiv preprint arXiv:1907.01845 (2019).

[15] Chu, X., Zhou, T., Zhang, B., and Li, J. Fair DARTS: Eliminating unfair advantages in differentiable architecture search. In European Conference on Computer Vision (2020), pp. 465–480.

[16] Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. BERT: Pre-training of deep bidirectional transformers for language understanding. arXiv preprint arXiv:1810.04805 (2018).

[17] Dong, X., and Yang, Y. Searching for a robust neural architecture in four GPU hours. In IEEE Conference on Computer Vision and Pattern Recognition (2019), pp. 1761–1770.

[18] Dong, X., and Yang, Y. NAS-Bench-201: Extending the scope of reproducible neural architecture search. arXiv preprint arXiv:2001.00326 (2020).

[19] Duddu, V., Samanta, D., Rao, D. V., and Balas, V. E. Stealing neural networks via timing side channels. arXiv preprint arXiv:1812.11720 (2018).

[20] Elsken, T., Metzen, J. H., and Hutter, F. Efficient multi-objective neural architecture search via lamarckian evolution. arXiv preprint arXiv:1804.09081 (2018).

[21] Elsken, T., Metzen, J. H., Hutter, F., et al. Neural architecture search: A survey. Journal of Machine Learning Research 20, 55 (2019), 1–21.

[22] Ezhilchelvan, P. D., and Mitra, I. Evaluating the probability of malicious co-residency in public clouds. IEEE Transactions on Cloud Computing 5, 3 (2015), 420–427.

[23] Goto, K., and Geijn, R. A. V. D. Anatomy of high-performance matrix multiplication. ACM Transactions on Mathematical Software 34, 3 (2008), 1–25.

[24] Götzfried, J., Eckert, M., Schinzel, S., and Müller, T. Cache attacks on Intel SGX. In European Workshop on Systems Security (2017), pp. 1–6.

[25] Gruss, D., Lipp, M., Schwarz, M., Genkin, D., Juffinger, J., O’Connell, S., Schoechl, W., and Yarom, Y. Another flip in the wall of rowhammer defenses. In IEEE Symposium on Security and Privacy (2018), pp. 245–261.

[26] Gu, Z., Huang, H., Zhang, J., Su, D., Jamjoom, H., Lamba, A., Pendarakis, D., and Molloy, I. Yerbabuena: Securing deep learning inference data via enclave-based ternary model partitioning. arXiv preprint arXiv:1807.09969 (2018).

[27] Guo, S., Zhang, T., Qu, H., Zeng, Y., Xiang, T., and Liu, Y. The hidden vulnerability of watermarking for deep neural networks. arXiv preprint arXiv:2009.06897 (2020).

[28] Hämnel, M., Cui, W., and Peinado, M. High-resolution side channels for untrusted operating systems. In USENIX Annual Technical Conference (2017), pp. 299–312.

[29] He, K., Zhang, X., Ren, S., and Sun, J. Deep residual learning for image recognition. In IEEE Conference on Computer Vision and Pattern Recognition (2016), pp. 770–778.

[30] Hinton, G., Vinyals, O., and Dean, J. Distilling the knowledge in a neural network. arXiv preprint arXiv:1503.02531 (2015).

[31] Hong, S., Davinroy, M., Kaya, Y., Dachman-Soled, D., and Dumitras, T. How to own NAS in your spare time. arXiv preprint arXiv:2002.06776 (2020).

[32] Hong, S., Davinroy, M., Kaya, Y., Locke, S. N., Rackow, I., Kulda, K., Dachman-Soled, D., and Dumitras, T. Security analysis of deep neural networks operating in the presence of cache side-channel attacks. arXiv preprint arXiv:1810.03487 (2018).

[33] Hu, X., Liang, L., Li, S., Deng, L., Zuo, P., Ji, Y., Xie, X., Ding, Y., Liu, C., Sherwood, T., et al. DeepSniffer: A DNN model extraction framework based on learning architectural hints. In International Conference on Architectural Support for Programming Languages and Operating Systems (2020), pp. 385–399.

[34] Hua, W., Zhang, Z., and Suh, G. E. Reverse engineering convolutional neural networks through side-channel information leaks. In ACM/ESDA/IEEE Design Automation Conference (2018), IEEE, pp. 1–6.

