DNN Latency Sequencing: Extracting DNN Architectures from Intel SGX Enclaves with Single-Stepping Attacks

Minkyung Park*, Zelun Kong*, Dave (Jing) Tian†, Z. Berkay Celik†, Chung Hwan Kim*
*University of Texas at Dallas †Purdue University
*{minkyung.park, zelun.kong, chungkim}@utdallas.edu †{daveti, zcelik}@purdue.edu

***Abstract*—Deep neural networks (DNNs) are integral to modern computing, powering applications such as image recognition, natural language processing, and audio analysis. The architectures of these models (e.g., the number and types of layers) are considered valuable intellectual property due to the significant expertise and computational effort required for their design. Although trusted execution environments (TEEs) like Intel SGX have been adopted to safeguard these models, recent studies on model extraction attacks have shown that side-channel attacks (SCAs) can still be leveraged to extract the architectures of DNN models. However, many existing model extraction attacks either do not account for TEE protections or are limited to specific model types, reducing their real-world applicability.**

In this paper, we introduce DNN Latency Sequencing (DLS), a novel model extraction attack framework that targets DNN architectures running within Intel SGX enclaves. DLS employs SGX-Step to single-step model execution and collect fine-grained latency traces, which are then analyzed at both the function and basic block levels to reconstruct the model architecture. Our key insight is that DNN architectures inherently influence execution behavior, enabling accurate reconstruction from latency patterns. We evaluate DLS on models built with three widely used deep learning libraries, Darknet, TensorFlow Lite, and ONNX Runtime, and show that it achieves architecture recovery accuracies of 97.3%, 96.4%, and 93.6%, respectively. We further demonstrate that DLS enables advanced attacks, highlighting its practicality and effectiveness.

I. Introduction

Deep learning, particularly deep neural networks (DNNs), has become foundational to a broad spectrum of applications, including image processing [1], sequence prediction [2], and audio recognition [3]. The architectures of these models (e.g., the number and types of layers, as well as various hyperparameters) are often considered valuable intellectual property, as their design demands significant human expertise and computational resources. As a result, adversaries have sought to extract proprietary DNN models and exploit this knowledge to launch *model extraction attacks*, including architecture stealing [4], [5], [6], [7], [8], [9] and parameter stealing [10], [11], [12]. These efforts, in turn, can facilitate more advanced attacks, such as membership inference [13], [14], [15] and evasion attacks [16], [17].

To protect the confidentiality of DNN models, researchers have proposed using trusted execution environments (TEEs) such as Intel SGX [18], [19], [20], [21], [22], [23], ARM TrustZone [24], [25], [26], and AMD SEV [27]. However, DNN models remain susceptible to sophisticated adversaries, particularly those exploiting side-channel attacks (SCAs), even in the presence of TEEs. Recent studies on model extraction attacks [5], [6], [7], [28], [10], [29] have shown that existing SCAs, including cache timing [30], [31], power consumption [32], and ciphertext collisions in memory [33], can be leveraged to infer information about DNN models. For clarity, we use the term *SCA primitive* to refer to an SCA technique employed for model extraction.

Although existing model extraction attacks have shown promising results, they often overlook TEE protections or rely on restrictive assumptions about the target models, limiting their practicality (Table I). First, many attacks do not assume the presence of a TEE, raising concerns about their feasibility in protected environments. For example, Intel SGX enclaves prohibit shared memory pages [4], rendering attacks such as Flush+Reload [31] ineffective. Additionally, modern Intel Xeon processors employ non-inclusive Last Level Caches (LLCs) [34], which diminishes the effectiveness of LLC-based attacks [5], [7]. Second, some attacks are limited to specific model types rather than being generally applicable. For instance, Cache Telepathy [5] can only extract models implemented using the General Matrix Multiplication (GEMM) algorithm. Or, the power-based attack within Intel SGX has been shown to work only for models using ReLU layers [10]. Third, attacks that exploit TEE vulnerabilities [29], [10] typically target model parameters and assume prior knowledge of the model architecture. As a result, a practical, generic, and effective model extraction attack to recover DNN models from within a TEE remains an open challenge.

In this paper, we present a new model extraction attack framework, namely **DNN Latency Sequencing (DLS)**, to recover the architecture of a DNN model protected by Intel SGX, without being limited to specific model types. By leveraging SGX-Step [35], we single-step the execution of the DNN model and record the latency at each step, producing a time series that serves as a unique *fingerprint* of the model.

Network and Distributed System Security (NDSS) Symposium 2026
23-27 February 2026, San Diego, CA, USA
ISBN 979-8-9919276-8-0
https://dx.doi.org/10.14722/ndss.2026.231455
www.ndss-symposium.org

Table I: Existing model extraction attacks on DNN models using SCA primitives. LLC: Last Level Cache, FR: Flush+Reload, PP: Prime+Probe, GEMM: General Matrix Multiplication.

Name	TEE Protection	SCA Primitives	Target	Target DNN Models
DeepRecon [4]	Not Considered	LLC (FR) [31]	Architecture	Any Models
Cache Telepathy [5]	Not Considered	LLC (FR & PP) [31], [30]	Architecture	Models with Known GEMM Only
DeepCache [6]	Not Considered	LLC/L1 Cache (PP) [30]	Architecture	Models with Compiler Optimization Only
GANRED [7]	Not Considered	LLC (PP) [30]	Architecture	Any Models
DeepTheft [28]	Not Considered	Power [32]	Architecture	Any Models
HyperTheft [29]	AMD SEV	Ciphertext Collision [33]	Functionality	Any Models
Zhang *et al.* [10]	Intel SGX	Power [32]	Parameters	Models with ReLU Only
DLS	**Intel SGX**	**IRQ Latency [35], [36]**	**Architecture**	**Any Models**

To bridge the semantic gap between low-level instruction latencies and high-level model architecture, DLS introduces an intermediate representation based on the program's execution behavior. The key insight is that a DNN's architecture is inherently encoded in its execution flows. Specifically, different layer types trigger distinct function calls, resulting in unique sequences of functions and, consequently, unique sequences of instructions that align with the measured latency trace. Based on this observation, we segment the latency sequence at the granularity of functions and basic blocks, and then feed these execution traces into our mapping models to infer the architecture of the target DNN model. Prior studies on SCA primitives themselves have demonstrated the feasibility of reconstructing simple execution flows for isolated functions [37], [31], [36], [38], [39], [40], [41]. In contrast, DLS addresses the unique challenges of (1) extracting the complex execution flows of DNN programs and (2) correlating these flows with the structural characteristics of DNN models, as detailed in §IV.

We implement DLS and evaluate its accuracy and effectiveness using three widely adopted deep learning libraries, Darknet [42], TensorFlow Lite [43], and ONNX Runtime [44], within Intel SGX enclaves. Our results show that the latency time series collected from randomly generated models can be used to recover model architectures with high accuracy: 97.3% for Darknet, 96.4% for TensorFlow Lite, and 93.6% for ONNX Runtime. In particular, for Darknet, using coarse-grained, function-level execution flows allows the attacker to recover the number of layers, layer types, and several key hyperparameters with 99.2% accuracy. Additionally, leveraging fine-grained, basic block-level execution flows enables the recovery of further hyperparameters with an average accuracy of 90.9%. To highlight the practical impact of DLS, we also demonstrate it to facilitate evasion attacks.

The main contributions of DLS are as follows:

- **A Novel Model Extraction Framework:** We propose DLS, a new attack framework for recovering the architecture of DNN models running inside Intel SGX enclaves. Unlike prior model extraction attacks on TEEs, DLS does not require prior knowledge of the target architecture or rely on specific DNN types.
- **Bridging the Semantic Gap:** DLS bridges the gap between high-level model architecture and low-level instruction latencies by partitioning the latency sequence at both the function and basic block levels. These execution flows are then analyzed to reconstruct the target model architecture, effectively closing the semantic gap.
- **Implementation and Evaluation:** We apply DLS on various models built using three well-known deep learning libraries, Darknet, TensorFlow Lite, and ONNX Runtime within SGX enclaves. Our experiments show that DLS can accurately reconstruct the model architecture and potentially facilitate further advanced attacks. To support reproducibility and future research, we release our code and datasets publicly [45].

II. Background

A. DNN Model Extraction Attacks

A model extraction attack compromises the privacy of a trained DNN model by extracting details of the model and reconstructing a similar or identical model [46]. These details include the *model architecture* and *model parameters*, the fundamental components of a DNN model that determine the model's performance. Model architecture is decided before model training and includes the *network structure* and *hyperparameters*. Specifically, the network structure represents the overall shape of the DNN, determined by the number of layers, the type of each layer, and their connections. Hyperparameters are configurable parameters of each layer, including activation functions, the number of filters, and filter sizes. In contrast to model architecture, model parameters include weights and biases that are optimized during the model training.

