A Novel Watermarking Framework for Ownership Verification of DNN Architectures

Anonymous authors
Paper under double-blind review

Abstract

We present a novel watermarking scheme to achieve the intellectual property (IP) protection and ownership verification of DNN architectures. Existing works all embedded watermarks into the model parameters while treating the architecture as public property. These solutions were proven to be vulnerable by an adversary to detect or remove the watermarks. In contrast, we are the first to claim model architectures as an important IP for model owners, and propose to implant watermarks into the 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. Such watermarks can be extracted via side-channel-based model extraction techniques with high fidelity. Extensive evaluations show our scheme has negligible impact on the model performance, and exhibits strong robustness against various model transformations and adaptive attacks.

1 Introduction

Commercialization of the deep learning technology has made Deep Neural Network (DNN) models the core Intellectual Property (IP) of AI products and applications. Hence, release of DNN models can incur illegitimate plagiarism, unauthorized distribution or reproduction. One common IP protection approach is DNN watermarking, which processes the protected model in a *unique* way such that the model owner is able to recognize the ownership of his model without affecting its *usability*.

All the existing solutions treat the model parameters as the IP, and embed the unique watermarks into the parameters for ownership verification (Uchida et al., 2017; Rouhani et al., 2019; Adi et al., 2018; Zhang et al., 2020; Chen et al., 2021a). Unfortunately, those watermarking solutions are not practically robust. An adversary can easily detect or remove them by slightly modifying the parameters of a plagiarized model (Chen et al., 2021c; Shafieinejad et al., 2019; Liu et al., 2020; Guo et al., 2021; Namba & Sakuma, 2019; Aiken et al., 2020).

In this paper, we propose a fundamentally different watermarking scheme. Instead of protecting the parameters, we treat the network architecture as the IP of the model. There are a couple of incentives for the adversary to plagiarize the architectures (Yan et al., 2020; Hong et al., 2020). First, it is costly to craft a qualified architecture for a given task. Architecture design and testing require lots of valuable human expertise and experience. AutoML is introduced to automatically search for architectures (Zoph & Le, 2016), which still needs a large amount of time, computing resources and data samples. Second, the network architecture is critical in determining the model performance. Obtaining it brings high commercial values. Additionally, the adversary can steal an architecture and apply it to multiple tasks with different datasets, significantly improving the benefit. *Hence, it is worthwhile to treat the architecture design as an important IP and provide particular protection to it.*

We aim to design a methodology to generate unique network architectures for the owners, which can serve as the evidence of ownership. This scheme is more robust than previous solutions, as refining the parameters cannot tamper with the watermarks. The adversary can only *remarkably* change the network architecture in order to erase the watermarks. This will not violate the copyright, since the new architecture is totally different from the original one, and can be legally regarded as the adversary's own IP. Two questions need to be answered in order to establish this scheme: *(1) how to systematically design 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) (Zoph & Le, 2016), which can automatically discover the optimal network architecture for a given task and dataset. 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 space to produce a high-quality network architecture. This architecture is unique enough to represent the ownership of the model.

For the second question, the owner can use cache 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 extend prior solutions (Yan et al., 2020; Hong et al., 2018) 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 the types and hyper-parameters of these new operations from a side-channel pattern. This enables us to precisely extract the watermark from the model.

The integration of these techniques leads to the design of our watermarking framework. Experiments show that our method is immune to common model parameter transformations (fine-tuning, pruning), which could compromise prior solutions. Furthermore, we test some new adaptive attacks that moderately refine the architectures (shuffling operation order, adding useless operations), and confirm their incapability of removing the watermarks in the architecture.

2 Preliminaries

2.1 Neural Architecture Search

NAS (Zoph & Le, 2016; Elsken et al., 2019) 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 on a task dataset.

The *search space* of a NAS method defines the scope of neural networks to be designed and optimized. A practical strategy is to decompose the target neural network into multiple *cells*, and search for the optimal structure of a cell instead of the entire network (Zoph et al., 2018). A cell is generally represented as a directed acyclic graph (DAG), where each edge is associated with an operation selected from a predefined operation set (Pham et al., 2018). Figure 1 gives a toy cell *supernet* that contains three computation nodes (gray squares) and a set of three candidate operations (circles). Such *supernet* enables the sharing of network parameters, and significantly accelerates the search process (Liu et al., 2018b; Dong & Yang, 2019; Chu et al., 2019; Chen et al., 2021b). A NAS method (Zoph et al., 2018; Pham et al., 2018; Real et al., 2019; Liu et al., 2018b; Chu et al., 2020; Dong & Yang, 2019) is adopted to search for the optimal architecture of two types of cells: a *normal cell* for interpreting the features and a *reduction cell* for reducing the spatial size. Then multiple normal cells construct a block, and multiple blocks are interconnected by reduction cells to form the final model.

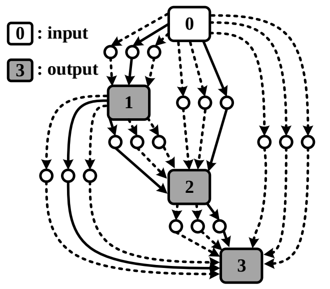


Figure 1: A toy cell *supernet*

Formally, we consider a NAS task, which aims to construct a model architecture containing N cells: $\mathfrak{A} = \{c_1, ..., c_N\}$. The search space of each cell is denoted as $S = (\mathcal{G}, \mathcal{O})$. $\mathcal{G} = (\mathcal{N}, \mathcal{E})$ is the DAG representing the cell *supernet*, where set $\mathcal{N}$ contains two inputs (a, b) from previous cells and $\mathcal{B}$ computation nodes in the cell, i.e., $\mathcal{N} = \{a, b, \mathcal{N}_1, ..., \mathcal{N}_\mathcal{B}\}$; $\mathcal{E} = \{\mathcal{E}_1, ..., \mathcal{E}_B\}$ is the set of all possible edges between nodes and $\mathcal{E}_j$ is the set of edges connected to the node $\mathcal{N}_j$ $(1 \leq j \leq \mathcal{B})$. Each node can only select at most two inputs from previous nodes. $\mathcal{O}$ is the set of candidate operations on these edges. Then we combine the search spaces of all cells as $\mathbb{S}$, from which we look for an optimal architecture $\mathfrak{A}$. The NAS method we consider is defined as below:

Definition 2.1. *(NAS) A NAS method is a machine learning algorithm that iteratively searches optimal cell architectures from the search space* $\mathbb{S}$ *on the proxy dataset* $\mathcal{D}$*. These cells construct one architecture* $\mathfrak{A} = \{c_1, ..., c_N\}$*, i.e.,* $\mathfrak{A} = \texttt{NAS}(\mathbb{S}, \mathcal{D})$.

After the search process, $\mathfrak{A}$ is trained from the scratch on the task dataset $\overline{\mathcal{D}}$ to obtain the optimal parameters and final DNN model $M = \texttt{train}(\mathfrak{A}, \overline{\mathcal{D}})$.