[35] Jagielski, M., Carlini, N., Berthelot, D., Kurakin, A., and Papernot, N. High accuracy and high fidelity extraction of neural networks. In USENIX Security Symposium (2020).

[36] Kaplan, D., Powell, J., and Woller, T. Amid memory encryption. In White Paper (2016).

[37] Karl, R., Takeshita, J., and Jung, T. Using Intel SGX to improve private neural network training and inference. In Symposium on Hot Topics in the Science of Security (2020), pp. 1–2.

[38] Kim, K., Kim, C. H., Rhee, J. J., Yu, X., Chen, H., Tian, D., and Lee, B. Vessels: efficient and scalable deep learning prediction on trusted processors. In ACM Symposium on Cloud Computing (2020), pp. 462–476.
[39] KRIZHEVSKY, A., SUTSKEVER, I., AND HINTON, G. E. ImageNet classification with deep convolutional neural networks. *Advances in Neural Information Processing Systems* 25 (2012), 1097–1105.

[40] KUNKEL, R., QUOC, D. L., GREGOR, F., ARNAUTOV, S., BHATOTIA, P., AND PETZER, C. TensorSConE: A secure TensorFlow framework using Intel SGX. *arXiv preprint arXiv:1902.04413* (2019).

[41] LE MERRER, E., PEREZ, P., AND TREDAN, G. Adversarial frontier stitching for remote neural network watermarking. *Neural Computing and Applications* (2019), 1–12.

[42] LEE, T., LIN, Z., PUSHP, S., LI, C., LIU, Y., LEE, Y., XU, F., XU, C., ZHANG, L., AND SONG, J. Occlumency: Privacy-preserving remote deep-learning inference using SGX. In *Annual International Conference on Mobile Computing and Networking* (2019), pp. 1–17.

[43] LI, Z., HU, C., ZHANG, Y., AND GUO, S. How to prove your model belongs to you: A blind-watermark based framework to protect intellectual property of DNN. In *Annual Computer Security Applications Conference* (2019), pp. 126–137.

[44] LIU, C., ZOPH, B., NEUMANN, M., SHELNS, J., HUA, W., LI, L.-J., FEI-FEI, L., YUille, A., HUANG, J., AND MURPHY, K. Progressive neural architecture search. In *European Conference on Computer Vision* (2018), pp. 19–34.

[45] LIU, F., YAROM, Y., GE, Q., HEISER, G., AND LEE, R. B. Last-level cache side-channel attacks are practical. In *IEEE Symposium on Security and Privacy* (2015), pp. 605–622.

[46] LIU, H., SIMONYAN, K., VINYALS, O., FERNANDO, C., AND KAVUKCUOGLU, K. Hierarchical representations for efficient architecture search. *arXiv preprint arXiv:1711.04346* (2017).

[47] LIU, H., SIMONYAN, K., AND YANG, Y. DARTS: Differentiable architecture search. *arXiv preprint arXiv:1806.09055* (2018).

[48] LIU, X., LI, F., WEN, B., AND LI, Q. Removing backdoor-based watermarks in neural networks with limited data. *arXiv preprint arXiv:2008.04007* (2020).

[49] MARSCHALEK, M. The wolf in SGX clothing. In *Bluehat LL* (2018).

[50] MCKEEN, F., ALEXANDROVICH, I., BERENZON, A., ROZAS, C. V., SHAPI, H., SHANBHOGUE, V., AND SAVAGAOIKAR, U. R. Innovative instructions and software model for isolated execution. *Hasp* *ifsca* 10, 1 (2013).

[51] NAMBA, R., AND SAKUMA, J. Robust watermarking of neural network with exponential weighting. In *ACM Asia Conference on Computer and Communications Security* (2019), pp. 228–240.

[52] OHRIMENKO, O., SCHUSTER, F., FOURNET, C., MEHTA, A., NOWOZIN, S., VASWANI, K., AND COSTA, M. Oblivious multi-party machine learning on trusted processors. In *USENIX Security Symposium* (2016), pp. 619–636.

[53] PHAM, H., GUAN, M. Y., ZOPH, B., LE, Q. V., AND DEAN, J. Efficient neural architecture search via parameter sharing. *arXiv preprint arXiv:1802.03268* (2018).

