This article has been accepted for publication in IEEE Transactions on Circuits and Systems for Video Technology. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TCSVT.2022.3184644

Ownership Verification of DNN Architectures via Hardware Cache Side Channels

Xiaoxuan Lou, Shangwei Guo, Jiwei Li, and Tianwei Zhang, *Member, IEEE*

***Abstract*—Deep Neural Networks (DNN) are gaining higher commercial values in computer vision applications, e.g., image classification, video analytics, etc. This calls for urgent demands of the intellectual property (IP) protection of DNN models. In this paper, we present a novel watermarking scheme to achieve the 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 claim the 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. We conduct comprehensive experiments on watermarked CNN models for image classification tasks and the experimental results show our scheme has negligible impact on the model performance, and exhibits strong robustness against various model transformations and adaptive attacks.**

***Index Terms*—Deep Neural Network, Watermarking, Cache Side Channels.**

I. 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 processing [1], [2], [3], video processing [4], [5], natural language processing [6], [7], bioinformatics [8]. 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 has made DNN models the core Intellectual Property (IP) of AI products and applications.

Release of DNN models can incur illegitimate plagiarism, unauthorized distribution or reproduction. Therefore, it is of great importance to protect the IP of such valuable assets. Similar to image watermarking [9], [10], [11], [12], [13], [14], [15], one common approach for IP protection of DNN models is DNN watermarking, which processes the protected model in a *unique* way such that its owner can recognize the ownership of his model. Existing solutions all implanted the watermarks into the parameters for ownership verification [16], [17], [18], [19], [20], [21]. The watermark also needs to guarantee satisfactory performance for the protected model. For example, Adi et al. [18] embedded backdoor images with certain trigger patterns into image classification models for IP protection.

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 [22], [23], [24] or input transformation [25] to successfully invalidate existing watermark methods. Second, watermarked models need to give unique behaviors, which inevitably make them detectable by the adversary. Some works [26], [27] introduced attacks to detect the verification samples and then manipulate the verification results.

Motivated by the above limitations, 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 [28], [29]. 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. Automated Machine Learning (AutoML) is introduced to search for architectures [30], which still needs a large amount of time, computing resources and data samples. Second, the network architecture is critical in determining the model performance. The adversary can steal an architecture and apply it to multiple tasks with different datasets, significantly improving the financial benefit. In short, “the industry considers top-performing architectures as intellectual property” [29], and “obtaining them often has high commercial value” [28]. *Therefore, 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 maliciously refining the parameters cannot tamper with the watermarks. The adversary can only *remarkably* change the network architecture with large amounts of resources and effort 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

S. Guo is the corresponding author with the College of Computer Science, Chongqing University, China. E-mail: swguo@cqu.edu.cn

X. Lou and T. Zhang are with the School of Computer Science and Engineering, Nanyang Technological University, Singapore. E-mail: xiaoxuan001@ntu.edu.sg, tianwei.zhang@ntu.edu.sg

J. Li is with Shannon.AI and Zhejiang University, China. E-mail: jiwei_li@shannonai.com

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) [30]. NAS is a very popular AutoML approach, which can automatically discover a good network architecture for a given task and dataset. A quantity of methods [31], [32], [33], [34], [35], [36] 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 V).

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. Recent works designed novel attacks to steal DNN models [28], [37], [38]. Our scheme applies such 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 [28], [39] 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 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 VI).

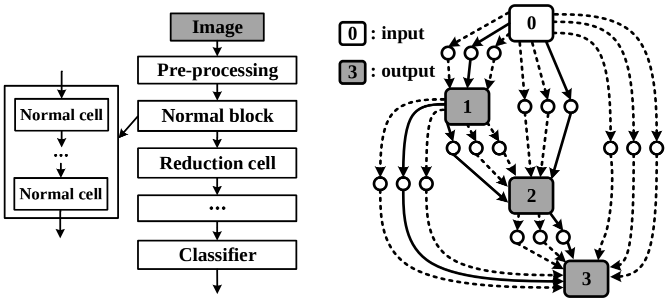
The integration of these techniques leads to the design of our watermarking framework. Experiments on DNN models for image classification show that our method is immune to common model parameter transformations (fine-tuning, pruning), which could compromise prior solutions. Furthermore, we test new adaptive attacks that moderately refine the architectures (e.g., shuffling operation order, adding useless operations), and confirm their incapability of removing the watermarks from the target architecture. In sum, we make the following contributions:

- It is the *first* work to protect the IP of DNN architectures. It creatively uses the NAS technology to embed watermarks into the model architectures.
- 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.

II. Background

A. Neural Architecture Search

NAS [30], [40] has gained popularity in recent years, due to its capability of building machine learning pipelines with high efficiency and automation. It systematically searches for good network architectures for a given task and dataset. Its effectiveness is mainly determined by two factors:



(a) CNN macro-architecture (b) A toy cell *supernet*

Fig. 1: Architecture of a NAS model based on cells

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 [31]. Then cells with the identified architecture are stacked in predefined ways to construct the final DNN models. Figure 1a shows the typical architecture of a CNN model based on the popular NAS-Bench-201 [41]. 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 (DAG), where each edge is associated with an operation selected from a predefined operation set [32]. Figure 1b gives a toy cell *supernet* that contains four computation nodes (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 methods [35], [36], [42], [43].

Search strategy. This defines the approach to seek for good architectures in the search space. Different types of strategies have been designed to enhance the search efficiency and results, based on reinforcement learning [30], [31], [32], evolutionary algorithm [44], [45] or gradient-based optimization [35], [46], [36]. Our watermarking scheme is general and independent of the search strategies.

B. 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.

This article has been accepted for publication in IEEE Transactions on Circuits and Systems for Video Technology. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TCSVT.2022.3184644

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 described as below. (1) In a PRIME-PROBE attack [47], 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. (2) A FLUSH-RELOAD attack [48] requires the adversary to share the critical memory lines with the victim, e.g., via shared library. The adversary first evicts these memory lines out of the cache using dedicated instructions (e.g., *clflush*). After a period of time, it reloads the lines into the cache and measures the access time. A shorter time indicates the lines were accessed by the victim.

C. Threat Model

We consider that a model owner designs an architecture 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 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 sorts of 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 (e.g., 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 [49], [50]), 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 [28], [37], [38] 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), e.g., Intel SGX [51] and AMD SEV [52], introduces new hardware extensions to provide execution isolation and memory encryption for user-space applications. However, an adversary can exploit TEE to hide their malicious activities, such as side-channel attacks [53], rowhammer attacks [54] and malware [55], [56]. 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 solution, the model owner can extract watermarks from the isolated enclaves. Note it has been quite common to adopt side-channel techniques to monitor the activities inside TEE enclaves for security purposes [57], [58], [59], [60], which is both technically and legitimately allowed.