To extract the model architecture, an attacker typically sends queries to the trained model, treating the model as a black box, and observes the model's execution through various channels that it has access to (e.g., query result and side channels). Since the model parameters are contingent on the model architecture, a prerequisite step of the attack is learning the model architecture first, which can then be used to steal the model parameters [10], [11], [12].

The model architecture is sensitive information and should be kept private, as its exposure can compromise the security and privacy of a DNN. An adversary can directly exploit the architectural knowledge to launch various attacks, such as

parameter extraction [11], membership inference [13], [14], [15], and input data extraction [47]. Moreover, it can even increase the success rates of functionality extraction [29] and evasion attacks [16], [17]. Therefore, DLS focuses on the extraction of model architecture.

Side-channel-based Model Extractions Attacks. A wide range of well-established SCA primitives have been shown to leak internal information from computing systems. These primitives primarily focus on identifying and exploiting novel side channels under diverse (micro)architectural properties. Also, they demonstrate their effectiveness in controlled environments by evaluating isolated functions that contain specific secret-dependent branches [37], [36], [38], [39], [40], [41].

Building on these SCA primitives, recent research has proposed model extraction attacks, which aim to reconstruct complex DNN models. In particular, when the target model is protected by Intel SGX, existing model extraction attacks are limited to those leveraging cache-based [5], [6], [7], [4], software-based power [28], [10], and ciphertext-based [29] side channels, as summarized in Table I. For example, model extraction techniques that rely on hardware-specific features of ASICs or GPUs are not applicable under our threat model, as detailed in §VII. Several of these approaches [5], [6], [7], [4], [28] were not initially designed for TEE-protected models and their effectiveness has been demonstrated in standard CPU environments.

Attacks leveraging CPU caches typically exploit Prime+Probe [30] or Flush+Reload [31]. Since Flush+Reload relies on shared memory, which is forbidden in enclave applications, DeepRecon [4], which depends only on Flush+Reload, cannot be applied to SGX-protected models. Cache Telepathy [5] and DeepCache [6] infer matrix dimensions by observing cache operations optimized by Goto's algorithm [48] and deep learning compilers [49], [50], respectively. The matrix operations are further mapped to model architectures. The effectiveness of these attacks decreases when implementations change, as they rely on specific optimizations. GANRED [7] leverages a Generative Adversarial Network (GAN) where the discriminator compares the victim's LLC cache trace against a candidate trace. For the first layer, it evaluates all potential model architectures to find a match between the two traces. This process is then repeated for each subsequent layer until reaching the final layer. GANRED assumes that the victim model uses a predefined set of model architectures. It limits its applicability when the victim model employs a hyperparameter not in this set. Also, the attacks based on inclusive LLC cache [5], [7] would be limited when targeting Intel Xeon processors with non-inclusive LLC [34].

Intel Running Average Power Limit (RAPL) interface allows an adversary to analyze fine-grained power consumption patterns to extract model architecture [28] or parameters [10]. However, due to these security implications, recent microcode updates obfuscate power information when Intel SGX is enabled, hindering the feasibility of such attacks.

HyperTheft [29] aims to recover model functionality via surrogate training, rather than reconstructing the exact model architecture or parameters. It targets AMD SEV by exploiting ciphertext leakage [51], [33] resulting from deterministic encryption in AES-XEX mode. In terms of generality, the reliance on ciphertext leakage limits its applicability to AMD architectures. Memory Encryption Engine (MEE) in Intel SGX for single-socket servers employs a non-deterministic AES mode [52], [53]; for multi-socket servers, Intel Multi-Key Total Memory Encryption (TME-MK) relies on deterministic AES-XTS mode, but how this is integrated with Intel SGX remains unclear [54], [55]. Also, since ciphertexts in Intel SGX are inaccessible even with root privileges, alternative approaches (e.g., memory bus snooping [56] or cold-boot attacks [57]) are required to observe them. However, these methods introduce additional challenges, including the need for physical access and susceptibility to noise. Moreover, HyperTheft leverages the CipherLeaks primitive [51], which can be mitigated through the microcode patch [33]. Lastly, HyperTheft achieves higher accuracy when the architecture knowledge is available. Therefore, DLS can complement it and enhance its overall effectiveness.

B. Single-Stepping Attacks on SGX Enclaves

Among various side channels on Intel SGX, SGX-Step [35] provides a fine-grained side-channel attack framework, allowing the attackers to single-step enclave execution. It enables a user-level application to trigger an APIC timer interrupt on a target enclave. When the interrupt is fired, an Asynchronous Enclave eXit (AEX) routine is triggered, causing the enclave to exit temporarily. The enclave resumes the execution after the interrupt is handled by an interrupt request (IRQ) handler. An attacker can control this to trigger an interrupt repeatedly at a high CPU frequency and establish a channel to single-step the enclave forcefully at a fine granularity (i.e., a single CPU instruction). This capability has been leveraged by various attacks to leak sensitive information about enclave programs, such as page table entries [58], [59], [60], [61] and instruction latency [36], [39]. Note that these attacks have also been ported to other TEEs, as detailed in §VIII.

We leverage SGX-Step to single-step the execution of a DNN model within an enclave and measure the IRQ latency of individual instructions. Similar to prior work [36], we record the elapsed time between subsequent enclave resumption and exit, and use the time to reflect the latency of each instruction. Although the single stepping generally provides a low-noise side channel, there are still various sources of measurement noises that it suffers from. For example, multiple instructions may be executed within a single measurement period (multi-stepping) or no instruction may be executed (zero-stepping), due to inconsistent duration of IRQ handling, privilege level switches, and cache pollution [62], [63]. Therefore, extracting the model architecture of a DNN model within an enclave should be robust against such noise in order to perform a successful model extraction attack.

III. Threat Model

DLS considers a scenario where a victim DNN model is pre-trained and subsequently deployed on an untrusted third-party environment, such as edge devices or MLaaS platforms. While the model can be trained on any platform (e.g., GPU), the deployment environment is assumed to be equipped with Intel SGX. In this setup, the victim DNN model, represented by a victim DNN program, is protected by SGX so that the inference process operates only within an enclave. An attacker's goal is to learn the architecture of the victim model.

Our threat model aligns with the standard SGX threat model. A privileged adversary has full control over privileged system software, such as the OS or hypervisor. For example, untrusted cloud providers may access their hosting servers. Alternatively, an attacker can gain direct access to an edge device by rooting it. In addition, the victim model is regarded as a black box. The adversary is capable of initiating the inference process (repeatedly) on the victim model. For instance, open APIs provided by MLaaS platforms enable the attacker to submit queries to request inference. Lastly, it is assumed that the victim model is developed using an open-source deep learning framework, and the attacker is aware of the library with its version used to develop the model and its binary running inside the enclave for analysis.

Under this threat model, the adversary leverages the single-stepping attack to measure instruction latencies at the inference process, as explained in §II-B. The adversary then uses the latency trace for its analysis. We note that the attack does not require physical access to the victim machine nor prior knowledge about the victim model. The measurement can be conducted remotely without physical access.

IV. Challenges

The goal of DLS is to learn the execution flows of a DNN program. We define two types of execution flows: the function-level execution flow (FEF) and the basic block-level execution flow (BBEF). The FEF represents a sequence of executed functions, and the BBEF represents a sequence of executed basic blocks.

Instruction latencies are obtained from the executed instructions of the victim DNN program, and thus the latency trace reflects the program behavior, such as called functions and taken branches. On the other hand, the model architecture is a semantic representation of the program whose behavior is decided by the model architecture. Consequently, the FEF and BBEF, which provide information on program execution behavior, bridge the gap between the low-level instruction latency and the high-level model architecture.

To correlate latencies with execution flows, several technical challenges arise due to instruction latency characteristics.

C1: Indistinguishability of a Latency. Firstly, an individual instruction latency is indistinguishable. Since each instruction is typically processed within a few cycles, the latency falls within a narrow range, making it difficult to reliably identify specific instructions based solely on one latency value. To tackle this, DLS analyzes a series of latencies to capture the sequential context within the trace.

C2: Excessive Trace Length. A subsequent challenge is the impracticality of learning the sequential context across an entire trace. A DNN model can generate an enormous volume of instructions, resulting in an excessively long trace. Thus, DLS adopts a two-step approach to narrow down the problem incrementally. Initially, the latencies are transformed into a coarse-grained FEF using short-term contexts. It is subsequently refined into a fine-grained BBEF. It deduces a latency's corresponding basic block by considering the full context of a given function.

C3: Noise in Measurement. Lastly, the noise inherent in latency measurement further complicates the accurate identification of the execution flows. As mentioned, the latency is influenced by various factors such as processor microcode, memory transfer speeds, preceding or succeeding instructions, memory or cache status, and speculative execution. Also, a single latency measurement may encompass multiple instructions. Thus, its measurements may fluctuate with each execution.