2.2 Threat Model

We consider that a model owner designs an architecture $\mathfrak{A}$ using a conventional NAS method, and trains a production-level DNN model M. An adversary may obtain an illegal copy of M and use it for profit without authorization. The goal of the model owner is to detect whether a suspicious model M' plagiarizes the architecture $\mathfrak{A}$ from M. He has black-box access to the target model M', without any knowledge about the architecture, parameters, training algorithms and hyper-parameters. We consider two techniques an adversary may employ to hide the evidence of architecture plagiarism. (1) *Parameter modification*: the adversary may alter the model parameters (e.g., fine-tuning, model compression, transfer learning) while maintaining similar performance. (2) *Architecture modification*: the adversary may moderately obfuscate the model architecture by changing the execution behaviors of model inference (reordering the operations, adding useless computations or neurons). However, we do not consider the case that the adversary redesigns the model architecture completely (e.g., knowledge distillation (Ba & Caruana, 2013; Hinton et al., 2015)), since the new model architecture is totally different, and can be legally regarded as the adversary's own asset.

We further follow the same assumption in (Yan et al., 2020; Batina et al., 2019; Hu et al., 2020) that the model owner can extract the inference execution trace of the target model M' via cache side channels. This is applied to the scenario where the suspicious application is securely packed with countermeasures against reverse-engineering, e.g., encryption. For instance, Trusted Execution Environment (TEE) (McKeen et al., 2013; Kaplan et al., 2016) has been abused by the adversary to launch attacks and hide malicious activities (Schwarz et al., 2017; Gruss et al., 2018). Similarly, an adversary can hide the stolen model in TEE (Ohrimenko et al., 2016; Kunkel et al., 2019) 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 framework, the model owner can extract watermarks from the isolated enclaves using cache side-channel techniques (Brasser et al., 2017).

3 Our Watermarking Scheme

3.1 Overview

We give the definition of the watermarking scheme for NAS architectures as below:

Definition 3.1. *A watermarking scheme for NAS is defined as a tuple of probabilistic polynomial time algorithms (***WMGen***,* **Mark***,* **Verify***), where*

- **WMGen***: takes the search space of a NAS method as input and outputs secret marking key mk and verification key vk.*
- **Mark***: outputs a watermarked architecture $\mathfrak{A}$ from a NAS method, a proxy dataset $\mathcal{D}$, and mk.*
- **Verify***: 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 has the following properties (Zhang et al., 2018; Adi et al., 2018). (1) **Effectiveness**: the watermarking scheme needs to guarantee the success of ownership verification over the watermarked $\mathfrak{A}$ using the verification key. (2) **Usability:** let $\mathfrak{A}_0$ be the original architecture without watermarks. For any data distribution $\mathcal{D}$, the watermarked architecture $\mathfrak{A}$ should exhibit competitive performance compared with $\mathfrak{A}_0$ on the data sampled from $\mathcal{D}$. (3) **Robustness**: since a probabilistic polynomial time adversary may modify f with common transformation or obfuscation operations, we expect the watermark remains in $\mathfrak{A}$ after those changes. (4) **Uniqueness**: a normal user can follow the same NAS method to learn an architecture from the same proxy dataset. Without the marking key, the probability that this architecture contains the same watermark should be smaller than a given threshold δ.

Figure 2 shows the overview of our watermarking framework, which consists of three stages. At stage 1 , the model owner generates a unique watermark and the corresponding key pair (mk, vk) using the algorithm **WMGen**. At stage 2, he adopts a conventional NAS method with the marking key mk to produce the watermarked architecture following the algorithm **Mark**. He then trains the model from this architecture. Stage 3 is to verify the ownership of a suspicious model: the owner collects the side-channel information at inference, and identifies any potential watermark based on the verification key vk using the algorithm **Verify**.

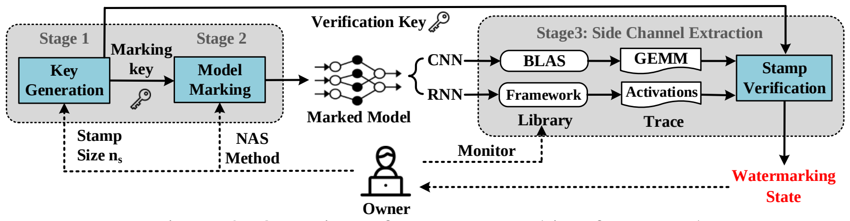


Figure 2: Overview of our watermarking framework

3.2 WATERMARK GENERATION (WMGEN)

According to Definition 2.1, a NAS architecture is a composition of cells. Each NAS cell is actually a sampled sub-graph of the *supernet* $\mathcal{G}$, where the attached operations are identified by the search strategy. To generate a watermark, the model owner selects some edges from $\mathcal{G}$ which can form a path[1]. Then he fixes each of these edges with a randomly chosen operation. The set of the fixed edge-operation pairs $\{s_e : s_o\}$ inside a cell is called a *stamp*, as defined below:

Definition 3.2. *(Stamp) A stamp for a cell is a set of edge-operation pairs $\{s_e : s_o\}$, where s_e, s_o denote the selected edges in a path and the corresponding operations, respectively.*

The combination of the stamps of all the cells form a watermark for a NAS architecture:

Definition 3.3. *(Watermark) Consider a NAS method with a proxy dataset $\mathcal{D}$ and search space $\mathbb{S}$. $\mathfrak{A} = \{c_1, ..., c_N\}$ represents the neural architecture produced from this method. A watermark for $\mathfrak{A}$ is a set of stamps $mk_1, ..., mk_N$, where mk_i is the stamp of cell c_i.*

Algorithm 1 illustrates the detailed procedure of constructing a watermark and the corresponding marking and verification keys (mk, vk). Given the supernet $\mathcal{G}$, we call function `GetPath` (Algorithm 4 in Appendix) to obtain a set S_e of all the possible paths with length n_s, where n_s is the predefined number of stamp edges ($1 \leq n_s \leq \mathcal{B}$). Then for each cell c_i, we randomly sample a path s_e from S_e. Each edge in the selected path is attached with a fixed operation chosen by the model owner, while in this paper we use a random operation, to form the cell stamp $mk_i = \{s_e : s_o\}$. Finally we can construct a marking key $mk = (mk_1, ..., mk_N)$. The verification key is $vk = (vk_1, ..., vk_N)$, where vk_i is the fixed operation sequence s_o in cell c_i.

```
Algorithm 1: Marking Key Generation (WMGen)
Input: # of fixed edges $n_s$, search space $\mathbb{S} = (\mathcal{G}, \mathcal{O})$
Output: marking key $mk$, verification key $vk$
$S_e$ = GetPath($\mathcal{G}, n_s$)
for $i$ from 1 to N do
    $s_e \leftarrow$ randomly select one path from $S_e$
    $s_o \leftarrow$ randomly select $n_s$ operations from $\mathcal{O}$ for $s_e$
    $mk_i = \{s_e : s_o\}, vk_i = s_o$
return $mk = (mk_1, ..., mk_N)$, $vk = (vk_1, ..., vk_N)$
```

3.3 WATERMARK EMBEDDING (MARK)

To generate a competitive DNN architecture embedded with the watermark, we fix the edges and operations in the marking key mk, and apply a conventional NAS method to search for the rest connections and operations for the optimal architecture. This process will have a smaller search space compared to the original method. However, as shown in previous works (Zoph et al., 2018; Liu et al., 2018b), there are multiple sub-optimal results with comparable performance in the NAS search space, which makes random search also feasible. Hence, we hypothesize that we can still find out qualified results from the reduced search space. Experimental evaluations in Section 5 verify that the reduced search space incurs negligible impact on the model performance.