III. RELATED WORKS

A. DNN Watermarking

Numerous watermarking schemes have been proposed for conventional DNN models. They can be classified into the following two categories:

White-box solutions. This strategy adopts redundant bits as watermarks and embeds them into the model parameters. For instance, Uchida et al. [16] introduced a parameter regularizer to embed a bit-vector (e.g. signature) into model parameters which can guarantee the performance of the watermarked model. Rouhan et al. [17] found that implanting watermarks into model parameters directly could affect their static properties (e.g histogram). Thus, they injected watermarks in the probability density function of the activation sets of the DNN layers. These methods require the owner to have white-box accesses to the model parameters during the watermark extraction and verification phase, which can significantly limit the possible usage scenarios.

Black-box solutions. This strategy takes a set of unique sample-label pairs as watermarks and embeds their correlation into DNN models. For examples, Le et al. [61] adopted adversarial examples near the frontiers as watermarks to identify the ownership of DNN models. Zhang et al. [62] and Adi et al. [18] employed backdoor attack techniques to embed backdoor samples with certain trigger patterns into DNN models. Namba et al. [26] and Li et al. [63] generated watermark samples that are almost indistinguishable from normal samples to avoid detection by adversaries.

Different from these works, we propose a 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.

B. 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. [39] 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 [29]. However, these attacks are very coarse-grained. They can only identify convolutions without the specific types and hyper-parameters. Yan et al. [28] 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 [28].

Other side channels. Some works leveraged other side channels to extract DNN models. Batina et al. [37] extracted a functionally equivalent model by monitoring the electromagnetic signals of a microprocessor hosting the inference program. Duddu et al. [64] 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 [38] and DNN accelerators [65]. Future work will apply those techniques to our scheme.

IV. Preliminaries

A. 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: $\mathfrak{A} = \{c_1, ..., c_N\}$. 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 $\mathcal{B}$ computing 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 $\mathcal{N}_j$ $(1 \leq j \leq B)$. Each node can only sum maximal 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 try to look for an optimal architecture $\mathfrak{A}$. The NAS method is defined as below:

Definition 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 learn the optimal parameters. The architecture $\mathfrak{A}$ and the corresponding parameters give the final DNN model $f = \texttt{train}(\mathfrak{A}, \overline{\mathcal{D}})$.

B. 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 2. *A watermarking scheme for NAS is 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 $\mathfrak{A}$, given 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 for NAS should have the following properties [62], [18].

Effectiveness. The watermarking scheme needs to guarantee the success of the ownership verification over the watermarked $\mathfrak{A}$ using the verification key. Formally,

$$Pr[\textbf{Verify}(vk, \mathbb{T}) = 1] = 1, \tag{1}$$

where $\mathbb{T}$ is the monitored side-channel trace from $\mathfrak{A}$.

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 $\mathcal{D}$, i.e.,

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

where $f = \texttt{train}(\mathfrak{A}, \overline{\mathcal{D}})$ and $f_0 = \texttt{train}(\mathfrak{A}_0, \overline{\mathcal{D}})$.

Robustness. Since a probabilistic polynomial time adversary may modify f moderately, we expect the watermark remains in $\mathfrak{A}$ after those changes. Formally, let $\mathbb{T}'$ be the side-channel leakage of a model f' transformed from f, where f' and f are from the same architecture $\mathfrak{A}$ with similar performance. We have

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

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 δ. Let $\mathbb{T}'$ be the side channel leakage of a common model learned with the same dataset and NAS method, we have

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

V. Our Watermarking Scheme

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** (Section V-A). At stage 2, he adopts a conventional NAS method with the marking key mk to produce the watermarked architecture following the algorithm **Mark** (Section V-B). 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** (Section V-C). Below we describe the details of each stage, followed by a theoretical analysis (Section V-D).

A. Watermark Generation (WMGen)

According to Definition 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. We select the edges in a path because the executions of their operations have dependency (see the red edges in Figure 6). So an adversary cannot remove the watermarks by shuffling the operation order at inference. Then the model owner fixes each of these edges with a randomly chosen operation. The

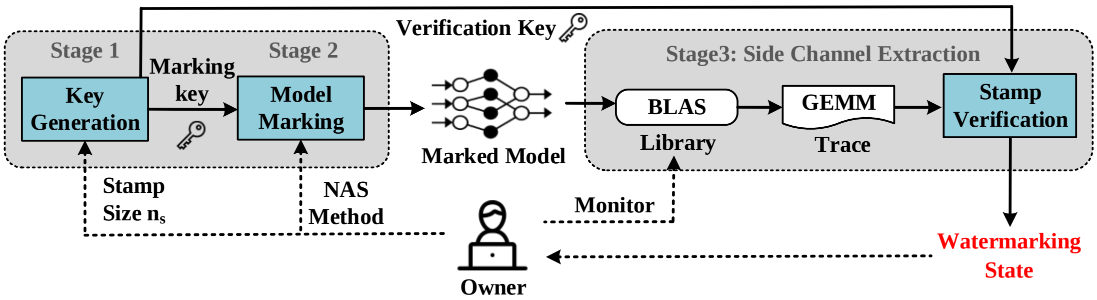


Fig. 2: Overview of our watermarking framework

```
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 $vk$
$S_e = \texttt{GetPath}(\mathcal{G}, n_s)$
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 $mk$ = ($mk_1$, ..., $mk_N$), $vk$ = ($vk_1$, ..., $vk_N$)
```

```
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($S$)
$\mathfrak{A} = \texttt{NAS}(\mathbb{S}, \mathcal{D})$
return $\mathfrak{A}$
```

set of the fixed edge-operation pairs $\{s_e : s_o\}$ inside a cell is called a *stamp*, as defined below:

Definition 3. *(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 4. *(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 B) 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. Edges in the selected path are attached with fixed operations s_o chosen by the model owner 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.

In our implementation, we randomly sample the paths and operations for the marking key. It is also possible the model owner crafts the stamps based on his own expertise. He needs to ensure the design is unique and has very small probability to conflict with other models from the same NAS method. We do not discuss this option in this paper.

B. 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 a good architecture. This process will have a smaller search space compared to the original method. However, as shown in previous works [31], [35], 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. Evaluations in Section VII 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 a good architecture $\mathfrak{A}$ containing the desired watermark.