In this paper, we address the above challenges to correlate the latencies with execution flows. DLS learns distinct latency patterns within each function and across multiple functions, which captures the sequential context for challenge `C1`. A specific series of instruction produces a unique latency pattern, which can be used to identify functions within the trace over short time intervals. Once functions are identified and the FEF is generated, basic blocks are subsequently recognized. To achieve this, DLS extracts latency patterns for basic blocks within a function, learns the sequential relationships between these patterns, and models how they appear throughout the function. This model is then used to estimate the sequence of basic blocks (i.e., BBEF) that best aligns with the trace. This two-step approach can address challenge `C2`. Additionally, to overcome challenge `C3`, DLS reduces noise by averaging latency measurements across multiple traces based on page index, enhancing the stability and accuracy of the attack.

Our Contributions Beyond SCA Primitives. SCA primitives typically explore new side channels under varying threat models or (micro)architectural properties. While these attacks often exploit secret-dependent branches to demonstrate feasibility, they primarily evaluate isolated functions (e.g., an inverse cosine discrete transform function [37] or a binary search function [36]) in controlled environments. As a result, key questions remain when applying such SCA primitives to model extraction attacks: Which real-world DNN applications are actually vulnerable, and why? How can side-channel information be effectively processed for realistic DNN workloads? What challenges arise in applying these techniques?

DLS is designed to address these questions. We first observe that model architectures can be reflected in side-channel traces as distinctive execution flows. Then, we design DLS to tackle the above technical challenges (`C1`–`C3`), which arise in practical model extraction. For example, simple latency-pattern matching is insufficient, as certain patterns may recur across

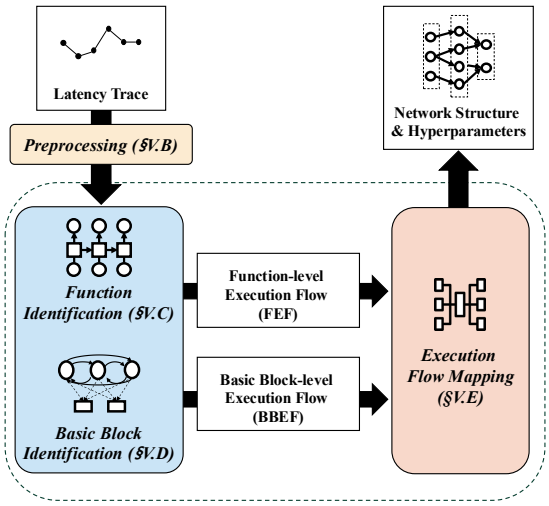


Figure 1: Overview of DLS.

Page Index	Latency	Page Index	Latency	Page Index	Latency	Page Index	Latency
P1	9273	P1	9733	P1	9317	P1	9441
P1	8675	P1	8955	P1	8637	P1	8756
P1	8978	P1	8537	P1	9013	P1	8843
P1	9978	P1	9426	P1	9103	P1	9502
P2	8464	P2	8856	P2	8102	P2	8660
P2	8734	P2	8954	P2	8823	P2	8844
P2	9524	P2	9462	P3	9302	P2	9493
P3	9245	P3	9215	P3	9103	P3	9259
P3	9163	P3	9145	P3	9045	P3	9124
P3	8967	P3	8964	P3	9013	P3	9005
P3	8953	P3	9045	P3	9001	P3	9029
P1	9267	P3	9134	1	9352	P4	9068
P1	8998	P1	8999	1	8892	P1	...
...	...	...	...	...	...	...	...
Trace T1		Trace T2		Trace T3		Averaged Trace	

Figure 2: Latency traces of DLS. Multiple traces are averaged based on their page index. The rows with the same color are averaged while the white-colored rows are ignored.

multiple functions throughout an excessively long trace. DLS incorporates more sophisticated mechanisms to disambiguate and extract meaningful structural information; we provide further details in §V. This makes DLS fundamentally different from SCA primitives themselves, enabling scalable and robust model extraction in real-world DNN scenarios.

V. Design

A. Overview

Figure 1 shows the process of DLS, which consists of four steps: (i) Preprocessing, (ii) Function Identification, (iii) Basic Block Identification, and (iv) Execution Flow Mapping. Given a victim latency trace, the attacker first preprocesses the trace, which smooths out noise. Then, function and basic block identifications extract FEF and BBEF. These flows are subsequently mapped to specific layer types or hyperparameters.

For function and basic block identification, the attacker must prepare training traces and build a Latency to Function classifier (L2F classifier) and a Latency to Basic Block model (L2BB model) in advance. Specifically, the attacker learns the sequential context of latencies to identify each function, with the L2F Classifier categorizing each latency into its corresponding function. To achieve this, DLS employs a CNN-BiLSTM (a Convolutional Neural Network with a Bidirectional Long Short-Term Memory network) network to extract features from latencies and learn their sequential dependencies. The function-classified trace is then reduced to derive the FEF. For the basic block identification, the attacker builds the L2BB model to represent the relationship between latencies and the BBEF using a Hidden Semi-Markov Model (HSMM). This model is used to capture the basic blocks (i.e., hidden states), their control flows (i.e., transitions between them), and the latencies (i.e., observations).

Execution flow mapping information is obtained through a semi-automated process by analyzing the target library. The attacker can extract common layer types and hyperparameters from training traces with the same FEF, and this process can be automated. To identify the BBEF mapping information, the attacker has to determine relationships between basic blocks and hyperparameters. We observe that model hyperparameters often affect program control flow; for example, the number of loop iterations is usually determined by the number of filters and their sizes in a convolutional layer [9]. Thus, the attacker needs to understand how hyperparameters influence the control flow of a function. Note that this mapping process only needs to be performed once per target library.

B. Data Preparation

The initial process is to collect a training dataset of latency traces. Because this dataset is used to train the L2F classifier and the L2BB model, it must comprehensively cover all functions exercised by the victim DNN program. In our evaluation, we used 300 models that is used to yield approximately 1,000 segments per class for the L2F classifier, ensuring redundant coverage of diverse execution paths and variations in latency patterns. Moreover, to mitigate noise impact, multiple traces (e.g., ten traces) are averaged for each model. Since individual traces may unexpectedly omit instructions, the attacker uses page index information to average the traces effectively. For each latency measurement, the attacker checks the index of the accessed page by examining the page table entry. Within each sequence of consecutive page identifiers, the attacker counts the number of latencies, identifies the most frequent count, and averages latencies with that count. For example, as illustrated in Figure 2, trace `T1` and trace `T2` have three instructions in page `P2`, while trace `T3` has only two. Then, only traces `T1` and `T2` are averaged for page `P2`. While this method cannot completely eliminate inaccuracies from averaging different instructions' latencies, it helps minimize their impact.

Finally, we normalize the averaged trace using Z-normalization to standardize the mean to zero and the standard deviation to one. This normalization ensures a consistent data scale across all traces and effectively handles outliers with significant noise. Consequently, the attacker collects normalized and averaged training traces from multiple models in advance. The attacker preprocesses the victim trace in the same way,

by averaging and normalizing it. For the sake of brevity, we use a 'trace' to refer to a normalized and averaged trace.

C. Function Identification

The next step involves determining the specific function associated with each instruction latency. Due to the inherent characteristics introduced in §IV, the L2F classifier analyzes consecutive instruction latencies to identify unique patterns embedded within functions, thus recognizing the corresponding functions.

We design the L2F classifier as a hybrid CNN-BiLSTM model that processes a trace and classifies each latency into its respective function. This model combines a Convolutional Neural Network (CNN) model with a Bidirectional Long Short Term Memory (BiLSTM) model. The CNN serves as an efficient mechanism to automatically learn distinctive features (patterns) in input data, while the BiLSTM captures the temporal dependencies within the input data. In the CNN-BiLSTM model, the CNN first extracts patterns from the input, and the extracted patterns are interpreted across time steps by the BiLSTM.

When the L2F classifier is trained, the CNN learns the features of functions from latencies in the training traces. The BiLSTM then captures the temporal and bidirectional dependencies across the features. The detailed architecture of the L2F classifier can be empirically optimized. Our evaluation consists of three convolutional layers with padding and two BiLSTM layers, followed by one fully connected layer. We observe that short subsequences of instructions (e.g., fewer than five) lack unique patterns, resulting in high similarity with other subsequences. Thus, to identify meaningful patterns, we set the kernel size of the CNN model to more than 16 and the number of filters to exceed the number of distinct functions.

To train the L2F classifier, each latency should be labeled with a function name, which can be obtained from the function symbols in an enclave's shared object file. However, rather than using the function name directly, DLS may merge multiple functions into a single label (pattern-based merge and depth-based merge) or split a function into multiple labels to enhance accuracy. Although the CNN-BiLSTM classifier is designed to learn unique patterns for each function, not all functions exhibit unique patterns. For example, commonly occurring patterns, such as iterative value assignments, may appear in multiple functions. In such cases, functions that share the same patterns are merged into a group and labeled as such.