Algorithm 2 shows the procedure of embedding the watermark to a NAS architecture. For each cell c_i in the architecture, we first identify the fixed stamp edges and operations $\{s_e : s_o\}$ from key mk_i. Then the cell search space S is updated as $(\mathcal{G} = (\mathcal{N}, \overline{\mathcal{E}}), \mathcal{O})$, where $\overline{\mathcal{E}}$ is the set of connection edges excluding those fixed ones: $\overline{\mathcal{E}} = \mathcal{E} - s_e$. The updated search spaces of all the cells are combined to form the search space $\mathbb{S}$, from which the NAS method is used to find the optimal architecture $\mathfrak{A}$ containing the desired watermark.

[1]We select the edges in a path as the executions of their operations have dependency (see the red edges in Figure 4). So an adversary cannot remove the watermarks by shuffling the operation order at inference.

3.4 Watermark Verification (Verify)

During verification, we utilize cache side channels to capture an execution trace $\mathbb{T}$ by monitoring the inference process of the target model M'. Details about side-channel extraction can be found in Section 4. Due to the existence of extra computations like concatenating and preprocessing, cells in $\mathbb{T}$ are separated with much larger time intervals and hence can be intuitively identified as sequential leakage windows. If $\mathbb{T}$ 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 verification process. First the side-channel leakage trace $\mathbb{T}$ is divided into cell windows, and for the i-th window, we retrieve its stamp operations s_o from vk_i. Then the cluster patterns in the window are analyzed in sequence. Since the adversary can possibly shuffle the operation order or add useless computations to obfuscate the trace, *we only verify if the stamp operations exist in the cell in the correct order, which is not affected by the obfuscations due to their execution dependency, while ignoring other operations*. We claim the architecture ownership once all cells contain the corresponding stamp operation sequences.

Algorithm 2: Watermark Embedding (**Mark**)

```
Input: marking key $mk$, NAS method,
       proxy dataset $\mathcal{D}$
Output: watermarked architecture $\mathfrak{A}$
$\mathbb{S} \leftarrow$ search space of the whole model
for each cell $c_i$ do
    retrieve $\{s_e : s_o\}$ from $mk_i$
    $\overline{\mathcal{E}} = \mathcal{E} - s_e$
    $S = (\mathcal{G} = (\mathcal{N}, \overline{\mathcal{E}}), \mathcal{O})$
    $\mathbb{S}$.append($S$)
$\mathfrak{A} = \mathtt{NAS}(\mathbb{S}, \mathcal{D})$
return $\mathfrak{A}$
```

Algorithm 3: Watermark Verification (**Verify**)

```
Input: verification key $vk$, monitored trace $\mathbb{T}$
Output: verification result $r$
Split $\mathbb{T}$ into cell windows
for each $window_i$ in $\mathbb{T}$ do
    retrieve $s_o$ from $vk_i$, $id \leftarrow 0$
    for each $cluster$ in $window_i$ do
        if match(cluster, $s_o[id]$) = True then
            $id += 1$
    $r = (id = n_s) ? True : False$
    if not $r$ then
        return $r$
return $r$
```

3.5 Theoretical Analysis

We theoretically prove that our proposed three algorithms (**WMGen**, **Mark**, **Verify**) form a qualified watermarking scheme for NAS architectures, satisfying the properties in Section 3.1. The proof can be found in Appendix A.

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

4 Side Channel Extraction

Given a suspicious model, we aim to extract the embedded watermark using cache side channels. Past works (Yan et al., 2020; Hong et al., 2018) only focused on conventional DNN models and normally required knowledge of the target model's architecture family. Hence, they cannot extract novel NAS architectures with more sophisticated operations (e.g., separable convolutions, dilated-separable convolutions). We design an enhanced methodology to extract the architectures of NAS models by monitoring the side-channel pattern. To accelerate the computation, complex DNN operations (e.g., convolutions) are generally transformed to General Matrix Multiply (GEMM) that is achieved by the low-level BLAS library. We monitor the function activities in BLAS to recover the model architecture and operations. In this paper, we take OpenBLAS as an example, while our method is also generalized to other BLAS libraries, such as Intel MKL.

Our method contains three steps: (1) we monitor the memory accesses to the *itcopy* and *oncopy* functions in OpenBLAS and record the pattern of *itcopy-oncopy* loops. These two functions are used to load matrix data for multiplication, and the number of their invocations is determined by the matrix dimensions. (2) we derive possible values of matrix dimensions (m, n, k) based on the obtained *itcopy-oncopy* pattern, for the multiplication of matrix A ($m \times k$) and matrix B ($k \times n$). (3) The operation type and hyper-parameters are revealed through deduced matrix dimensions, following the rules of matrix transformation for different operations.

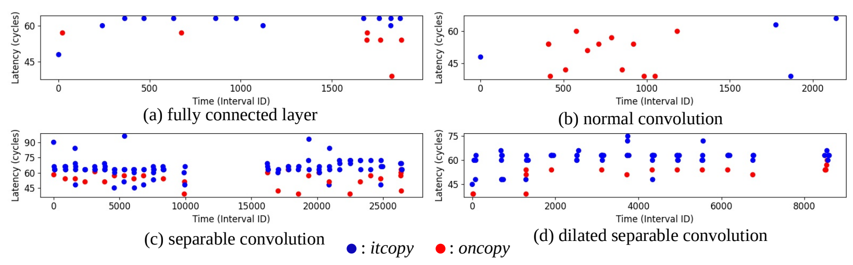


Figure 3: Side-channel patterns of four operations in NAS (sampling interval is 2000 CPU cycles).

Figure 3 shows the leakage pattern of four representative operations used in NAS. Figure 3(a) is for a classifier with two fully-connected (FC) layers, where we can clearly identify two separate clusters. Figure 3(b) is for a normal convolution. In terms of the leakage pattern, a normal convolution is hard to be distinguished from a FC layer, so that past work (Yan et al., 2020) needs to know the architecture family to distinguish the operations. 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. Figures 3(c) and 3(d) show the trace of a separable convolution and a dilated separable (DS) convolution with the same configurations, respectively. The leakage patterns of such two operations are fairly distinguishable, which contains C consecutive clusters and one individual cluster at the end, where C is the number of input channels. Note that in a NAS model, the separable convolution is always applied twice (Zoph et al., 2018; Real et al., 2019; Liu et al., 2018a;b) to improve the performance, which makes its leakage pattern more recognizable. Besides, it is easy to see the performance advantage of the DS convolution (8400 intervals) over the separable convolution (10000 intervals). The reason is that the input matrix in a DS convolution contains more padding zeros to reduce the computation complexity. More details about the leakage patterns and hyper-parameter recovery of different NAS operations can be found in Appendix D.

5 Evaluation

Configurations and implementation. Our approach is general for different types of deep learning frameworks and libraries, and can be applied to all cell-based NAS methods. Without loss of generality, we adopt Pytorch (1.8.0) and OpenBLAS (0.3.15). We mainly focus on the CNN tasks, and select a state-of-the-art NAS method GDAS (Dong & Yang, 2019). It contains eight candidate operations: identity, zeroize, 3×3 and 5×5 separable convolutions (SC), 3×3 and 5×5 dilated separable convolutions (DS), 3×3 average pooling (AP), 3×3 max pooling (MP). We adopt CIFAR10 as the proxy dataset to search the optimal architecture, and the searched architecture is trained over different datasets, e.g., CIFAR10, CIFAR100, ImageNet. We also consider watermarking RNN models. We choose the DARTS (Liu et al., 2018b) method. The candidate operations for a RNN cell contains *tanh*, *relu*, *sigmoid* activations, identity and zeroize. We use the PTB dataset to search and train RNN models. More details can be found in Appendix E.