Discussion. We describe our watermarking scheme with the NAS technique. It is worth noting that our methodology can also be applied to the hand-crafted architectures. The model owner only needs to inject the stamp edges to some locations inside his designed architecture and then train the model. We consider the evaluation of this strategy as future work.

C. Watermark Verification (Verify)

During verification, we utilize cache side channels to capture an execution trace $\mathbb{T}$ by monitoring the inference process of the

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

```
Input: verification key $vk$, monitored trace $\mathbb{T}$, # of
       fixed edges $n_s$
Output: verification result
Split $\mathbb{T}$ into cell windows
go_on = 1, verified_wins = 0
for each $window_i$ in $\mathbb{T}$ do
    if go_on == 1 then
        retrieve $s_o$ from $vk_i$, $id \leftarrow 0$
    for each $cluster$ in $window_i$ do
        if match(cluster, $s_o[id]$) = True then
            $id += 1$
    if $id == n_s$ then
        go_on = 1, verified_wins += 1
    else
        go_on = 0
return (verified_wins == vk.size()) ? True : False
```

target model M'. Details about side-channel extraction can be found in Section VI. Due to the existence of extra computations like concatenating and preprocessing, cells in $\mathbb{T}$ are separated by much larger time intervals and can be identified as sequential leakage windows. If $\mathbb{T}$ does not have observable windows, we claim it is not generated by a cell-based NAS method and is out of the consideration. 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*. Besides, since the adversary may inject useless cell windows to obfuscate the verification, we only consider cells that contain the expected side-channel patterns and skip other cells. Once the number of verified cells is equal to the size of generated verification key, we can claim the architecture ownership of the DNN model.

D. Theoretical Analysis

We theoretically prove that our algorithms (**WMGen**, **Mark**, **Verify**) form a qualified watermarking scheme for NAS architectures. We first assume the search space restricted by the watermark is still large enough for the 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 keys 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 the above two assumptions, we prove the following theorem, and the proof can be found in Appendix A.

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

VI. Side Channel Extraction

Given a suspicious model, we aim to extract the embedded watermark using cache side channels. Past works proposed cache side channel attacks to steal DNN models [28], [39]. 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.

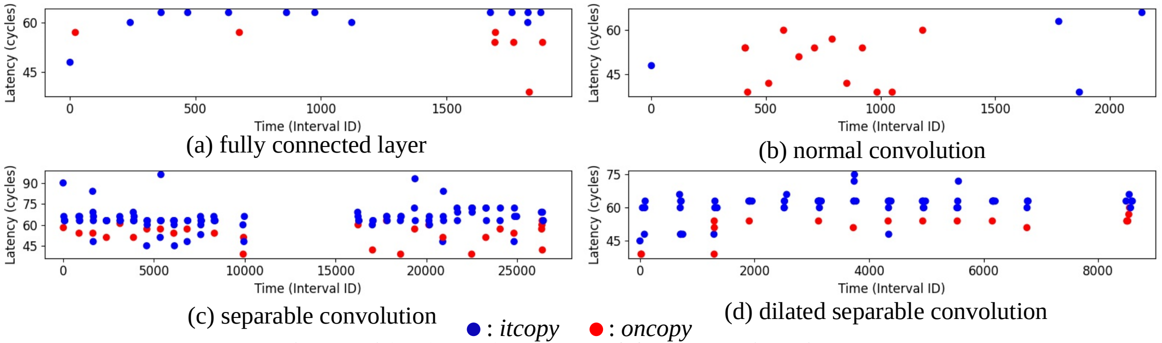
We design an improved methodology over Cache Telepathy [28] to extract the architecture of NAS models by monitoring the side-channel pattern from the BLAS library. We take OpenBLAS as an example, which is a mainstream library for many deep learning frameworks (e.g., Tensorflow, PyTorch). Our method is 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.

A. Method Overview

State-of-the-art NAS algorithms [31], [33], [35], [41] 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 functions. 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 α and β are scalars; (2) The low-level BLAS library performs the matrix multiplication with the GEMM algorithm, which divides matrix A and B into smaller ones with size of $P \times Q$ and $Q \times R$, so that they can be loaded into the cache for faster computations. Constants of P, Q and R are determined by the host machine configuration. More details about GEMM can be found in Appendix C.

Following Cache Telepathy [28], we take the *itcopy* and *oncopy* APIs in OpenBLAS as the monitoring targets. Since these two APIs are used to load matrix data into the cache, we can analyze the access pattern to them to reveal the dimension information of computing matrix. Besides, the variance of



(a) fully connected layer

(b) normal convolution

(c) separable convolution

(d) dilated separable convolution

Fig. 3: Side-channel patterns of four operations in NAS.

API access pattern also leaks the type of running operation. 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.

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

B. Recovery of NAS Operations

Fully connected (FC) layer. This operation can be transformed to 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 iteration counts of *itcopy* and *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 pattern of a classifier with two FC layers, where two separate clusters can be easily identified.

Normal convolution. Although this operation was adopted in earlier NAS methods [44], [31], recent works [35], [36], [46] removed it from the search space as it is hardly used in the searched cells. However, this operation is the basis of the following complex convolutions. So it is necessary to perform detailed analysis about it.

Figure 4 shows the structure of a normal convolution at the i-th layer (upper part), and how it is transformed to a matrix multiplication (lower part). Each patch in the input tensor is stretched as a row of matrix in_i, and each filter is stretched as a column of matrix F_i. Hence, the number of filters D_{i+1} can be recovered from the column size n of the filter matrix F_i. The kernel size R_i can be revealed from the column size $k = R_i^2 D_i$ of the matrix in_i, as we assume D_i has been obtained from the previous layer. With the recovered R_i, the padding size P_i can be inferred as the difference between

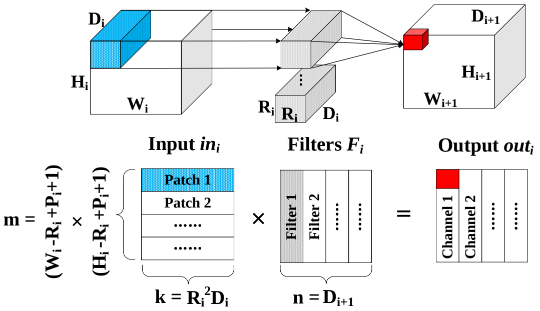


Fig. 4: Implementing a convolution operation as matrix multiplication

the row sizes of out_{i-1} and in_i, which are $W_i \times H_i$ and $(W_i - R_i + P_i + 1)(H_i - R_i + P_i + 1)$, respectively. The stride can be deduced based on the modification between the input size and output size of the convolution. In a NAS model, the convolved feature maps are padded to preserve their spatial resolution, so we have $P_i = R_i - 1$. A normal cell takes a stride of 1, while a reduction cell takes a stride of 2.