Additionally, functions are merged based on call depth. A function often invokes other functions, which collectively characterize it, as the instructions of the called functions help in identifying the primary function. Therefore, if a function's call depth exceeds a predefined threshold, it is labeled with the name of its nearest parent function within the threshold. This depth threshold should be determined based on the characteristics of the target library.

Similarly, a function can be divided into multiple labels based on its operational branches. Consider the following code snippet from Darknet.

```
float activate(float x, ACTIVATION a)
{
    switch(a){
        case LINEAR:
            return x;
        case LOGISTIC:
            return 1./(1. + exp(-x));
        ...
}
```

The `activate` function takes different branches depending on the activation function type. Even though all activation functions (e.g., linear, logistic, *etc*.) are implemented within one function (i.e., `activate`), the execution patterns vary by branch. In this case, we label branches of `activate` with additional information, such as `activate-LINEAR` and `activate-LOGISTIC`. It aids in identifying the activation function type through the FEF and improves classification accuracy, especially when a specific branch shares a pattern with other functions.

The labels impact the accuracy of the L2F classifier, but determining the correct labels initially is challenging. Therefore, we incrementally update the labels as follows. Initially, DLS applies a depth-based merge and trains the L2F classifier with these labels. If latencies associated with a function are frequently misclassified as another, adjustments are made by either merging or diverging the labels for that function. After the label update, the L2F model is retrained, and this process is repeated until high accuracy is achieved. For the sake of exposition, we use 'function' to refer to one or more functions that share the same label.

Given the extensive length of the trace, we divide it into fixed-size segments (e.g., 500 instructions to avoid gradient vanishing), which facilitates a more manageable analysis. Additionally, to address the class imbalance, frequently occurring classes, such as matrix multiplication in the convolutional layer, are downsampled to balance the dataset.

When the attack is launched, the victim trace is segmented and processed by the L2F classifier. Its output, where each instruction is labeled, is then reduced by eliminating consecutive identical functions, as shown in the following example:

$$(\mathbf{f_1},\ ...,\ \mathbf{f_1},\ \mathbf{f_2},\ ...,\ \mathbf{f_2},\ \mathbf{f_3},\ ...) \rightarrow (\mathbf{f_1},\ \mathbf{f_2},\ \mathbf{f_3},\ ...)$$

This reduced sequence is then segmented by layer, resulting in an FEF for each layer, by identifying functions that demarcate a boundary of layers. For instance, the code snippet below from the Darknet library delineates a boundary function:

```
void forward_network(network *netp)
{
    ...
    for(i = 0; i < net.n; ++i){
        layer l = net.layers[i];
        l.forward(l, net);
        ...
    }
    ...
}
```

In this instance, the for loop iteratively invokes the forward function of each layer (i.e., `l.forward`), identifying "`forward_function`" as the boundary function. At this

stage, the attacker can infer the number of layers and obtain an FEF for each layer.

D. Basic Block Identification

To identify the basic block of each instruction and construct a BBEF for each function, we build an L2BB model by extending the Hidden Semi-Markov Model (HSMM). The HSMM, similar to the Hidden Markov Model (HMM), represents a system with hidden states, where state transitions are governed by a Markov process, and each state produces observable outputs. In a standard HMM, the system transitions from one state to another with each step, associating observations with the current state. On the other hand, the HSMM models state duration, which allows each state to persist for a designated number of time steps before transitioning.

The concept of duration in the HSMM aligns well with modeling basic blocks. In a given function, basic blocks, which serve as hidden states, are executed inside the enclave, and thus they are hidden from the attacker. Still, each basic block, composed of multiple instructions, produces a sequence of instruction latencies observable to the attacker. The number of these latencies corresponds to the duration of the current basic block. Once this duration is complete, another basic block executes, which represents the state transition (or, control transition from one basic block to another). On the other hand, there exists a difference between the original HSMM and the L2BB model due to the control flow of basic blocks. Since each basic block transition depends on the control flow dictated by the program, we approximate these transitions by applying constraint-based modeling [64] where transition probabilities are conditioned on predefined control paths.

The L2BB model is defined by the following parameters:

- The set of hidden states S corresponds to the basic blocks. $S_{[t_1:t_2]} = i$ means that the state i stays in the period from t_1 to t_2. $S_{[t:]} = i$ or $S_{[:t]}$ represents that the state i starts or ends at time t, respectively.
- The end state E is a subset of hidden states S. It represents the last basic blocks in the function.
- Each basic block emits a series of observations O, which are latencies.
- Initial probability is denoted by π. $\pi_{i,j}$ is set to 1 if the state i is the entry basic block and j is its length (i.e., the number of its instructions). It is set to 0, otherwise.
- We denote the *constrained* state transition probability from state i to state j as

$$a_{ij} = P[S_{[t+1:]} = j | S_{[:t]} = i].$$

  If a control flow exists from basic block i and basic block j, then $a_{ij} = 1/N$ where N is the number of hidden states. Otherwise, $a_{ij} = 0$ when no direct control flow exists between basic block i and j.
- The constrained transition probability from state i having duration h to state j having duration d can be expressed as

$$a_{(i,h)(j,d)} = P[S_{[t+1:t+d]} = j | S_{[t-h+1:t]} = i].$$

  $a_{(i,h)(j,d)} = 0$ if the length of basic block i does not equal to h or the length of basic block j does not equal to d. Otherwise, it is a_{ij}.
- The state duration D is a set of lengths of basic blocks in S.
- The reference state K is a set of ground truth latency sequences of S. K_i is the reference sequence for the basic block i.
- The emission probability, denoted as b, represents the probability of state j producing d observed latencies.

$$b_{j,d}(o_{[t+1:t+d]}) = P[o_{[t+1:t+d]} | S_{[t+1:t+d]} = j].$$

  We design the emission probability as the normalized similarity between the observation sequence $o_{[t+1:t+d]}$ and the reference sequence K_j where d is the length of the basic block j. Consequently, the greater the similarity between the observation sequence and the basic block's reference sequence, the higher the likelihood that the latency sequence originates from that basic block.

To construct the L2BB model, the attacker analyzes the training traces to extract hidden states S, end states S, initial probability π, constrained transition probability a, and their reference sequences K. The hidden states S are identified by analyzing unique instruction sequences within the training traces, with each sequence treated as an individual state. To get details such as an instruction pointer (RIP) corresponding to latency, the attacker can utilize Intel SGX's debugging capabilities. This includes sequences that omit latencies during the measurement, which are treated as separate states to capture potential measurement variances. Also, if a state contains only a few latencies, it may not exhibit discernible patterns, potentially impacting the reliability of the emission probability. To avoid it, the attacker aggregates consecutive basic blocks into a single state to ensure that each state possesses a sufficient number of latencies. Thus, while a state might not directly represent a single basic block, we use this term for the sake of exposition. The reference sequence K_i is calculated by averaging the latency values at each time step across corresponding sequences in the training trace. The constrained transition probability a is determined by examining valid transitions in the training traces. If a state transition from state i to state j exists, a_{ij} is set to $1/N$; otherwise, it is set to 0.

Once the L2BB model for a given function is built, the attacker identifies the BBEF of a victim trace by applying an extended Viterbi algorithm [65]. The details of this algorithm are provided in Algorithm 1 where the emission probability b is computed with the normalized similarity function $NormSim$ and the Euclidean distance EuD as follows:

$$b_{j,d}(o_{[t+1:t+d]}) = NormSim(EuD(o[t+1:t+d], K_s))$$

In our evaluation, although we employ a logistic function to normalize the emission probabilities, alternative methods can also be used. Finally, this process enables the attacker to obtain the most similar state sequence, which is the BBEF.

Algorithm 1: Algorithm for the basic block identification. $len(s)$ represents the number of latencies in a given state s.

Input: HSMM $(S, E, O, K, \pi, a, b, D, C)$
Output: Most likely state sequence $Q = (q_1, q_2, \ldots, q_T)$

foreach *state* $s \in S$ **do**
  foreach *duration* $dinD$ **do**
    $\delta_1(s,d) = \pi_{s,d} \cdot b_{s,d}(o_{[1:d]})$;
    $\Psi_1(s,d) = 0$;
for $t = 2$ **to** T **do**
  foreach *state* $s \in S$ **do**
    foreach *duration* $d \in D$ **do**
      $\delta_{t(s,d)} = \max_{s' \in S, h \in D}[\delta_{t-h}(s',h) \cdot a_{s's} \cdot b_s(o_{t:t+d-1})]$;
      $(s'^*, h^*) = \arg\max_{s' \in S, h \in D}[\delta_{t-h}(s',h) \cdot a_{s's} \cdot b_s(o_{t:t+d-1})]$;
      $\Psi_t(s,d) = (t - h^*, s'^*, h^*)$;
$j^* = \arg\max_{j \in E}[\delta_{t-len(j)}(s,d)]$;
$d^* = len(j^*)$;
$t^* = T - len(j^*)$;
$q_1 = j^*$;
$i = 1$;
repeat
  $i = i + 1$;
  $(t^*, j^*, d^*) = \Psi_{t^*-len(j^*)}(j^*, d^*)$;
  $q_i = j^*$;
until $t^* >= 0$;
return BBEF $Q = q_i, q_{i-1}, ..., q_1$

E. Execution Flow Mapping

To reconstruct the model architecture of the victim model, we establish a mapping between execution flows and specific layer types or hyperparameters.