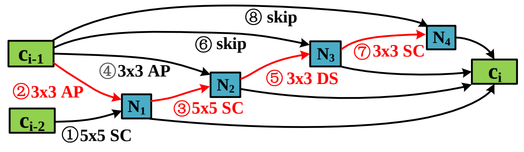
Side channel extraction. For CNN models, we monitor the *itcopy* and *oncopy* functions in OpenBLAS. For RNN models, we monitor the activation functions (*tanh*, *relu* and *sigmoid*) in Pytorch, since executions in OpenBLAS do not leak information about the models. We adopt the Flush+Reload side-channel technique (Yarom & Falkner, 2014), 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 F.

5.1 Effectiveness

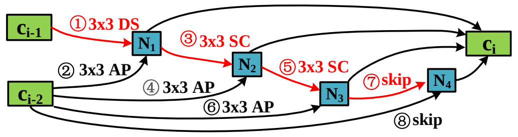
mk_n	s_{en}	$c_{i-2} \rightarrow \mathcal{N}_0$	$\mathcal{N}_0 \rightarrow \mathcal{N}_1$	$\mathcal{N}_1 \rightarrow \mathcal{N}_2$	$\mathcal{N}_2 \rightarrow \mathcal{N}_3$
	s_{on}	3×3 AP	5×5 SC	3×3 DS	3×3 SC
mk_r	s_{er}	$c_{i-1} \rightarrow \mathcal{N}_0$	$\mathcal{N}_0 \rightarrow \mathcal{N}_1$	$\mathcal{N}_1 \rightarrow \mathcal{N}_2$	$\mathcal{N}_2 \rightarrow \mathcal{N}_3$
	s_{or}	3×3 DS	3×3 SC	3×3 SC	skip

Table 1: An example of the marking key mk. We use *skip* to denote the *identity* operation.

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 $mk = (mk_n, mk_r)$, where $mk_n = \{s_{en} : s_{on}\}$ and $mk_r = \{s_{er} : s_{or}\}$ represent the stamps embedded to the normal and reduction cells, respectively. Each cell has four computation nodes ($\mathcal{B} = 4$), and we set



(a) Normal cell



(b) Reduction cell

Figure 4: Architectures of the searched cells. c_{i-1} and c_{i-2} are the inputs from the previous cells.

the number of stamp edges $n_s = 4$ for both cells, indicating four causal edges in each cell are fixed and attached with random operations. We follow Algorithm 1 to generate one example of mk (Table 1). The verification key $vk = (vk_n, vk_r)$ is also recorded, where $vk_n = s_{on}$ and $vk_r = s_{or}$.

Watermark embedding. We follow Algorithm 2 to embed the watermark determined by mk to the DNN architecture during the search process. Figure 4 shows the architectures of two cells searched by GDAS, where stamps are marked as red edges, and the computing order of each operation is annotated with numbers. These two cells are further stacked to construct a complete DNN architecture, including three normal blocks (each contains six normal cells) connected by two reduction cells. The number of filters is doubled in the reduction cells, and the channel sizes of three normal blocks are set as 33, 66 and 132. The searched model achieves a 3.51% error rate over CIFAR10, which is just slightly higher than the original model (3.32%) searched from the complete search space.

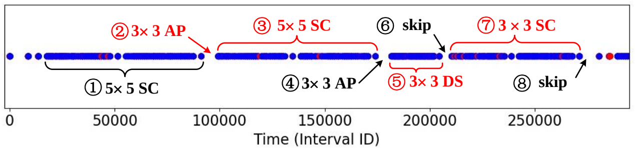
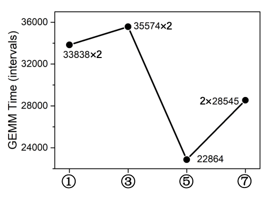
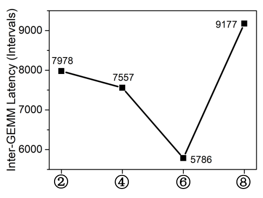


Figure 5: A side-channel trace of the first normal cell.

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. Following Algorithm 3, we divide the trace into cells and check if each cell contains the fixed operation sequence given by vk. Figure 5 illustrates the leakage trace of the first normal cell as an example. Other cells can be analyzed in the same way. From the figure, we can observe four large clusters, which can be easily identified according to their leakage patterns that ①, ③ and ⑦ are SCs while ⑤ is a DS. Figure 6a shows the measured execution time of these four GEMM operations. An interesting observation is that 5×5 convolution takes much longer time than 3×3 convolution, because it computes on a larger matrix. Such timing difference enables us to identify the kernel size when the search space is limited. Besides, we can also infer that the channel size is 33, since each operation contains $C = 33$ consecutive sub-clusters[2]. Figure 6b gives the inter-GEMM latency in the cell. The latency of ② and ④ is much larger, indicating they are pooling operations. Particularly, the latency of ⑧ contains two parts: *skip* and interval between two cells. The three small clusters at the beginning of the trace are identified as three normal convolutions used for preprocessing the input. Finally, after identifying the fixed operation sequences (s_o) in all cells, we can claim the architecture ownership of the DNN model.

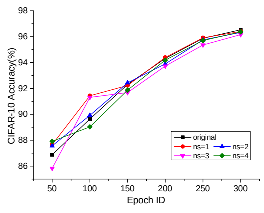


(a) GEMM operations

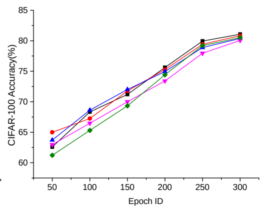


(b) Inter-GEMM latency

Figure 6: Execution time of the operations



(a) CIFAR-10



(b) CIFAR-100

Figure 7: Top-1 validation accuracy

5.2 USABILITY

To evaluate the usability property, we vary the number of stamp edges n_s from 1 to 4 to search watermarked architectures. Then we train the models over CIFAR10, CIFAR100 and ImageNet, and measure the validation accuracy. Figure 7 shows the average results on CIFAR dataset of five

[2]The value of C can be identified if we zoom in Figure 5, which is not shown in this paper due to page limit.

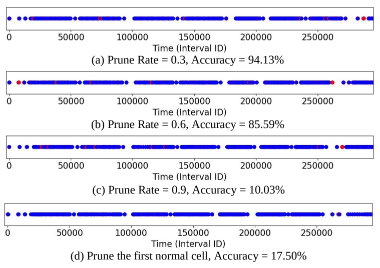


Figure 8: Traces of pruned models

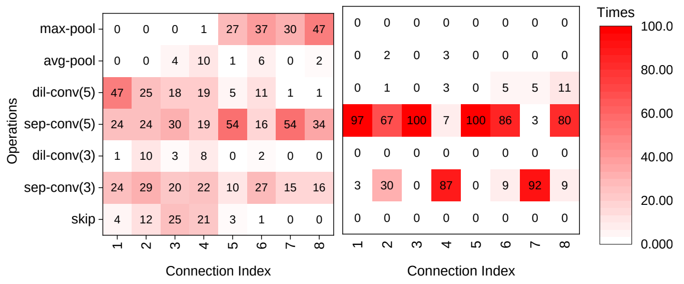


Figure 9: Left: normal cell; Right:reduction cell

experiments versus the training epochs. We observe that models with different stamp sizes have quite distinct performance at epoch 100. 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 ($n_s = 3$) gives an accuracy of 96.16%. Similarly for CIFAR100, the baseline accuracy and worst accuracy ($n_s = 4$) are 81.07% and 80.35%. 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 original model and two watermarked models ($n_s = 2$ and 4), which are also roughly the same (73.97%, 73.16% and 72.73%). This confirms our watermarking scheme does not affect the usability of the model.