In terms of the leakage pattern, a normal convolution is hard to be distinguished from a FC layer, as both of their accesses to the *itcopy* and *oncopy* functions can be denoted as $xI - yO - zI$, where (x, y, z) indicate the repeated times of the functions, determined by the operation hyper-parameters. This is why Cache Telepathy [28] needs to know the architecture family of the target DNN to distinguish the operations. Figure 3(b) shows the leakage pattern of a normal convolution. 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_i^2, the actual size covered by the dilated kernel becomes $R_i'^2$, where $R_i' = R_i + d(R_i - 1)$. This changes the number of patches to $(W_i - R_i' + P_i + 1)(H_i - R_i' + P_i + 1)$. As a dilated convolution is normally implemented as a dilated

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	$col(F_i)$	$\sqrt{\frac{col(in_i)}{col(out_{i-1})}}$	$diff(row(in_i), row(out_{i-1}))$	$\sqrt{\frac{row(out_{i-1})}{row(in_i)}}$	0
Dilated Conv			NAS: $R_i - 1$ (non-dilated)		d
Separable Conv	Filters ①: # of same matrix muls	Filters ①: $\sqrt{row(F_i)}$	$R_i' - 1$ (dilated), where	NAS: = 1 (normal cells)	0
Dil-Sep Conv	Filters ②: $col(F_i)$	Filters ②: 1	$R_i' = R_i + d(R_i - 1)$	= 2 (reduction cells)	d

TABLE I: Mapping between operation hyper-parameters and matrix dimensions.

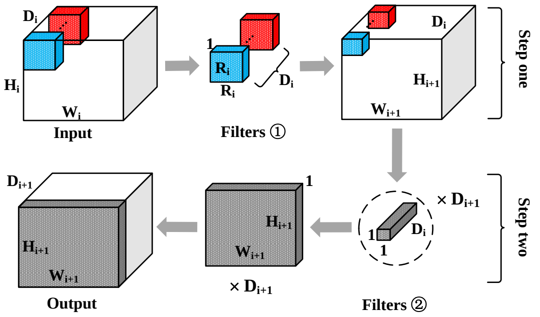


Fig. 5: Procedure of separable convolutions.

separable convolution in practical NAS methods [35], [36], the leakage pattern of the operation will be discussed with the dilated separable convolution.

Separable convolution. According to [28], 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 $R_i^2 \times 1$. 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 (Filters ②) of size $1^2 \times D_i$ 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 [31], [44], [66], [35], [36] to improve the performance, which makes its leakage pattern more recognizable. Figure 3(c) shows the trace of a separable convolution. 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 1×1 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 $R_i' = R_i + d(R_i - 1)$ in calculating the number of patches $m = (W_i - R_i + P_i + 1)(H_i - R_i + P_i + 1)$ 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 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 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. While [28] 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.

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 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. 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× 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

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 II: An example of the marking key mk.

by monitoring the corresponding pooling functions in the deep learning framework.

Other DNN components. Besides the above operations, other common components like batch normalization, dropout and activation functions are also critical to the model performance. Our method can be generalized to watermark these components as well, by just changing the monitored library targets. For instance, to protect activation functions, e.g., *relu* and *tanh*, we can monitor accesses to the corresponding function APIs in Pytorch. In Appendix E, we give more details about the monitored code lines in Table IV and an example side-channel trace of monitored activation functions in Figure 17.

VII. Evaluation

A. Experimental Setup

Testbed. Our approach is general for different deep learning frameworks and libraries. Without loss of generality, we adopt 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 [36], which can produce qualified network designs within five GPU hours. We follow the default configurations to perform NAS [31], [36]: the search space of a CNN cell contains: *identity (skip), 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)*. 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 D.

Side channel extraction. To capture the side-channel leakage of CNN models, we monitor the *itcopy* and *oncopy* functions in OpenBLAS. We adopt the Flush+Reload side-channel technique [48], 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 IV in Appendix E.

B. Effectiveness

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 $mk = (mk_n, mk_r)$, 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 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 II). The verification key $vk = (vk_n, vk_r)$ is also recorded, where $vk_n = s_{on}$ and $vk_r = s_{or}$.

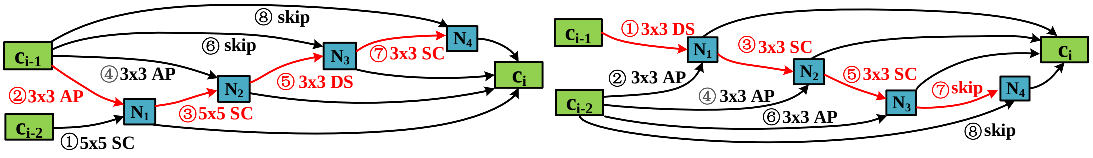
2) Watermark Embedding: We follow Algorithm 2 to embed the watermark determined by mk to the DNN architecture during the search process. Figure 6 shows the architectures of two cells searched by GDAS, where stamps are marked as red edges, and the computing order of each operation is annotated with numbers. 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 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, and the channel sizes (i.e., filter number) of three normal blocks are set as 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.

3) Watermark Extraction and Verification: Given a suspicious model, we launch a spy process to monitor the 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.

Second, we focus on the internal structure of each cell and check if it contains the fixed operation sequence given by vk. 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. Recall that in the figure the blue node denotes an access to itcopy and red node denotes an access to oncopy. From this 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 8a 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[1]. Figure 8b 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,

[1] The value of C can be identified if we zoom in Figure 7, which is not shown in this paper due to page limit.



(a) Normal cell

(b) Reduction cell

Fig. 6: Architectures of the searched cells. c_{i-1} and c_{i-2} are the inputs from the previous cells.