1) FEF Mapping: Through our observation, we find that different layers are implemented using unique combinations of functions and some hyperparameters also impact the invoked functions. To create the FEF mapping information, the attacker first recovers FEFs from the training traces, and then identifies common layer types or hyperparameters associated with these FEFs. For example, from the training traces, the attacker observed that a convolutional layer with ReLU may produce the FEF `f1-f2-f3`, the same layer with ELU may yield `f1-f2-f4`, and a fully connected layer may produce `f1-f5-f6-f7`. If the reconstructed victim FEF matches `f1-f2-f3`, the attacker can infer that it is a convolutional layer with ReLU. This allows the attacker to link a victim's FEF with corresponding layer types and hyperparameters. However, inaccuracies in the L2F classifier may introduce errors into the FEFs, making it difficult to find an exact match. If the precise FEF sequence is not found, the most similar one is determined using edit distance. The edit distance is defined as the minimum number of operations (i.e., replacement, insertion, and deletion) required to transform a victim FEF into an FEF listed in the map. Based on this distance, the attacker can identify the most similar FEF from the mapping information.

2) BBEF Mapping: Certain hyperparameters commonly influence the number of loop iterations or the paths taken by branches of the victim program. Consider the following code snippet from the Darknet library, which is executed in a fully connected layer:

```
void add_bias(float *output, float *biases, int
↪ batch, int n, int size)
{
    ...
    for(i = 0; i < n; ++i)
      for(j = 0; j < size; ++j)
        output[(b*n + i)*size + j] += biases[i];
}
```

In this code, the parameter `n` indicates the number of outputs, which allows the attacker to correlate the BBEF with this specific hyperparameter. Constructing the BBEF mapping requires some manual effort for the attacker; however, it can be reused once established. The attacker examines the target library to determine the relationships between loop iteration or branch path and relevant hyperparameters. Note that the attacker only needs to identify BBEFs for functions related to the hyperparameters. Finally, this process enables the attacker to reconstruct the network architecture of the victim model. For example, consider a recovered BBEF sequence `b1-b2-b3-b2-b3-b2-b3-b4`, where the number of occurrence of basic block `b3` indicates the number of outputs n. In this case, the attacker can infer that $n = 3$.

F. Attack Workflow

Algorithm 2 presents the workflow of our attack. The offline phase includes three main stages: preprocessing process (Line 1–3), function identification (Line 4–10), and basic block identification (Line 11–12). While most of the procedures are automated, four sub-algorithms marked with an asterisk require manual analysis. $Rule_{Depth}$ is an integer threshold (e.g., 3 or 4) that denotes the maximum call depth used to determine a single functional unit. This value is manually chosen by analyzing the source code to ensure that each labeled function encapsulates a self-contained semantic unit (e.g., an activation function or matrix multiplication).

$Rule_{merge}$ is a list of label groupings to be merged based on analysis of the confusion matrix C. The confusion matrix quantifies misclassifications during training, and we merge two labels when more than 2% of the instructions from one label are misclassified as another. In cases where misclassification occurs only under specific branch conditions, especially when those conditions reflect hyperparameter values, splitting the label may be more effective than merging. Technically, label merging suffices for function identification. In our evaluation, all label updates involved merging, except for one instance: the `activate` function in the previous example, which required branching-based labeling to distinguish activation types. If not split, such branching behavior can still be identified later

Algorithm 2: Algorithm illustrating the attack workflow. C represents a confusion matrix. Procedures marked with * involve manual analysis.

Input: Latency trace dataset for training D, victim latency trace V, and DNN library (binary) lib.
Output: Identified model structure S and identified hyperparameter H

// **Off-line phase**
$D_A \leftarrow Average(D)$
$Rule_{Depth} \leftarrow AnalyzeDepth^*(lib)$
$D_L \leftarrow Label(D_A, lib, Rule_{Depth})$
repeat
  $M_{L2F}, C, \alpha \leftarrow TrainModel_{L2F}(D_L)$
  $Rule_{Merge} \leftarrow AnalyzeMatrix^*(C)$
  $D_L \leftarrow UpdateLabel(D_L, lib, Rule_{Merge})$
until *accuracy* $\alpha \geq 0.99$;
$Rule_{Bound} \leftarrow AnalyzeBoundary^*(lib)$
$Map_{FEF} \leftarrow FindFEFMapping(D_L, Rule_{Bound})$
$F_{BB}, Map_{BB} \leftarrow AnalyzeHP^*(lib)$
$M_{L2BB} \leftarrow TrainModel_{L2BB}(D_L, F_{BB})$
// **On-line phase**
$V_A \leftarrow Average(V)$
$V_L \leftarrow InferenceModel(V_A)$
$S, h_1 \leftarrow Mapping_{FEF}(V_L, Map_{FEF}, Rule_{Bound})$
$h_2 \leftarrow Mapping_{BBEF}(V_L, M_{L2BB}, F_{BB}, Map_{BB})$
return Identified architecture $S, H = h_1 + h_2$

through basic block analysis. It is important to note that the final labeling configuration is not unique, as different groupings can still yield similarly accurate identification results. $Rule_{Bound}$ specifies how to extract individual layer boundaries from a full execution trace. For instance, in our analysis, each layer begins with over ten instructions belonging to the `forward_network` function.

F_{BB} denotes a set of functions that require basic block-level hyperparameter analysis, and Map_{BB} represents the mapping between BBEFs and the corresponding hyperparameters. Map_{BB} takes a BBEF sequence as an input and returns the inferred hyperparameter (e.g., the number of occurrences of a specific basic block). Once the relevant functions are selected, the L2BB model is trained automatically. It automatically identifies HSMM-related parameters such as a list of states. Among all manual steps, deriving Map_{BB} is the most labor-intensive, as it demands domain-specific knowledge of the target DNN library's implementation. Nevertheless, since most libraries rely on well-established algorithms (e.g., GEMM), relevant documentation are typically available.

Finally, when a target library version is changed, any part of the pipeline that involves manual analysis must be re-executed. However, since libraries often share common algorithms, and even updated versions often retain similar source code, the re-analysis typically requires less effort than the original process.

VI. Evaluation

A. Experimental Setup

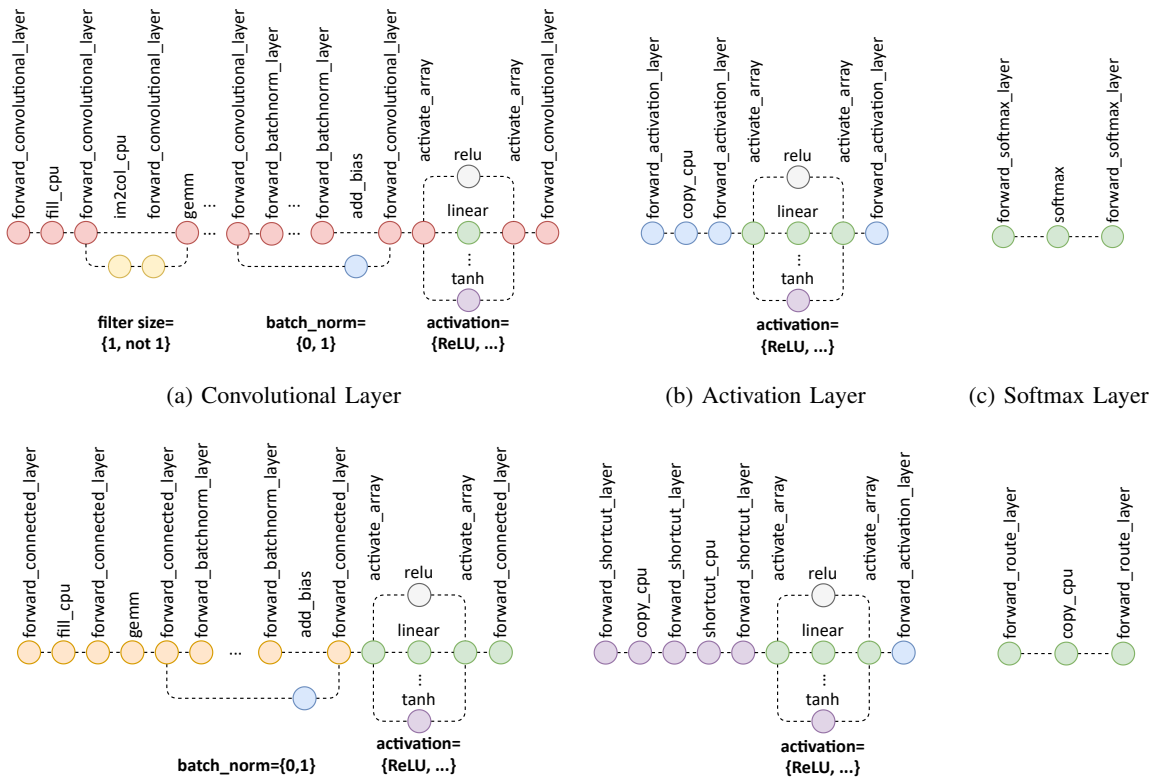
We instantiate our attacks on SGX-protected DNN models implemented using Darknet, TensorFlow Lite, and ONNX Runtime. Darknet is a widely-used deep learning library, which has been frequently adopted in recent studies on SGX-based DNN protection [19], [20], [21], [66], [25], [26], [23]. We first present the evaluation results for Darknet and discuss those for TensorFlow Lite and ONNX Runtime in §VI-E. We run these models on a Linux desktop equipped with Intel Core i7-10700 CPU 2.90GHz with SGX support and 48 GB memory. As recommended in SGX-Step, we configure the system parameters to mitigate the measurement noise by isolating a CPU core and disabling dynamic frequency scaling.