5.3 Robustness

We consider the robustness of our watermarking scheme against two types of model operations.

Model transformation. Prior parameter-based solutions (Adi et al., 2018; Zhang et al., 2020; Chen et al., 2021a) are proven to be vulnerable against model fine-tuning or image transformations (Chen et al., 2021c; Shafieinejad et al., 2019; Liu et al., 2020; Guo et al., 2021). In contrast, our scheme is robust against these transformations as it only modifies the network architecture. First, we consider four types of fine-tuning operations evaluated in (Adi et al., 2018) (*Fine-Tune Last Layer*, *Fine-Tune All Layers*, *Re-Train Last Layer*, *Re-Train All Layers*). We verify that they do not corrupt our watermarks embedded to the model architecture. Second, we consider model compression. Common pruning techniques set certain parameters to 0 to shrink the network size. However, the GEMM computations are still performed over pruned parameters, which give similar side-channel patterns. Figures 8(a)-(c) show the extraction trace of the first normal cell after the entire model is pruned with different rates (0.3, 0.6, 0.9) using L_2-norm. Figure 8(d) shows one case where we prune all the parameters in the first normal cell. We observe that a bigger pruning rate can decrease the length of the leakage window, as there are more zero weights to simplify the computation. However, the pattern of the operations in the cell keeps unchanged, indicating the weight pruning cannot remove the embedded watermark.

Model obfuscation. An adversary may also obfuscate the inference execution to interfere with the verification results. (1) He can shuffle the orders of some operations which can be executed in parallel. However, since the selected stamp operations are in a path, they have high dependency and must be executed in the correct order. Hence, we can still identify the fixed operation sequence from the leakage trace of obfuscated models. (2) The adversary can add useless computations (e.g., matrix multiplications), operations or neurons to obfuscate the side-channel trace. Again, the critical stamp operations are still in the trace, and the owner is able to verify the ownership regardless of the extra operations. We conduct experiments to show the robustness of our solution against those obfuscations (Appendix G). In short, *the stamp operations must be executed sequentially and cannot be removed in a lightweight manner.* This makes it difficult to remove the watermarks in the architecture.

Note that an adversary can leverage some powerful methods (e.g., knowledge distillation (Ba & Caruana, 2013; Hinton et al., 2015)) to fundamentally change the architecture of the target model and possibly erase the watermarks. However, this is not flagged as copyright violation, since the adversary needs to spend a quantity of effort and cost (computing resources, time, dataset) to obtain a new model. This model is significantly different from the original one, and is regarded as the adversary's legitimate property.

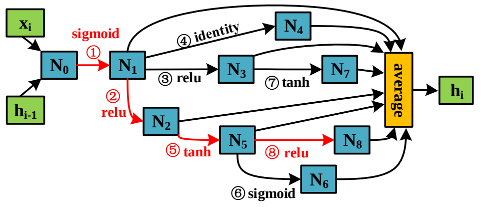


Figure 10: Recurrent cell learned on PTB.

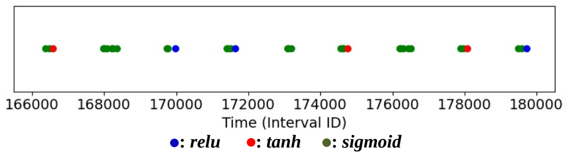


Figure 11: Side-channel trace of a recurrent cell.

5.4 Uniqueness

The theoretical analysis assumes each edge selects various operations with equal probability, and shows the collision rate is less than 0.03% (see Appendix A). We further empirically evaluate the uniqueness of our watermarking scheme. Specifically, 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. Figure 9 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, which does not match our assumptions. Such feature can help us to select more unique operation sequence as the marking key. Besides, the collision probability is decreased when the stamp size n_s is larger. A stamp size of 4 with fixed edge-operation selection can already achieve strong uniqueness.

5.5 Watermarking RNN Models

Besides CNN models, our scheme can also watermark RNN architectures. In a recurrent cell, each 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 10 shows an example of a recurrent cell searched on the PTB dataset with DARTS. Two inputs (input x_i and hidden state of the last layer) are added and passed to a *tanh* function for the initial node (Liu et al., 2018b; Pham et al., 2018).

Our scheme selects a path of $n_s = 4$ edges with random operations as the stamp (red edges in Figure 10). Since the search space for a recurrent cell only contains activation functions, which cannot be observed from the GEMM trace, we monitor the function accesses in PyTorch instead. Figure 11 shows the trace of a RNN model. We observe a recurrent cell contains 9 separate clusters: the first one denotes the *tanh* operation at the cell input and the remaining clusters are the operations attached to the nodes. The input to each node is processed by *sigmoid*, followed by the searched NAS operations. Compared with the CNN trace, a RNN model has a much more observable pattern, where each cluster corresponds to a node in order. The watermark has relatively larger impact on the performance of the searched RNN model, as the recurrent cell is more sensitive from the initial state. Table 2 shows the perplexity of the watermarked models over the validation and test datasets, with different stamp sizes n_s. The watermark can increase the perplexity of the RNN model. However, such performance is still satisfactory (Liu et al., 2018b), proving the usability of our scheme.

Perplexity	Original	Stamp Size n_s							
		1	2	3	4	5	6	7	8
Valid	58.1	59.73	59.65	62.85	61.62	60.33	60.27	62.23	62.93
Test	55.7	57.96	57.7	60.82	59.48	58.44	57.86	59.94	61.03

Table 2: Perplexity of watermarked RNN models with various n_s.

6 Conclusion

In this paper, we propose a new direction for DNN IP protection. We show a carefully-crafted network architecture can be utilized as the ownership evidence, which exhibits stronger resilience against model transformations than previous solutions. We leverage Neural Architecture Search to produce the 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

Yossi Adi, Carsten Baum, Moustapha Cisse, Benny Pinkas, and Joseph Keshet. Turning your weakness into a strength: Watermarking deep neural networks by backdooring. In *USENIX Security Symposium*, pp. 1615–1631, 2018.

William Aiken, Hyoungshick Kim, and Simon Woo. Neural network laundering: Removing black-box backdoor watermarks from deep neural networks. *arXiv preprint arXiv:2004.11368*, 2020.

Lei Jimmy Ba and Rich Caruana. Do deep nets really need to be deep? *arXiv preprint arXiv:1312.6184*, 2013.

Lejla Batina, Shivam Bhasin, Dirmanto Jap, and Stjepan Picek. CSI NN: Reverse engineering of neural network architectures through electromagnetic side channel. In *USENIX Security Symposium*, pp. 515–532, 2019.

Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. Software grand exposure: SGX cache attacks are practical. In *USENIX Workshop on Offensive Technologies*, 2017.

Kangjie Chen, Shangwei Guo, Tianwei Zhang, Shuxin Li, and Yang Liu. Temporal watermarks for deep reinforcement learning models. In *International Conference on Autonomous Agents and Multiagent Systems*, 2021a.

Minghao Chen, Houwen Peng, Jianlong Fu, and Haibin Ling. One-shot neural ensemble architecture search by diversity-guided search space shrinking. *arXiv preprint arXiv:2104.00597*, 2021b.