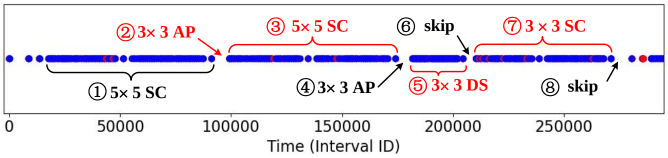
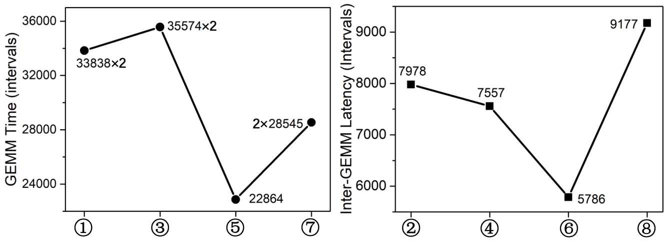


Fig. 7: A side-channel trace of the first normal cell.



(a) GEMM operations

(b) Inter-GEMM latency

Fig. 8: Execution time of the operations in a cell

after identifying the fixed operation sequences (s_o) in all cells, we can claim the architecture ownership of the DNN model.

The above analysis can already give us 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), according to Table I. Figure 9 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 equals the channel size. Deduction of k is more difficult, since the filter size k in a NAS model is normally smaller than the GEMM constant in OpenBLAS, it does not leak useful messages in the side-trace 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.

C. 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 10 shows the average results on CIFAR dataset of five experiments versus the training epochs.

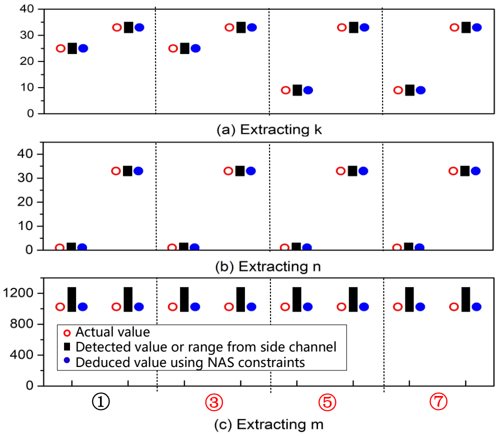
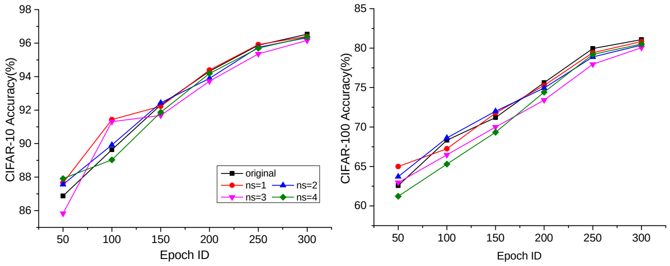


Fig. 9: Extracted values of the matrix parameters (m, n, k).



(a) CIFAR-10

(b) CIFAR-100

Fig. 10: Top-1 validation accuracy

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.

Selection of the stamp size. The setting of the stamp size is a trade-off between model usability and watermark reliability. Since our watermark scheme requires the stamp edges to form a dependent path to be robust against operation shuffling attacks (Section VII-D), the largest number of stamps is restricted by the number of nodes in the NAS cell. For conventional NAS architectures, the range of stamp size is [0, 4]. Our evaluation results (Figure 10) indicates that 4 stamp edges incur negligible performance degradation. Therefore, we recommend to adopt this setting in our watermark scheme.

D. Robustness

We consider the robustness of our watermarking scheme against four types of scenarios.

System noise. It is worth noting that the noise in the side-channel traces (e.g., from the system activities, interference with other applications) could possibly make it difficult for the model owner to identify the watermarks. To evaluate this, we follow previous works [38] to inject up to 30% scales of Gaussian noise into the time interval between events in the side-channel trace, which can well simulate the system noise. We find that it is still feasible to extract the watermarking operations with high fidelity. We conclude that the impact of system noise on operation extraction is actually negligible. The reason behind is that the most important operation features, such as the operation class, channel size and kernel size, are all revealed by analyzing the holistic pattern of side-channel leakage traces. System noise that just disturbs local patterns will not mislead the inference of these operation features. The recovery of matrix dimensions (m, n, k) is indeed affected by side-channel noise, but as we only need to deduce a range of these parameters, such impact is acceptable. Besides, according to our threat model in Section II-C, the model owner takes control of the host TEE platform, so he can disable other applications on the same machine to further improve the verification reliability.

Model transformation. Prior parameter-based solutions [18], [19], [20] are proven to be vulnerable against model fine-tuning or image transformations [22], [23], [24], [25]. 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 [18] (*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. The GEMM computations are still performed over pruned parameters, which give similar side-channel patterns. Figures 11(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 11(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. (3) The adversary may add useless cell windows to obfuscate the watermark verification.

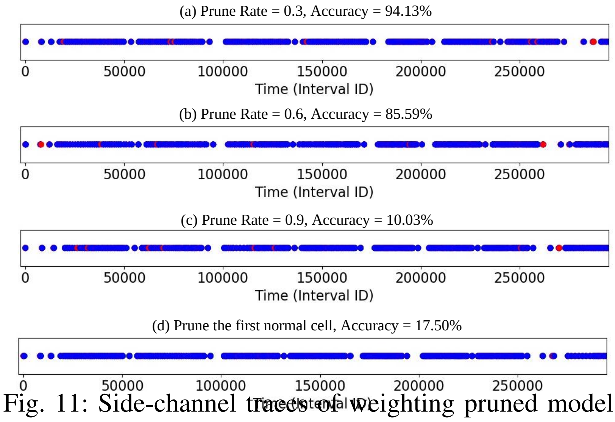


Fig. 11: Side-channel traces of weighting pruned models.

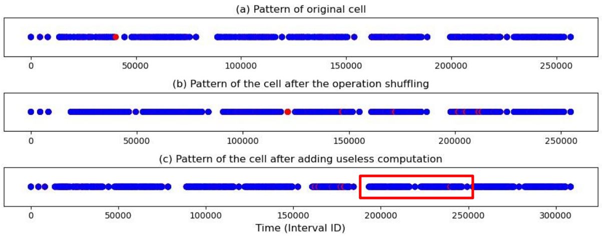


Fig. 12: Traces of obfuscated models.

Figure 12 illustrates the leakage pattern of the original cell as well as the cells after being obfuscated by above two techniques. Specifically, in Figure 12(b), the attacker shuffles the operation execution order, which first executes ②, ④, ⑥ and ⑧ and then runs the watermarked path. We can see that the watermark (i.e., fixed operations) can still be identified in the sequence. In Figure 12(c), the attacker adds an unused 3×3 separable convolution (red block) in the pipeline, which does not affect the watermark extraction, as the fixed sequence of stamp operations remains. 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.