Training Model. To build the L2F classifier and the L2BB model, we randomly generate model architectures for training. This randomness ensures both comprehensiveness and generalization. Given the infinite number of potential network structure and hyperparameters, the generated models should be designed to inclusively reflect the diversity of the potential models. Also, the random generation helps prevent overfitting to specific models, thereby reducing dependency on specific latency patterns.

We take into account ten commonly used types of layers: a convolutional layer, a fully connected layer, an activation layer, a maxpool layer, an avgpool layer, a softmax layer, a route layer, a shortcut layer, a dropout layer, and a cost layer. To avoid runtime errors that may arise from entirely random configuration, we select a layer structure from one of 24 predefined layer combinations for each generation. For example, we enforce rules such as starting with a convolutional layer and ending with a softmax or cost layer. Additionally, we choose 12 hyperparameters frequently used in practice based on both established library guidelines and our experiences. These include the number of filters, filter size, activation function type, stride, padding, and batch normalize for the convolutional layer, output size, activation function type, and batch normalize for the fully connected layer, pool size and stride for the maxpool layer, and activation function type for the activation layer. The hyperparameters are also randomly selected from predefined sets to maintain consistency and control. As a result, we generate 300 random models, collecting and averaging ten traces for each model.

Victim Model. We validate our attacks on two types of victim models. First, we generate 100 random models using the same rules as for the training model generation. Additionally, we utilize two well-known models to demonstrate attack effectiveness in §VI-D: VGG-7 [67], which includes four convolutional layers and three fully connected layers, and LeNet [68], which includes two convolutional layers and three fully connected layers. For each victim model, we average fifty traces. For simplicity, we assume that the input image dimensions are known and that the height and width are equal.

In the following subsections, we justify our assumptions that FEFs and BBEFs can be used to identify hyperparameters by



(a) Convolutional Layer (b) Activation Layer (c) Softmax Layer

(d) Fully Connected Layer (e) Shortcut Layer (f) Route Layer

Figure 3: FEF sequences for the fully connected, shortcut, and route layers (nodes) used in Darknet.

presenting the execution flow mapping information. Next, we demonstrate the accuracy of reconstructing the victim models. To further illustrate the effectiveness of model architecture extraction, we show how the reconstructed architecture enhances the success rate of evasion attacks. Finally, we evaluate the attack accuracy on additional platforms, TensorFlow Lite and ONNX Runtime.

B. Darknet Library Analysis

First, we analyze how FEFs and BBEFs are mapped to layer types and hyperparameters. In our evaluation, we merge five functions for the pattern-based merges. Additionally, although activation functions are implemented as a single function, we differentiate instruction sequences based on the specific type of activation function. Thus, during the forwarding pass, we classify instruction latencies into 33 distinct classes (function groups). Under this setup, Figure 3 illustrates FEF sequences for each of the layers. Each of the maxpool, avgpool, dropout, and cost layers consists of one function (i.e., `forward_maxpool_layer`, `forward_avgpool_layer`, `forward_dropout_layer`, and `forward_cost_layer`, respectively.) We can observe that FEF sequences vary distinctly depending on layer type. Also, hyperparameters such as filter size, batch normalization, activation function type influence the FEF depending on their specific values.

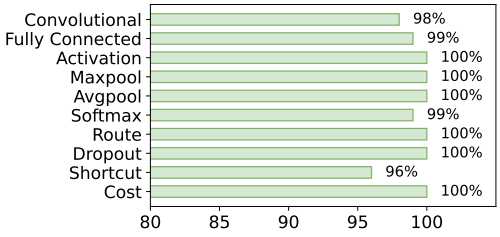
Next, we analyze how BBEFs correlate with hyperparameters. As shown in Table II, BBEFs can effectively reveal eight hyperparameters (i.e., $\text{fsize}_{\text{Conv}}$, $\#\text{filter}_{\text{Conv}}$, $\text{output}_{\text{Conv}}$, $\text{stride}_{\text{Conv}}$, $\text{padding}_{\text{Conv}}$, $\text{output}_{\text{FC}}$, $\text{size}_{\text{Maxpool}}$ and $\text{stride}_{\text{Maxpool}}$). For instance, some hyperparameters, such as the dimension of the output ($\text{output}_{\text{FC}}$), can be inferred from the BBEFs of several different functions. This redundancy allows the attacker to recover hyperparameters accurately even if noise affects one function's BBEF. These findings support the conclusion that Darknet model architectures can be reconstructed using DLS.

C. Accuracy of Extracted Model Architecture

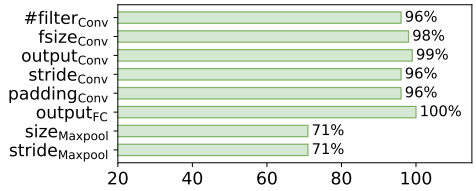
We evaluate the accuracy of the network structures and hyperparameters reconstructed by DLS. In all test cases, DLS accurately recovers the number of layers. The accuracy is measured as the ratio of correctly identified model architectures to the total number of them. Figure 4(a) presents the accuracy of reconstructed layer types and hyperparameters recoverable through the FEF mapping relationships across the 100 random models. Similarly, Figure 4(b) shows the accuracy of reconstructed hyperparameters that can be recovered through the BBEF mapping relationships. We can find that most layers were accurately identified. Also, we find that although the dropout layer and cost layer invoke different functions, they share identical instructions as they do not process input during the forward pass. Nonetheless, they are

Table II: Mapping relationships between each BBEF and its corresponding hyperparameter. $\phi(X)$ represents the number of iterations extracted from a function X's BBEF. A notation A|B means A or B. A notation A&B means A and B.

$\text{HP}_{\text{Layer Type}}$	Description	Basic Block-level Execution Flow (BBEF) Mapping
$\#\text{filter}_{\text{Conv}}$	The number of filters in the convolution	$\phi(\text{add\_bias}_{\text{batch\_norm}=0} \mid \text{gemm\_cpu} \mid \text{gemm\_nn})$
$\text{fsize}_{\text{Conv}}$	The size of the convolution filter	$\phi(\text{gemm\_nn} \mid \text{im2col\_cpu})$ & $\text{Input}_{\text{channel}}$
$\text{output}_{\text{Conv}}$	The width and height size of the output	$\phi(\text{add\_bias} \mid \text{fill\_cpu} \mid \text{normalize\_cpu} \mid \text{scale\_bias} \mid \text{gemm\_cpu} \mid \text{gemm\_nn} \mid \text{im2col\_cpu})$
$\text{stride}_{\text{Conv}}$	The stride size of the convolution filter	$(\text{Input}_{\text{height}} + 2 \times \text{padding}_{\text{Conv}} - \text{fsize}_{\text{Conv}})/(\text{output}_{\text{Conv}} - 1)$
$\text{padding}_{\text{Conv}}$	The size of padding of the input	$(\text{stride}_{\text{Conv}} \times \text{output}_{\text{Conv}} - 1 + \text{fsize}_{\text{Conv}} - \text{Input}_{\text{height}})/2$
$\text{output}_{\text{FC}}$	The output size of the fully connected layer	$\phi(\text{fill\_cpu} \mid \text{gemm\_nt} \mid \text{gemm\_cpu} \mid \text{normalize\_cpu} \mid \text{scale\_bias} \mid \text{add\_bias})$
$\text{size}_{\text{Maxpool}}$	The size of the pooling window	$\phi(\text{forward\_maxpool\_layer})$
$\text{stride}_{\text{Maxpool}}$	The stride size of the sliding window	$\phi(\text{forward\_maxpool\_layer})$

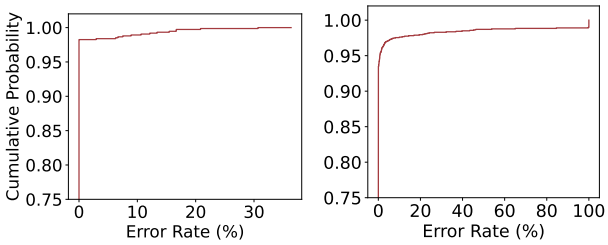


(a) Layer Type Accuracy (%)



(b) Hyperparameter Accuracy (%)

Figure 4: Accuracy across layer types and hyperparameters.