Xinyun Chen, Wenxiao Wang, Chris Bender, Yiming Ding, Ruoxi Jia, Bo Li, and Dawn Song. REFIT: A unified watermark removal framework for deep learning systems with limited data. *ACM ASIA Conference on Computer and Communications Security*, 2021c.

Xiangxiang Chu, Bo Zhang, Ruijun Xu, and Jixiang Li. Fairnas: Rethinking evaluation fairness of weight sharing neural architecture search. *arXiv preprint arXiv:1907.01845*, 2019.

Xiangxiang Chu, Tianbao Zhou, Bo Zhang, and Jixiang Li. Fair DARTS: Eliminating unfair advantages in differentiable architecture search. In *European Conference on Computer Vision*, pp. 465–480, 2020.

Xuanyi Dong and Yi Yang. Searching for a robust neural architecture in four GPU hours. In *IEEE Conference on Computer Vision and Pattern Recognition*, pp. 1761–1770, 2019.

Thomas Elsken, Jan Hendrik Metzen, Frank Hutter, et al. Neural architecture search: A survey. *Journal of Machine Learning Research*, 20(55):1–21, 2019.

Kazushige Goto and Robert A van de Geijn. Anatomy of high-performance matrix multiplication. *ACM Transactions on Mathematical Software*, 34(3):1–25, 2008.

Daniel Gruss, Moritz Lipp, Michael Schwarz, Daniel Genkin, Jonas Juffinger, Sioli O'Connell, Wolfgang Schoechl, and Yuval Yarom. Another flip in the wall of rowhammer defenses. In *IEEE Symposium on Security and Privacy*, pp. 245–261, 2018.

Shangwei Guo, Tianwei Zhang, Han Qiu, Yi Zeng, Tao Xiang, and Yang Liu. Fine-tuning is not enough: A simple yet effective watermark removal attack for dnn models. *International Joint Conference on Artificial Intelligence (IJCAI)*, 2021.

Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. Distilling the knowledge in a neural network. *arXiv preprint arXiv:1503.02531*, 2015.

Sanghyun Hong, Michael Davinroy, Yiğitcan Kaya, Stuart Nevans Locke, Ian Rackow, Kevin Kulda, Dana Dachman-Soled, and Tudor Dumitraş. Security analysis of deep neural networks operating in the presence of cache side-channel attacks. *arXiv preprint arXiv:1810.03487*, 2018.

Sanghyun Hong, Michael Davinroy, Yiğitcan Kaya, Dana Dachman-Soled, and Tudor Dumitraş. How to 0wn NAS in your spare time. *arXiv preprint arXiv:2002.06776*, 2020.

Xing Hu, Ling Liang, Shuangchen Li, Lei Deng, Pengfei Zuo, Yu Ji, Xinfeng Xie, Yufei Ding, Chang Liu, Timothy Sherwood, 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*, pp. 385–399, 2020.

David Kaplan, Jeremy Powell, and Tom Woller. Amd memory encryption. In *White Paper*, 2016.

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

Chenxi Liu, Barret Zoph, Maxim Neumann, Jonathon Shlens, Wei Hua, Li-Jia Li, Li Fei-Fei, Alan Yuille, Jonathan Huang, and Kevin Murphy. Progressive neural architecture search. In *European Conference on Computer Vision*, pp. 19–34, 2018a.

Hanxiao Liu, Karen Simonyan, and Yiming Yang. DARTS: Differentiable architecture search. *arXiv preprint arXiv:1806.09055*, 2018b.

Xuankai Liu, Fengting Li, Bihan Wen, and Qi Li. Removing backdoor-based watermarks in neural networks with limited data. *arXiv preprint arXiv:2008.00407*, 2020.

Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R Savagaonkar. Innovative instructions and software model for isolated execution. *Hasp@ isca*, 10(1), 2013.

Ryota Namba and Jun Sakuma. Robust watermarking of neural network with exponential weighting. In *ACM Asia Conference on Computer and Communications Security*, pp. 228–240, 2019.

Olga Ohrimenko, Felix Schuster, Cédric Fournet, Aastha Mehta, Sebastian Nowozin, Kapil Vaswani, and Manuel Costa. Oblivious multi-party machine learning on trusted processors. In *USENIX Security Symposium*, pp. 619–636, 2016.

Hieu Pham, Melody Y Guan, Barret Zoph, Quoc V Le, and Jeff Dean. Efficient neural architecture search via parameter sharing. *arXiv preprint arXiv:1802.03268*, 2018.

Esteban Real, Alok Aggarwal, Yanping Huang, and Quoc V Le. Regularized evolution for image classifier architecture search. In *AAAI Conference on Artificial Intelligence*, volume 33, pp. 4780–4789, 2019.

Bita Darvish Rouhani, Huili Chen, and Farinaz Koushanfar. 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.

Michael Schwarz, Samuel Weiser, Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Malware guard extension: Using SGX to conceal cache attacks. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pp. 3–24. Springer, 2017.

Masoumeh Shafieinejad, Jiaqi Wang, Nils Lukas, Xinda Li, and Florian Kerschbaum. On the robustness of the backdoor-based watermarking in deep neural networks. *arXiv preprint arXiv:1906.07745*, 2019.

Yusuke Uchida, Yuki Nagai, Shigeyuki Sakazawa, and Shin'ichi Satoh. Embedding watermarks into deep neural networks. In *ACM on International Conference on Multimedia Retrieval*, pp. 269–277, 2017.

Mengjia Yan, Christopher W Fletcher, and Josep Torrellas. Cache telepathy: Leveraging shared resource attacks to learn DNN architectures. In *USENIX Security Symposium*, pp. 2003–2020, 2020.

Yuval Yarom and Katrina Falkner. FLUSH+ RELOAD: A high resolution, low noise, L3 cache side-channel attack. In *USENIX Security Symposium*, pp. 719–732, 2014.

Jialong Zhang, Zhongshu Gu, Jiyong Jang, Hui Wu, Marc Ph Stoecklin, Heqing Huang, and Ian Molloy. Protecting intellectual property of deep neural networks with watermarking. In *ACM Asia Conference on Computer and Communications Security*, pp. 159–172, 2018.

Jie Zhang, Dongdong Chen, Jing Liao, Han Fang, Weiming Zhang, Wenbo Zhou, Hao Cui, and Nenghai Yu. Model watermarking for image processing networks. In *AAAI Conference on Artificial Intelligence*, volume 34, pp. 12805–12812, 2020.

Barret Zoph and Quoc V Le. Neural architecture search with reinforcement learning. *arXiv preprint arXiv:1611.01578*, 2016.

Barret Zoph, Vijay Vasudevan, Jonathon Shlens, and Quoc V Le. Learning transferable architectures for scalable image recognition. In *IEEE Conference on Computer Vision and Pattern Recognition*, pp. 8697–8710, 2018.

A PROOF SKETCH OF THEOREM 1

Proof Sketch. 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 $\mathbb{S}_0$, $\mathbb{S}$ be the search spaces before and after restricting a watermark in a NAS method, $\mathbb{S}_0 \supseteq \mathbb{S}$. $\mathfrak{A}_0 \in \mathbb{S}_0$ is the optimal architecture for an arbitrary data distribution $\mathcal{D}$. $\mathfrak{A}$ is the optimal architecture in $\mathbb{S}$, The model accuracy of $\mathfrak{A}$ is no smaller than that of $\mathfrak{A}_0$ by a relaxation of $\frac{\epsilon}{N}$.*

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

Assumption 2. *Let mk and vk be the marking and verification key of a DNN architecture $\mathfrak{A} = \{c_1, ..., c_N\}$. For $\forall\ mk, vk$, and $\mathfrak{A}$, 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 Assumptions 1-2, we prove that our algorithms (**WMGen**, **Mark**, **Verify**) form a qualified watermarking scheme for NAS models.