Figure 13 shows the influence of injected useless cell windows. In the side-channel trace, it contains three cell windows, where ① and ③ are NAS cell windows and ② is the injected useless cell window. We just need to check if the monitored side-channel trace contains N identical cells and identify if the watermark exists in the cells. Even there are other cells, we can also claim that this model is watermarked and then require for further arbitration.

Structure pruning. We further consider the structured pruning, which can explicitly modify the model structure. This technique is indeed possible to remove our watermark embedded into

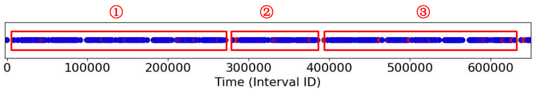


Fig. 13: Influence of useless cell windows.

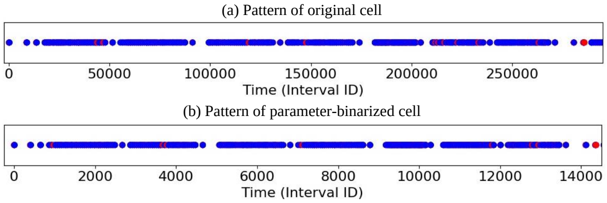
# of pruned stamp ops	0	1	2	3	4
Accuracy (%)	96.53	89.34/93.38/78.71/55.62	54.89	44.52	37.66

TABLE III: Accuracy of structured pruned models on CIFAR10

the network architecture. However, it has two drawbacks: (1) since the watermarking key is secret, the adversary does not know which operation should be pruned; (2) Pruning the stamp operation can cause significant performance drop. To validate this, we random prune 1 to 4 stamp operations in the normal cell, and Table III shows the prediction accuracy of pruned models on CIFAR10. For the case of pruning one stamp operation, we give four prediction accuracy values corresponding to four possible pruning scheme (pruning one operation from ②, ③, ⑤ and ⑦ in Fig. 6a). For models with more than one stamp operations pruned, we give the average accuracy of pruned models. We observe that even only pruning one stamp operation can lead to great accuracy drop (96.53% to 55.62%). Hence, removing the watermark with structured pruning is not practical.

Note that an adversary can leverage some powerful methods (e.g., knowledge distillation [49], [50]) 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.

Parameter binarization. This technique [67] is used to accelerate the model execution by binarizing the model parameters. If corresponding Binary Neural Network (BNN) still adopts the BLAS library to accelerate the matrix multiplications, the side-channel leakage pattern keeps similar. Only the time interval between each monitored API access becomes shorter, as the parameter binarization would cause much faster model execution. Figure 14 shows the comparison of side-channel traces between the original NAS cell and binarized cell. We observe that although parameter binarization achieves about 20 times faster inference (2.8e5 vs. 1.4e4 intervals), the leakage trace still keeps the similar pattern. Hence, our scheme can still be applied to verify BNN models. If the BNN model adopts other acceleration libraries, we can also switch to monitor that library to perform similar analysis.


(a) Pattern of original cell

(b) Pattern of parameter-binarized cell

Fig. 14: Influence of parameter binarization.

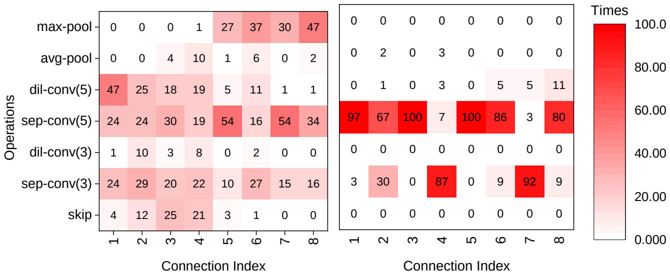


Fig. 15: Operation distributions for a normal cell (left) and reduction cell (right). The connection index is the index of the connection edge in the NAS cell.

E. Uniqueness

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.

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 15 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. 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.

VIII. Conclusion

In this paper, we propose a new direction for IP protection of DNNs. 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.

Acknowledgments

This project is in part supported by National Natural Science Foundation of China under Grants U21A20463 and 62102052, Natural Science Foundation of Chongqing, China, under Grant cstc2021jcyj-msxmX0744, Singapore National Research Foundation under its National Cybersecurity R&D Programme (NCR Award NRF2018NCR-NCR009-0001), Singapore Ministry of Education (MOE) AcRF Tier 2 MOE-T2EP20121-0006, AcRF

Tier 1 RS02/19, and NTU Start-up grant. Any opinions, findings and conclusions or recommendations expressed in this paper are those of the authors and do not reflect the views of National Research Foundation, Singapore.

References

[1] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet classification with deep convolutional neural networks," *Advances in Neural Information Processing Systems*, vol. 25, pp. 1097–1105, 2012.

[2] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *IEEE CVPR*, 2016.

[3] S. Guo, T. Zhang, G. Xu, H. Yu, T. Xiang, and Y. Liu, "Topology-aware differential privacy for decentralized image classification," *IEEE Transactions on Circuits and Systems for Video Technology*, 2021.

[4] Y. Wang, X. Fan, R. Xiong, D. Zhao, and W. Gao, "Neural network-based enhancement to inter prediction for video coding," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 32, no. 2, pp. 826–838, 2021.

[5] L. Li, Y. Zhang, S. Tang, L. Xie, X. Li, and Q. Tian, "Adaptive spatial location with balanced loss for video captioning," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 32, no. 1, pp. 17–30, 2022.

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

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

[8] A. W. Senior, R. Evans, J. Jumper, J. Kirkpatrick, L. Sifre, T. Green, C. Qin, A. Žídek, A. W. Nelson, A. Bridgland *et al.*, "Improved protein structure prediction using potentials from deep learning," *Nature*, vol. 577, no. 7792, pp. 706–710, 2020.

[9] L. Luo, Z. Chen, M. Chen, X. Zeng, and Z. Xiong, "Reversible image watermarking using interpolation technique," *IEEE Transactions on Information Forensics and Security*, vol. 5, no. 1, pp. 187–193, 2009.

[10] A. Roy and R. S. Chakraborty, "Toward optimal prediction error expansion-based reversible image watermarking," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 30, no. 8, pp. 2377–2390, 2019.

[11] H. Fang, D. Chen, Q. Huang, J. Zhang, Z. Ma, W. Zhang, and N. Yu, "Deep template-based watermarking," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 31, no. 4, pp. 1436–1451, 2020.