(a) FEF (b) BBEF

Figure 5: CDFs of the error rates for two execution flows.

distinguished by examining the preceding or following layers. Regarding hyperparameters, unlike those associated with the maxpool layer, the other hyperparameters can be identified through multiple functions, potentially improving the accuracy.

For the LeNet model, all layer types are correctly identified. Although three layers have errors in their FEFs, in one instance, the ground truth is matched by the candidate with the highest similarity, while for the other two, the top two candidates with the same highest similarity include the ground truth. In the case of hyperparameters, hyperparameters for one convolutional and one maxpool layers are not correctly reconstructed through BBEF mapping. In the case of VGG-7, four FEFs do not match the FEF mapping information. Three of them are corrected using the edit distance approach. However, for the remaining FEF, while the layer type is correctly identified, an error occurs in determining the activation function type. All hyperparameters are correctly reconstructed through BBEF mapping. Note that these two architectures do not conform to any of the predefined 24-layer combinations, thereby showing the attack robustness.

1) Accuracy of Function Identification: To further analyze the accuracy, we measure the performance of the L2F classifier, as it plays an important role in classifying each latency to its corresponding function, which subsequently influences the accuracy of the basic block identification. We measure precision, recall, and F1 score for each of the 33 classes, all of which achieve 100% macro-average. On average, our victim trace dataset comprises approximately 4 million latency points. Across 100 traces, latencies in 87 traces are accurately classified. Among the remaining 13 traces, 12 traces have fewer than 10 misclassifications. The number of misclassifications within the other trace is around 570. For this case, we average the trace based on RIP and re-classify it using the L2F classifier, which reduces misclassifications. This analysis highlights that measurement noise is the primary cause of misclassification, and noise in the other traces is successfully mitigated.

Lastly, we evaluate the accuracy of the identified FEFs using the execution flow error rate (ER), defined as follows:

$$\text{ER} = \frac{\text{ED}\left(EF^{+}, EF\right)}{\max\left(|EF^{+}|, |EF|\right)}$$

where $\text{ED}(EF^{+}, EF)$ represents the edit distance between the ground truth execution flow EF^{+} and the identified execution flow EF, and $|EF|$ represents a length of the execution flow EF. Figure 5(a) shows the cumulative distribution function (CDF) of the error rate across all found FEFs. The results indicate a high accuracy in the identified FEFs, with 98% of FEFs correctly recovered.

2) Accuracy of Basic Block Identification: Likewise, we assess the accuracy of the identified BBEFs using the same error rate metric used for the FEFs. Figure 5(b) illustrates the CDF of the error rate across all identified BBEFs. Unlike the FEF results, some BBEFs exhibit a 100% error rate. This occurs when function identification errors prevent the correct determination of the function's start and end ranges. Those cases are classified as 100% error cases. Nonetheless, the results demonstrate high accuracy in the identified BBEFs, with 93% of BBEFs accurately recovered.

D. Evasion Attacks via DLS

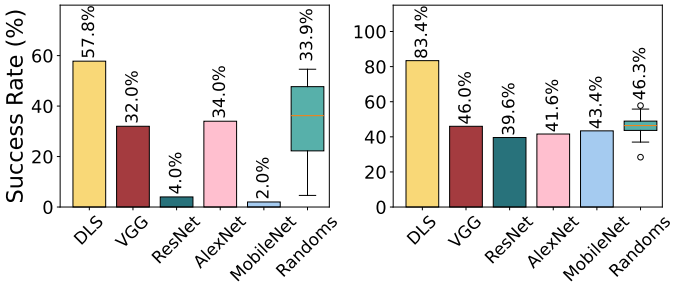
We show that the extracted model architectures can be further utilized to facilitate downstream attacks to compromise a target model's functionality. Specifically, evasion attacks [16], [17] undermine a victim model by exploiting adversarial examples. The adversarial images closely resemble the originals but include small perturbations, that lead to misclassification. These adversarial examples are transferable across different models. This transferability means that adversarial examples generated via one model can also lead to misclassification in other models. In addition, the likelihood of a successful attack increases when models share similar network structures and hyperparameters [17], [8]. Thus, we demonstrate the effectiveness of DLS by evaluating the success rate of evasion attacks on the LeNet model.

We employ the Fast Gradient Sign Method (FGSM) [69] to generate adversarial examples. To achieve this, we use TensorFlow to compute gradients and implement attacks, converting the Darknet victim model into a TensorFlow model. Fashion-MNIST [70] is used as the dataset for both training and testing. For evaluation, we randomly select 500 images from the test set and generate adversarial examples using the reconstructed model. For comparison, we also generate adversarial examples using other models. Specifically, we test four well-known architectures (VGG-16, ResNet50, AlexNet, and MobileNet) along with 50 randomly initialized models. We then apply these adversarial examples to the LeNet and VGG-7 models. As shown in Figure 6, the success rates of the adversarial examples are calculated by dividing the number of misclassified examples by the total number of test cases (i.e., 500). Consistent with previous findings [8], our attack demonstrates an increased success rate for evasion attacks compared to using black-box models.

E. DLS on Other Platforms

1) TensorFlow Lite: We instantiate our attack on another popular deep learning library, TensorFlow Lite [43]. TensorFlow Lite is a lightweight version of TensorFlow, optimized for memory efficiency and computational performance for DNN models. It is also employed in secureTF [18], an SGX-based model protection solution. For the evaluation, we use the open-source port for Intel SGX [71].

Since TensorFlow Lite supports only forward pass operations, we first train a DNN model using TensorFlow and then convert it to TensorFlow Lite without further optimization.



(a) LeNet (b) VGG-7

Figure 6: Effectiveness of evasion attacks.

Similar to our Darknet setup, we use 300 random models for training and 100 random models for testing. While Darknet utilizes eight activation functions, we use five activation functions in TensorFlow Lite that overlap as built-in functions. Additionally, we use the default stride mode for convolutional layers. In TensorFlow Lite, each model is represented as multiple nodes, with each layer comprising one or more nodes. These nodes are processed sequentially in a for-loop similar to Darknet. Therefore, to reconstruct the model, we identify node operations as follows: logistic, average pool, reshape, add, elu, mul, conv, softmax, fully connected, tanh, and max pool. For the sake of consistency, we use the term 'layer' to refer to node. In our evaluation, we apply a depth-based merge with a threshold of three and a pattern-based merge for four functions. Overall, the forward pass in TensorFlow Lite involves 105 functions, and we classify instruction latencies into 68 groups.

Similar to Darknet, each layer exhibits a unique FEF. Notably in TensorFlow Lite, certain hyperparameters (i.e., $\text{fsize}_{\text{Conv}}$, $\#\text{filter}_{\text{Conv}}$, $\text{output}_{\text{Conv}}$, and $\text{padding}_{\text{Conv}}$) can be identified by analyzing the frequency of function occurrences.

For example, the convolutional layer (i.e., CONV_2D) invokes the following code, simplified here for readability.

```
1  inline void Conv(...)
2  {
3    ...
4    for (int i = 0; i < filter_height; ++i)
5      ...
6      for (int j = 0; j < filter_width; ++j)
7        ...
8        for (int k = 0; k < input_depth; ++k) {
9          float iv = input_data[Offset(...)];
10         float fv = filter_data[Offset(...)];
11         total += (input_value * filter_value);
12       }
13     }
14   }
15   ...
16   output_data[Offset(...)] =
     ↪  ActivationFunctionWithMinMax(...);
17   ...
18 }
```

When the inner loop in Line 8 is repeated, the number of instructions between two `Offset` functions (from Line 10 to Line 9) is relatively small. In contrast, when the outer loop (Line 4) is evaluated, a larger number of instructions exists between the same `Offset` functions (from Line 10 to Line 9).