Effectiveness. The property can be guaranteed by Assumption 2.

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

$$Pr[f_{c_i,0}(x) = y|(x, y) \sim \mathcal{D}] - Pr[f_{c_i}(x) = y|(x, y) \sim \mathcal{D}] \leq \frac{\epsilon}{N}. \quad (1)$$

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.,

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

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 $\frac{n_s}{|c_i|}$, where $|c_i|$ is the number of connected edges in c_i. Thus, the expected value of the total number of modification is $\delta \times \frac{N \times n_s}{\sum_{i=1}^{N} |c_i|}^{\tau \times N_s}$. 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. Given a watermarked model, we expect that benign users have a very low probability to obtain the same architecture following the original NAS method. 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. Assume the selection of candidate operations is independent and identically distributed, the probability that an operation is chosen on an edge is $\frac{1}{|\mathcal{O}|}$. For a DNN model that contains $\mathcal{B}$ computation nodes, there are $2\mathcal{B}$ connection edges, from which we select n_s causal edges. There are $\binom{2\mathcal{B}}{n_s}$ combinations. Hence, the probability of the stamp collision in a cell can be computed as $\binom{2\mathcal{B}}{n_s} \times (\frac{1}{|\mathcal{O}|})^{n_s}$. In our experiment configurations, the collision rate is smaller than 1.7%. Considering both the normal and reduction cells, the collision rate is smaller than $(1.7\%)^2 \approx 0.03\%$, which can be neglected.

$\square$

B GET PATH FROM CELL SUPERNET

Algorithm 4 illustrates how to extract consecutive paths from the cell *supernet* $\mathcal{G}$. The operation $\{set\} \circ \mathcal{N}_i$ appends the node $\mathcal{N}_i$ to each element in the *set*, generating a set P_i of possible paths from the cell inputs to node $\mathcal{N}_i$. Specifically, $P_{\mathcal{B}}$ contains all the candidate paths in the cell supernet $\mathcal{G}$. Given the number of fixed stamp edges n_s , our goal is to identify a path of length n_s from $\mathcal{G}$. Note that the longest consecutive path in $\mathcal{G}$ contains $\mathcal{B}$ edges, so that it has $1 \leq n_s \leq \mathcal{B}$. For each candidate path p in $P_{\mathcal{B}}$, if its length is larger than n_s, we would extract all the subpaths with length n_s from it (`GetSubPath`), and save them to S_e.

Algorithm 4: `GetPath` from Cell Supernet

```
Input: cell supernet $\mathcal{G}$, # of fixed edges $n_s$
Output: set $S_e$ of all the possible paths with
        length $n_s$
$S_e = \{\}$,
$P_1 = \{a, b\} \circ \mathcal{N}_1$
for i from 2 to $\mathcal{B}$ do
    $P_i = (P_{i-1} \cup ... \cup P_1 \cup \{a\} \cup \{b\}) \circ \mathcal{N}_i$
for p in $P_{\mathcal{B}}$ do
    if $|p| \geq n_s$ then
        $S_e = S_e \cup$ GetSubPath$(p, n_s)$
return $S_e$
```

Algorithm 5: *GEMM* in OpenBLAS

```
Input: matrice A, B, C; scalars $\alpha$, $\beta$
Output: $C = \alpha A \times B + \beta C$
for j in (0:R:n) do // Loop 1
    for l in (0:Q:k) do // Loop 2
        call itcopy
        for jj in (j:3UNROLL:j+R) do
         // Loop 4
            call oncopy
            call kernel
        for i in (P:P:m) do // Loop 3
            call itcopy
            call kernel
```

C 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 α and β are scalars. OpenBLAS adopts Goto's algorithm Goto & Geijn (2008) 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 12. 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 5 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, $loop_3$ and $loop_4$, where $loop_4$ 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.

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 α and β are scalars; (2) The low-level BLAS library performs the matrix multiplication with the GEMM algorithm (Algorithm 5). Constants of P, Q, R and $UNROLL$ are determined by the host machine configuration. Our testbed 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 C.

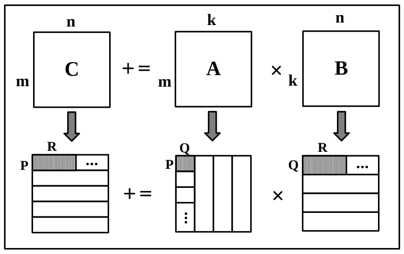


Figure 12: The procedure of GEMM.

We take three steps to recover each operation and its hyper-parameters. First, we monitor the memory accesses to the *itcopy* and *oncopy* functions in Algorithm 5, and count the number of iterations $iter_n$

for each loop n. 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 Yan et al. (2020) to derive the range of the matrix dimension (m, n, k) from $iter_n$, based on the equations: $iter_1 \equiv 1$, $iter_2 = \lceil k/Q \rceil$, $iter_3 = \lceil (m-P)/P \rceil$ and $iter_4 = \lceil n/3UNROLL \rceil$. 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. Finally, we derive the hyper-parameters of each operation based on the matrix dimension, as described below.

D Side-channel Leakage patterns of NAS Operations

Fully Connected (FC) Layer. The operation is computed as the multiplication of a learnable weight matrix θ $(m \times k)$ and an input matrix in $(k \times n)$, to generate the output matrix 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 *itcopy-oncopy* pattern, hyper-parameters (e.g., neurons number, input size) of the FC layer can be recovered.

Normal Convolution. Although this operation was adopted in earlier NAS methods Real et al. (2019); Zoph et al. (2018), recent works Liu et al. (2018b); Dong & Yang (2019); Chu et al. (2020) removed it from the search space as it is hardly used in the searched cells. A normal convolution at the i-th layer can be transformed to the multiplication of input matrix in_i $(m \times k)$ and filter matrix F_i $(k \times n)$. The matrix dimensions are: $m = (W_i - R_i + P_i + 1)(H_i - R_i + P_i + 1)$, $k = R_i^2 D_i$ and $n = D_{i+1}$, where (W_i, H_i, D_i) is the size of input tensor, P_i is the padding size, R_i is the kernel size of filters and D_{i+1} is the output channel size. In a NAS model, normal cells take the stride of 1, while reduction cells take the stride of 2.

Separable Convolution. This operation aims to achieve more efficient computation with less complexity by separating the filters. It first uses D_i filters to convolve each input channel to generate an intermediate tensor, which can be regarded as D_i normal convolutions with the same pattern. Then a 1×1 convolution with D_{i+1} filters is applied to generate the final output. 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 Zoph et al. (2018); Real et al. (2019); Liu et al. (2018a;b); Dong & Yang (2019) to improve the performance, which makes its leakage pattern more recognizable.

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, the actually used kernel size is changed to $R_i' = R_i + d(R_i - 1)$. Second, a DS convolution needs much shorter execution time.

Skip Connect. The operation is also called *identity* in the NAS search space, which just sends out_i to in_j without any processing. This operation cannot be directly detected from the side-channel leakage trace, as it does not invoke any GEMM computations. Our experiments show that while the skip connect cannot be distinguished in a CNN model, it can still be identified in an RNN model, as it results in a distinguishable blank area that shortens the cluster length (e.g., the fifth cluster in Figure 11).