[54] REAL, E., AGGARWAL, A., HUANG, Y., AND LE, Q. V. Regularized evolution for image classifier architecture search. In *AAAI Conference on Artificial Intelligence* (2019), vol. 33, pp. 4780–4789.

[55] RISTENPART, T., TROMER, E., SHACHAM, H., AND SAVAGE, S. Hey, you get off my cloud: exploring information leakage in third-party compute clouds. In *ACM Conference on Computer and Communications Security* (2009), pp. 199–212.

[56] ROUHANI, B. D., CHEN, H., AND KOUHSHAFAR, F. DeepSigns: An end-to-end watermarking framework for protecting the ownership of deep neural networks. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (2019).

[57] SCHWARZ, M., WEISER, S., AND GRUSS, D. Practical enclave malware with Intel SGX. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment* (2019), Springer, pp. 177–196.

[58] SCHWARZ, M., WEISER, S., GRUSS, D., MAURICE, C., AND MANGARD, S. Malware guard extension: Using SGX to conceal cache attacks. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment* (2017), Springer, pp. 3–24.

[59] SENIOR, A. W., EVANS, R., JUMPER, J., KIRKPATRICK, J., SIFRE, L., GREEN, T., QIN, C., ZÍDEK, A., NELSON, A. W., BRIDLAND, A., ET AL. Improved protein structure prediction using potentials from deep learning. *Nature* 577, 7792 (2020), 706–710.

[60] SHAHINEJAD, M., WANG, J., LUKAS, N., LI, X., AND KERSCHBAUM, F. On the robustness of the backend-based watermarking in deep neural networks. *arXiv preprint arXiv:1906.07745* (2019).

[61] SILVER, D., HUANG, A., MADDISON, C. J., GUEZ, A., SIFRE, L., VAN DEN DRIESSCHE, G., SCHRITTWIESER, J., ANTONoglou, I., PANNEERShelvam, V., LANCTOT, M., ET AL. Mastering the game of go with deep neural networks and tree search. *Nature* 529, 7587 (2016), 484–489.

[62] SIMONYAN, K., AND ZISSERMAN, A. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556* (2014).

[63] TRAMER, F., ZANG, F., JUELS, A., REITER, M. K., AND RISTENPART, T. Stealing machine learning models via prediction apis. In *USENIX Security Symposium* (2016), pp. 601–618.

[64] UCHIDA, Y., NAGAI, Y., SAKAZAWA, S., AND SATOH, S. Embedding watermarks into deep neural networks. In *ACM on International Conference on Multimedia Retrieval* (2017), pp. 269–277.

[65] VARADARAJAN, V., ZHANG, Y., RISTENPART, T., AND SWIFT, M. A placement vulnerability study in multi-tenant public clouds. In *USENIX Security Symposium* (2015), pp. 913–928.

[66] WANG, B., AND GONG, N. Z. Stealing hyperparameters in machine learning. In *IEEE Symposium on Security and Privacy* (2018), pp. 36–52.

[67] XU, Z., WANG, H., AND WU, Z. A measurement study on co-residence threat inside the cloud. In *USENIX Security Symposium* (2015), pp. 929–944.

[68] YAN, M., Fletcher, C. W., AND Torrellas, J. Cache telepathy: Leveraging shared resource attacks to learn DNN architectures. In *USENIX Security Symposium* (2020), pp. 2003–2020.

[69] YAROM, Y., AND FALKNER, K. FLUSH+ RELOAD: A high resolution, low noise, L3 cache side-channel attack. In *USENIX Security Symposium* (2014), pp. 719–732.

[70] YU, H., YANG, K., ZHANG, T., TSAI, Y.-Y., HO, T.-Y., AND JIN, Y. CloudLeak: Large-scale deep learning models stealing through adversarial examples. In *Network and Distributed Systems Security Symposium* (2020).

[71] ZHANG, J., CHEN, D., JIAO, J., FANG, H., ZHANG, W., ZHOU, W., CUI, H., AND YU, N. Model watermarking for image processing networks. In *AAAI Conference on Artificial Intelligence* (2020), vol. 34, pp. 12805–12812.

[72] ZHANG, J., GU, Z., JANG, J., WU, H., STOECKLIN, M. P., HUANG, H., AND MOLLOJ, I. Protecting intellectual property of deep neural networks with watermarking. In *ACM Asia Conference on Computer and Communications Security* (2018), pp. 159–172.