[12] Q. Li, X. Wang, B. Ma, X. Wang, C. Wang, S. Gao, and Y. Shi, "Concealed attack for robust watermarking based on generative model and perceptual loss," *IEEE Transactions on Circuits and Systems for Video Technology*, 2021.

[13] L. Xiong, X. Han, C.-N. Yang, and Y.-Q. Shi, "Robust reversible watermarking in encrypted image with secure multi-party based on lightweight cryptography," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 32, no. 1, pp. 75–91, 2021.

[14] J. You, Y.-G. Wang, G. Zhu, and S. Kwong, "Truncated robust natural watermarking with hungarian optimization," *IEEE Transactions on Circuits and Systems for Video Technology*, 2021.

[15] F. Peng, B. Long, and M. Long, "A general region nesting-based semi-fragile reversible watermarking for authenticating 3d mesh models," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 31, no. 11, pp. 4538–4553, 2021.

[16] Y. Uchida, Y. Nagai, S. Sakazawa, and S. Satoh, "Embedding watermarks into deep neural networks," in *ACM on International Conference on Multimedia Retrieval*, 2017, pp. 269–277.

[17] B. D. Rouhani, H. Chen, and F. Koushanfar, "DeepSigns: An end-to-end watermarking framework for protecting the ownership of deep neural networks," in *ACM ASPLOS*, 2019.

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

[19] J. Zhang, D. Chen, J. Liao, H. Fang, W. Zhang, W. Zhou, H. Cui, and N. Yu, "Model watermarking for image processing networks," in *AAAI Conference on Artificial Intelligence*, vol. 34, no. 07, 2020, pp. 12 805–12 812.

[20] K. Chen, S. Guo, T. Zhang, S. Li, and Y. Liu, "Temporal watermarks for deep reinforcement learning models," in *International Conference on Autonomous Agents and Multiagent Systems*, 2021.

[21] H. Wu, G. Liu, Y. Yao, and X. Zhang, "Watermarking neural networks with watermarked images," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 31, no. 7, pp. 2591–2601, 2020.

[22] X. Chen, W. Wang, C. Bender, Y. Ding, R. Jia, B. Li, and D. Song, "REFIT: A unified watermark removal framework for deep learning systems with limited data," *ACM AsiaCCS*, 2021.

[23] M. Shafieinejad, J. Wang, N. Lukas, X. Li, and F. Kerschbaum, "On the robustness of the backdoor-based watermarking in deep neural networks," *arXiv preprint arXiv:1906.07745*, 2019.

[24] X. Liu, F. Li, B. Wen, and Q. Li, "Removing backdoor-based watermarks in neural networks with limited data," *arXiv preprint arXiv:2008.00407*, 2020.

[25] S. Guo, T. Zhang, H. Qiu, Y. Zeng, T. Xiang, and Y. Liu, "Fine-tuning is not enough: A simple yet effective watermark removal attack for DNN models," *International Joint Conference on Artificial Intelligence*, 2021.

[26] R. Namba and J. Sakuma, "Robust watermarking of neural network with exponential weighting," in *ACM AsiaCCS*, 2019.

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

[28] M. Yan, C. W. Fletcher, and J. Torrellas, "Cache telepathy: Leveraging shared resource attacks to learn DNN architectures," in *USENIX Security Symposium*, 2020, pp. 2003–2020.

[29] S. Hong, M. Davinroy, Y. Kaya, D. Dachman-Soled, and T. Dumitraş, "How to 0wn NAS in your spare time," *arXiv preprint arXiv:2002.06776*, 2020.

[30] B. Zoph and Q. V. Le, "Neural architecture search with reinforcement learning," *arXiv preprint arXiv:1611.01578*, 2016.

[31] B. Zoph, V. Vasudevan, J. Shlens, and Q. V. Le, "Learning transferable architectures for scalable image recognition," in *IEEE CVPR*, 2018.

[32] H. Pham, M. Y. Guan, B. Zoph, Q. V. Le, and J. Dean, "Efficient neural architecture search via parameter sharing," *arXiv preprint*, 2018.

[33] H. Liu, K. Simonyan, O. Vinyals, C. Fernando, and K. Kavukcuoglu, "Hierarchical representations for efficient architecture search," *arXiv preprint arXiv:1711.00436*, 2017.

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

[35] H. Liu, K. Simonyan, and Y. Yang, "DARTS: Differentiable architecture search," *arXiv preprint arXiv:1806.09055*, 2018.

[36] X. Dong and Y. Yang, "Searching for a robust neural architecture in four GPU hours," in *IEEE Conference on Computer Vision and Pattern Recognition*, 2019.

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

[38] X. Hu, L. Liang, S. Li, L. Deng, P. Zuo, Y. Ji, X. Xie, Y. Ding, C. Liu, T. Sherwood *et al.*, "DeepSniffer: A DNN model extraction framework based on learning architectural hints," in *Architectural Support for Programming Languages and Operating Systems*, 2020.

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

[40] T. Elsken, J. H. Metzen, F. Hutter *et al.*, "Neural architecture search: A survey." *Journal of Machine Learning Research*, vol. 20, no. 55, pp. 1–21, 2019.

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

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

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

[44] E. Real, A. Aggarwal, Y. Huang, and Q. V. Le, "Regularized evolution for image classifier architecture search," in *AAAI Conference on Artificial Intelligence*, vol. 33, 2019, pp. 4780–4789.

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

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

[47] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, "Last-level cache side-channel attacks are practical," in *IEEE S&P*, 2015.

[48] Y. Yarom and K. Falkner, "FLUSH+ RELOAD: A high resolution, low noise, L3 cache side-channel attack," in *USENIX Security Symposium*, 2014.

[49] L. J. Ba and R. Caruana, "Do deep nets really need to be deep?" *arXiv preprint arXiv:1312.6184*, 2013.

[50] G. Hinton, O. Vinyals, and J. Dean, "Distilling the knowledge in a neural network," *arXiv preprint arXiv:1503.02531*, 2015.

[51] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar, "Innovative instructions and software model for isolated execution." *Hasp@ isca*, vol. 10, no. 1, 2013.

[52] D. Kaplan, J. Powell, and T. Woller, "Amd memory encryption," 2016.

[53] M. Schwarz, S. Weiser, D. Gruss, C. Maurice, and S. Mangard, "Malware guard extension: Using SGX to conceal cache attacks," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2017, pp. 3–24.

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

[55] M. Schwarz, S. Weiser, and D. Gruss, "Practical enclave malware with Intel SGX," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2019, pp. 177–196.