Pooling. Given that pooling can reduce the size of the input matrix in_i from the last output matrix out_{i-1}, the size of the pooling layer can be obtained by performing square root over the quotient of the number of rows in out_{i-1} and in_i. Besides, pooling introduces much longer latency (nearly $1.5\times$) than the normal inter-GEMM latency. Hence, we can identify this operation by monitoring the matrix size and execution intervals. The pooling type can be revealed from the accesses on pooling functions in the deep learning framework.

Activation Functions. 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., *relu*, *sigmod*, *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.

The relationships between the operation hyper-parameters and the matrix dimensions are summarized in Table 3.

Operations	Parameters	Value	Operations	Parameters	Value
Fully Connected	C_l: # of layers	# of matrix muls	Pooling Layer	pool width/height	$\approx \sqrt{\frac{row(out_{i-1})}{row(in_i)}}$
	C_n: # of neurons	$row(\theta)$			
Operations	**D_{i+1}: Number of Filters**	**R_i: Kernel Size**	**P_i: Padding**	**Stride**	**d: Dilated Space**
Normal Conv	$col(F_i)$	$\sqrt{\frac{col(in_i)}{col(out_{i-1})}}$	$diff(row(in_i), row(out_{i-1}))$ NAS: $R_i - 1$ (non-dilated) $R_i' - 1$ (dilated), where $R_i' = R_i + d(R_i - 1)$	$\sqrt{\frac{row(out_{i-1})}{row(in_i)}}$ NAS: = 1 (normal cells) = 2 (reduction cells)	0
Dilated Conv					d
Separable Conv	Filters ①: # of same matrix muls Filters ②: $col(F_i)$	Filters ①: $\sqrt{row(F_i)}$ Filters ②: 1			0
Dil-Sep Conv					d

Table 3: Mapping between operation hyper-parameters and matrix dimensions.

E Details about the NAS Algorithms

E.1 Architecture Search

CIFAR10. We adopt GDAS Dong & Yang (2019) 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 Dong & Yang (2019). The search process on CIFAR10 takes about five hours with a single NVIDIA Tesla V100 GPU.

PTB. We adopt DARTS Liu et al. (2018b) 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 5×10^{-7}. Other setting of the optimization of the architecture is also the same as Liu et al. (2018b). The search process takes 6 hours on a single GPU.

E.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 Dong & Yang (2019). 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 8×10^{-7}. The training process takes 3 days on a single GPU.

F Monitored Functions in Pytorch and OpenBLAS

Table 4 gives the monitored lines of the code in the latest Pytorch 1.8.0 and OpenBLAS 0.3.15.

Library	Functions	Code Line
OpenBLAS	Itcopy	kernel/generic/gemm_tcopy_8.c:78
	Oncopy	kernel/x86_64/sgemm_ncopy_4_skylakex.c:57
Pytorch	Relu	aten/src/ATen/Functions.cpp:6332
	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.

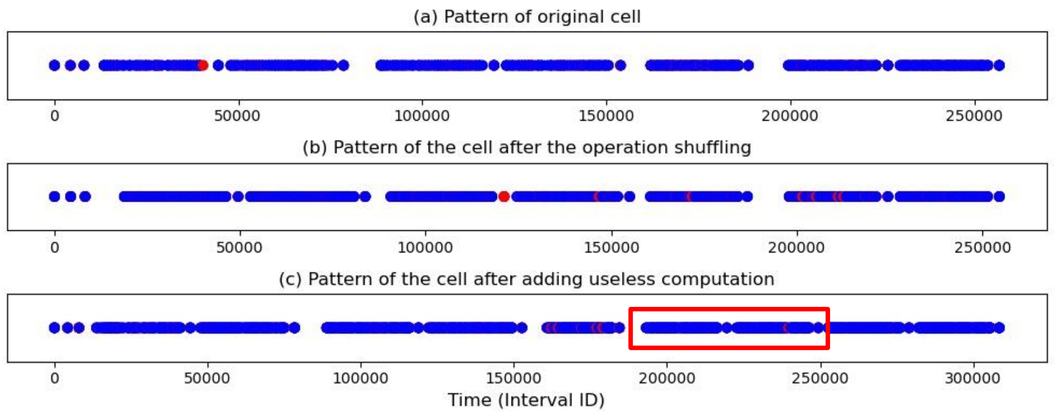


Figure 13: Side-channel patterns of cells after the model obfuscation.

G Robustness against model obfuscation

We consider two classes of model obfuscation attacks: (1) model operation shuffling and (2) useless computations injection. Take the normal cell in Figure 4a as an example, Figure 13 illustrates the leakage pattern of original cell and corresponding cells after being obfuscated by above two attacks. In Figure 13(b), the attacker shuffles the operation execution order, which first executes ②, ④, ⑥ and ⑧ in Figure 13 and then run the watermarked path. From the figure, we can see that the watermark (i.e., fixed operations) still can be identified in sequence. In Figure 13(c), the attacker add an unused 3×3 separable convolution (red block) in the pipeline, but it still does not influence us to extract the watermark, as the fixed sequence of stamp operations remains.

H Whole side channel leakage trace

Figure 14 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 identity the first block from interval 0 to around $2 \times 1e6$, 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.

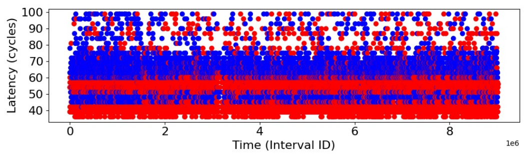


Figure 14: Whole leakage trace of the NAS model.

I Deduction of Matrix Dimensions from the Leakage Trace

Figure 15 shows the values of (m, n, k) extracted from $iter_n$ in the normal cell, where each operation contains two types of normal convolutions. For certain matrix dimensions that cannot be extracted precisely, we empirically deduce their values based on the constraints of NAS models. For instance, m is detected to be between [961, 1280]. We can fix it as $m = 1024$ since it denotes the size of input to the cell and 32×32 is the most common setting. The value of n can be easily deduced as it is equal to the channel size. Deduction of k is more difficult, since the filter size k in a NAS model is normally smaller than the constant Q in OpenBLAS, which only invokes one $loop_2$ in Algorithm 5. So we can only tell $k \leq Q$ from the trace. However, an interesting observation is that 5×5 convolution takes much longer time than 3×3 convolution, because it computes on a larger matrix. Such timing difference enables us to identify the kernel size R_i when the search space is limited. Analysis on the reduction cells is similar.

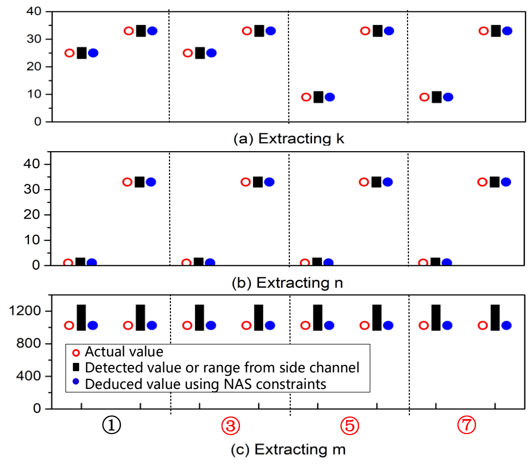


Figure 15: Extracted values of the matrix parameters (m, n, k).