[73] ZHANG, Y., JUELS, A., REITER, M. K., AND RISTENPART, T. CrossVM side channels and their use to extract private keys. In *ACM Conference on Computer and Communications Security* (2012), pp. 305–316.

[74] ZHANG, Y., JUELS, A., REITER, M. K., AND RISTENPART, T. Cross-tenant side-channel attacks in Paas clouds. In *ACM SIGSAC Conference on Computer and Communications Security* (2014), pp. 990–1003.

[75] ZOPH, B., AND LE, Q. V. Neural architecture search with reinforcement learning. *arXiv preprint arXiv:1611.01578* (2016).

[76] ZOPH, B., VASUDEVAN, V., SHLENS, J., AND LE, Q. V. Learning transferable architectures for scalable image recognition. In *IEEE Conference on Computer Vision and Pattern Recognition* (2018), pp. 8697–8710.
A Proof Sketch of Theorem 1

Proof Sketch. We prove the proposed watermarking scheme satisfies the watermarking properties.

Effectiveness. The property can be guaranteed by Assumption 2.

Usability. Let $S_{c_i,0}$, $S_{c_i}$ be the architecture search spaces before and after restricting the stamp $k_i$ of $c_i$. $S_{c_i,0}$ and $S_{c_i}$ are the two architecture searched from $S_{c_i,0}$ and $S_{c_i}$, respectively. $f_{c_i,0}$ and $f_{c_i}$ are the corresponding models trained on the the same data distribution $D$. From Assumption 1, we have

$$Pr[f_{c_i,0}(x) = y|(x,y) \sim D] - Pr[f_{c_i}(x) = y|(x,y) \sim D] \leq \frac{\varepsilon}{N}. \quad (5)$$

Let $f_0, f$ are the DNN models that are learned before and after restricting their architecture search spaces by a watermark. One can easily use the mathematical induction to prove the usability of our watermarking scheme, i.e.,

$$Pr[f_0(x) = y|(x,y) \sim D] - Pr[f(x) = y|(x,y) \sim D] \leq \varepsilon. \quad (6)$$

Robustness. We classify the model modification attacks into two categories. The first approach is to only change the parameters of $f$ using existing techniques such as fine-tuning and model compression. Since the architecture is preserved, the stamps of all cells are also preserved. According to Assumption 2, the idea analyzer can extract the stamps and verify the ownership of the modified models.

The other category of attacks modifies the architecture of the model. Since the marking key (watermark) is secret, the adversary can uniformly modify the operation of an edge or delete an edge in a cell. The probability that the adversary can successfully modify one edge/operation of a stamp is not larger than $\frac{\varepsilon}{|c_i|}$, where $|c_i|$ is the number of connected edges in $c_i$. Thus, the expected value of the total number of modification is $\delta \times \sum_{i} \frac{|c_i|}{N}$. However, since the adversary cannot access the proxy and task datasets, he cannot obtain new models with competitive performance by retraining the modified architectures.

Uniqueness. Without loss of generality, we assume the NAS algorithm can search the same architecture if the search spaces of all cells are the same. Thus, the uniqueness of the watermarked model is decided by the probability that the adversary can identify the same search spaces. Because the marking key is secret, the adversary has to guess the edges and the corresponding operations of each stamp if he wants to identify the same search spaces (the similarity ratio $\tau$ is 100% here). The probability that he can identify the same search space of a cell is much smaller than $\left(\frac{1}{10}\right)^{\varepsilon}$. Therefore, the uniqueness property of our watermarking scheme is proved if $\delta < \left(\frac{1}{10}\right)^{\varepsilon}$. As we empirically evaluated in Section 6.5, the watermarked models are unique and the probability of model collision is very low.