[56] M. Marschalek, "The wolf in SGX clothing," in *Bluehat IL*, 2018.

[57] F. Brasser, U. Müller, A. Dmitrienko, K. Kostiainen, S. Capkun, and A.-R. Sadeghi, "Software grand exposure: SGX cache attacks are practical," in *USENIX Workshop on Offensive Technologies*, 2017.

[58] J. Götzfried, M. Eckert, S. Schinzel, and T. Müller, "Cache attacks on Intel SGX," in *European Workshop on Systems Security*, 2017, pp. 1–6.

[59] M. Hähnel, W. Cui, and M. Peinado, "High-resolution side channels for untrusted operating systems," in *USENIX ATC*, 2017.

[60] L. Zhou, X. Ding, and F. Zhang, "Smile: Secure memory introspection for live enclave," in *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, 2022, pp. 1536–1536.

[61] E. Le Merrer, P. Perez, and G. Trédan, "Adversarial frontier stitching for remote neural network watermarking," *Neural Computing and Applications*, pp. 1–12, 2019.

[62] J. Zhang, Z. Gu, J. Jang, H. Wu, M. P. Stoecklin, H. Huang, and I. Molloy, "Protecting intellectual property of deep neural networks with watermarking," in *ACM AsiaCCS*, 2018.

[63] Z. Li, C. Hu, Y. Zhang, and S. Guo, "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.

[64] V. Duddu, D. Samanta, D. V. Rao, and V. E. Balas, "Stealing neural networks via timing side channels," *arXiv preprint*, 2018.

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

[66] C. Liu, B. Zoph, M. Neumann, J. Shlens, W. Hua, L.-J. Li, L. Fei-Fei, A. Yuille, J. Huang, and K. Murphy, "Progressive neural architecture search," in *European Conference on Computer Vision*, 2018, pp. 19–34.

[67] M. Courbariaux, I. Hubara, D. Soudry, R. El-Yaniv, and Y. Bengio, "Binarized neural networks: Training deep neural networks with weights and activations constrained to+ 1 or-1," *arXiv preprint arXiv:1602.02830*, 2016.

[68] K. Goto and R. A. v. d. Geijn, "Anatomy of high-performance matrix multiplication," *ACM Transactions on Mathematical Software*, vol. 34, no. 3, pp. 1–25, 2008.

Appendix

A. Proof Sketch of Theorem 1

Proof Sketch. We prove that our algorithms (**WMGen**, **Mark**, **Verify**) form a qualified watermarking scheme for NAS models.

Effectiveness. The property is 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

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

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.,

$$Pr[f_0(x)=y|(x,y)\sim\mathcal{D}] - Pr[f(x)=y|(x,y)\sim\mathcal{D}] \leq \epsilon. \tag{6}$$

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 $\frac{n_s}{|c_i|}$, 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.

□

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
          $S_e = S_e \cup \texttt{GetSubPath}(p, n_s)$
  return $S_e$
```

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 [68] 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 16. 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.

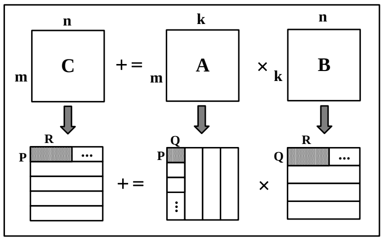


Fig. 16: The procedure of GEMM.

D. Details about the NAS Algorithms

1) Architecture Search: We adopt GDAS [36] 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 [36]. The search process on CIFAR10 takes about five hours with a single NVIDIA Tesla V100 GPU.

```
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
```

2) Model Retraining: After obtaining the searched cells, we construct the CNNs for CIFAR and ImageNet. For the CIFAR, 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 [36]. The training process takes about 11 GPU hours. 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.

E. Monitored Functions in Pytorch and OpenBLAS

Table IV gives the monitored code lines in the latest Pytorch 1.8.0 and OpenBLAS 0.3.15. To identify computationally intensive operations (i.e., convolutions), we need to monitor accesses to functions *itcopy* and *oncopy*. To protect other DNN components like activations, we turn to monitor corresponding activation APIs in the library. Figure 17 shows an example side-channel leakage trace of activation functions monitored from a NAS cell. We observe that the trace contains 9 separate clusters, each of which represents the existence of activation functions in a DNN model layer.

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 IV: Monitored code lines in OpenBLAS and Pytorch.

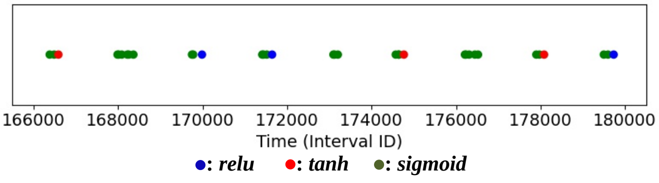
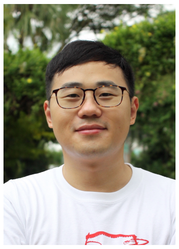
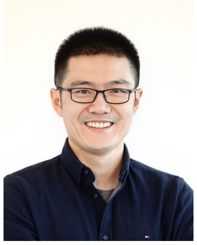


Fig. 17: The side-channel trace of a recurrent cell.

Xiaoxuan Lou received the BS degree and Master degree from Zhejiang University, China in 2017 and 2020, respectively. He is currently pursuing the PhD degree at Nanyang Technological University, Singapore. His research interests include machine learning security and system security.



Shangwei Guo is an associate professor in College of Computer Science, Chongqing University. He received the Ph.D. degree in computer science from Chongqing University, Chongqing, China in 2017. He worked as a postdoctoral research fellow at Hong Kong Baptist University and Nanyang Technological University from 2018 to 2020. His research interests include machine learning security, cloud/edge computing security, and database security.



Jiwei Li is a researcher in the Department of Computer Science and Technology of Zhejiang University, P.R. China. He received his Ph.D. degree in Computer Science from Stanford Unviersity, USA, in 2017. He has published over 60 papers on top-tier AI conferences and journals including ICLR, ICML, NeurIPS, ACL, EMNLP, NAACL, Elife, etc. He is a recipient of the 2020 MIT TR35 global innovator award, and Forbes 40 under 40.

Tianwei Zhang is an assistant professor in School of Computer Science and Engineering, at Nanyang Technological University. His research focuses on computer system security. He is particularly interested in security threats and defenses in machine learning systems, autonomous systems, computer architecture and distributed systems. He received his Bachelor's degree at Peking University in 2011, and the Ph.D. degree in at Princeton University in 2017.