B Details about GEMM in OpenBLAS

BLAS realizes the matrix multiplication with the function `gemm`. This function computes $C = \alpha A \times B + \beta C$, where $A$ is an $m \times k$ matrix, $B$ is a $k \times n$ matrix, $C$ is an $m \times n$ matrix, and both $\alpha$ and $\beta$ are scalars. OpenBLAS adopts Goto’s algorithm [23] to accelerate the multiplication using modern cache hierarchies. This algorithm divides a matrix into small blocks (with constant parameters $P$, $Q$, $R$), as shown in Figure 15. The matrix $A$ is partitioned into $P \times Q$ blocks and $B$ is partitioned into $Q \times R$ blocks, which can be fit into the L2 and L3 caches, respectively. The multiplication of such two blocks generates a $P \times R$ block in the matrix $C$. Algorithm 4 shows the process of `gemm` that contains 4 loops controlled by the matrix size $(m, n, k)$. Functions `itcopy` and `oncopy` are used to allocate data and functions. `kernel` runs the actual computation. Note that the partition of $m$ contains two loops, `loop1` and `loop4`, where `loop4` is used to process the multiplication of the first $P \times Q$ block and the chosen $Q \times R$ block. For different cache sizes, OpenBLAS selects different values of $P, Q$ and $R$ to achieve the optimal performance.

C Details about the NAS Algorithms

C.1 Architecture Search

CIFAR10. We adopt GDAS [17] to search for the optimal CNN architectures on CIFAR10. We set the number of initial channels in first convolution layer as 16, the number of the computation nodes in a cell as 4 and the number of normal cells in a block as 2. Then we train the model for 240 epochs. The setting of the optimizer and learning rate schedule is the same as that in [17]. The search process on CIFAR10 takes about five hours with a single NVIDIA Tesla V100 GPU.

PTB. We adopt DARTS [47] to search for the optimal RNN architecture on PTB. Both the embedding and hidden sizes are set as 300, and the network is trained for 50 epochs using SGD optimization. We set the learning rate as 20, the batch size as 256, BPTT length as 35, and the weight decay as $5 \times 10^{-7}$. Other setting of the optimization of the architecture is also the same as [47]. The search process takes 6 hours on a single GPU.
C.2 Model Retraining

CIFAR After obtaining the searched cells, we form a CNN with 33 initial channels. We set number of computation nodes in a cell as 4 and the number of normal cells in a block as 6. Then we train the network for 300 epochs on the dataset (both CIFAR10 and CIFAR100), with a learning rate reducing from 0.025 to 0 with the cosine schedule. The preprocessing and data augmentation is the same as [17]. The training process takes about 11 GPU hours.

ImageNet For the CNN on ImageNet, we set the initial channel size as 52, and the number of normal cells in a block as 4. The network is trained with 250 epochs using the SGD optimization and the batch size is 128. The learning rate is initialized as 0.1, and is reduced by 0.97 after each epoch. The training process takes 12 days on a single GPU.

PTB A RNN with the searched recurrent cell is trained on PTB with the SGD optimization and the batch size of 64 until the convergence. Both the embedding and hidden sizes are set as 850. The learning rate is set as 20 and the weight decay is $8 \times 10^{-7}$. The training process takes 3 days on a single GPU.

D Monitored Functions in Pytorch and OpenBLAS

Table 4 gives the monitored lines of the code in the latest Pytorch 1.7.0 and OpenBLAS 0.3.13.

| Library   | Functions   | Code Line                                      |
|-----------|-------------|-----------------------------------------------|
| OpenBLAS  | Itcopy      | kernel/generic/gemm_tcopy_8.c:78              |
|           | Oncopy      | kernel/x86_64/gemm_ncopy_4_skylake.x.c:57    |
| Pytorch   | Relu        | aten/src/ATen/Functions.cpp:8332              |
|           | Tanh        | aten/src/ATen/native/UnaryOps.cpp:452         |
|           | Sigmoid     | aten/src/ATen/native/UnaryOps.cpp:389         |
|           | Avgpool     | aten/src/ATen/native/AdaptiveAveragePooling.cpp:325 |
|           | Maxpool     | aten/src/ATen/native/Pooling.cpp:47           |

Table 4: Monitored code lines in OpenBLAS and Pytorch.

E Whole side channel leakage trace

Figure 16 shows the whole side channel leakage trace of the tested NAS model in our end-to-end watermarking process. While the nodes representing the function accesses are stacked up, we can still identify the first block from interval 0 to around $2 \times 10^6$, where there are more accesses to itcopy (blue nodes). For the following two blocks, since the number of channels increases, the length of leakage windows also increases.

![Figure 16: Whole leakage trace of the NAS model.](image-url)