In addition, there is always a fixed number of instructions between two `Offset` functions when moving from Line 9 to Line 10. By analyzing the instruction counts for the `Offset` and `Conv` functions, we can deduce the relationship between the FEF and filter size (i.e., $\text{fsize}_{\text{Conv}}$). Similarly, we can infer values for $\#\text{filter}_{\text{Conv}}$, $\text{output}_{\text{Conv}}$, and $\text{padding}_{\text{Conv}}$. For that, we record the instruction counts for each function in FEF and use these counts to inform the FEF mapping.

In our analysis of the 100 victim models, we successfully recovered the exact number of layers and layer types in every model. Additionally, $\#\text{filter}_{\text{Conv}}$, $\text{fsize}_{\text{Conv}}$, $\text{output}_{\text{Conv}}$, $\text{padding}_{\text{Conv}}$, and $\text{output}_{\text{FC}}$ achieve 89%, 100%, 88%, 88%, and 100% accuracy through the FEF mapping. The accuracy of the FEF mapping is attributed to the high precision of the L2F classifier: with a victim trace dataset containing approximately 600 million latency points and only around 1,200 misclassified instances, we achieved macro-average F1 score, precision, and recall values of 1.

For the BBEF mapping, we can recover four hyperparameters (i.e., , $\text{size}_{\text{Maxpool}}$, $\text{stride}_{\text{Maxpool}}$, $\text{size}_{\text{Avgpool}}$, $\text{stride}_{\text{Avgpool}}$). Specifically, 58% of $\text{size}_{\text{Maxpool}}$ and $\text{stride}_{\text{Maxpool}}$ values were accurately identified, while 88% of $\text{size}_{\text{Avgpool}}$ and $\text{stride}_{\text{Avgpool}}$ values were correctly inferred. The results demonstrate that our attacks are not limited to a specific library or implementation, although these may influence accuracy. As a result, DLS can be applied to other platforms as well.

2) ONNX Runtime: Lastly, we evaluated our attack on ONNX Runtime [44], using the same sets of training and testing models converted from the TensorFlow Lite models. We identify 13 node operation types, including AveragePool, Softmax, MatMul, Sigmoid, Relu, Tanh, Elu, MaxPool, Mul, Transpose, Add, Reshape, and Conv. A fully connected layer is implemented using a combination of MatMul and Add. From a total of 493 function labels, we grouped instruction latencies into 216 distinct categories. Across the 100 victim models, we successfully recovered the exact number and types of layers using the FEF mapping. For hyperparameter inference, the hyperparameters $\#\text{filter}_{\text{Conv}}$, $\text{fsize}_{\text{Conv}}$, $\text{output}_{\text{Conv}}$, $\text{stride}_{\text{Conv}}$ and $\text{padding}_{\text{Conv}}$ were inferred with accuracies of 83%, 92%, 88%, 88%, and 83%, respectively. The hyperparameter $\text{output}_{\text{FC}}$ was recovered with 98% accuracy. Additionally, $\text{size}_{\text{Maxpool}}$, $\text{stride}_{\text{Maxpool}}$, $\text{size}_{\text{Avgpool}}$ and $\text{stride}_{\text{Avgpool}}$ were recovered with 72%, 72%, 83% and 83% accuracy, respectively. We provide the mapping relationships for TensorFlow Lite and ONNX Runtime in our repository [45].

Summary. Figure 7 presents a histogram of the accuracy of the extracted model architectures for each of the 100 models. For each model, we measure an accuracy as the ratio of correctly identified model architecture components to the total number of them. The average accuracy for Darknet is 97.3%, with a range of 75.0% to 100% (σ=5.2). For TensorFlow Lite, the average accuracy is 96.4%, with a range of 68.2% to 100% (σ=3.6). For ONNX Runtime, the average accuracy is 93.6% with a range of 65.5% to 100% (σ=9.4). DLS achieves high attack accuracy across the libraries, demonstrating its effectiveness in recovering model architectures.

VII. Related Work

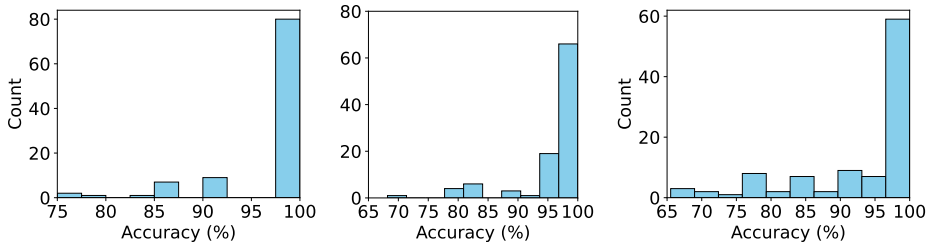
Privacy Attacks on Deep Learning Models. Side-channel attacks have proven effective in extracting DNN model information across various target platforms, each leveraging different hardware features. Beyond CPUs and TEEs, as discussed in §II, many studies have demonstrated attacks on FPGA [72], [73], [74], [75], Application-Specific Integrated Circuit (ASIC) [76], and GPU [8], [9], [77], [78]. These attacks can exploit electromagnetic emanations [72], [8], [75], memory access patterns [73], bus snooping [8], GPU PCIe traffic [9], DRAM [11], [76], remote power analysis [74], *etc*.

In addition to hardware-based side channels, adversaries can exploit DNN binaries [79], [80] or query responses [81], [82], [83]. Binary-based approaches decompile DNN executables, where victim models are compiled and embedded, to conduct symbolic analysis [79] or static analysis [80]. These attacks can be mitigated in the SGX setup, where the executable can remain confidential [84]. On the other hand, query-based attacks send numerous queries to the victim model using different inputs. Then, the attacker infers model behavior from the corresponding results. The input-output pairs are used to train a model to achieve comparable fidelity to the victim model. Those approaches leverage algorithmic features such as predicted class labels [81], gradients of objective function [82], and labeling oracle [83]. Because these attacks assume prior knowledge of the model architecture, DLS can be employed to obtain this information beforehand.

Side-channel Attacks on Intel SGX. Since Intel SGX assumes a privileged attacker, it enables an attacker to manipulate the system state (e.g., core isolation) or exploit resources unavailable to non-privileged attackers. Numerous studies have reported side-channel attacks targeting Intel SGX under this environment.

When adversarial and victim processes run on the same core, the attacker can exploit per-core resources such as Branch Prediction Units [38], [37], L1/L2 caches [85], and Translation Lookaside Buffer (TLB) [86]. While these per-core resources generally produce lower noise and are easy to deploy in practice, they may not always be accessible to the attacker. Still, the adversary can exploit cross-core resources such as last-level cache (LLC) [87] and DRAM row buffer [86]. Additionally, the untrusted OS can utilize page access patterns for controlled-channel attacks [88], [61]. It monitors a trace of page accesses to analyze deterministic patterns. Furthermore, after the discovery of microarchitectural vulnerabilities such as Spectre [89] and Meltdown [90], its variant has been successfully demonstrated on Intel SGX as well [91], [92], [93]. These attacks allow the attacker to breach security boundaries or gain control over the victim's control transfer.

These attacks often demonstrate their efficiency by targeting specific secret-dependent branches [38], [37], [39], and cryptographic algorithms such as AES [40], [41] and RSA [94]. Although only a few studies [29], [10] have focused on side channels to extract SGX-protected DNN models, DLS can



(a) Darknet (b) TensorFlow Lite (c) ONNX Runtime

Figure 7: Accuracy of the extracted model architectures.

incorporate these attacks to gather additional side information and extract other model details that are dependent of the victim's execution flow.

VIII. Limitations and Discussion

A. Countermeasures

Our attack extracts features from instruction latencies that are unique to each function. One type of countermeasure is algorithm hardening to obscure these unique patterns [95], [96], [97], [98]. The compiler rewrites a victim program to eliminate secret-dependent branches. However, our attack identifies patterns through a consecutive series of instruction latencies rather than relying on specific branch conditions. Therefore, it limits the effectiveness of such countermeasures.

In addition, several obfuscation techniques specifically targeting DNN models have been proposed [99], [100], [101], [102], [103]. However, these methods were not designed with single-stepping attacks or TEEs in mind, making their adaptation to our threat model infeasible. For example, NeurObfuscator [99] transforms a model into a functionally equivalent but architecturally different version. Since the transformed model often maintains high inference accuracy, it can become a new target for model extraction. ObfuNAS [102] intentionally degrades model performance under FLOPs (floating-point operations) constraints. Also, these approaches require the development of the specific full-stack ML inference framework [104]. Moreover, ModelObfuscator [101] focuses on software-level reverse engineering rather than side-channel attacks, NNReArch [103] assumes the prsence of specialized hardware, and DNNClock [100] targets memory side-channel attacks.

Since DLS leverages single-stepping attacks, it can potentially be mitigated at the outset. A single-stepping attack utilizes APIC interrupts, which cause the running enclave to frequently exit asynchronously (AEX) to handle IRQs. Although interrupts are only delivered to the OS kernel, recent studies [105], [106], [107], [108] propose software-based approaches to recognize AEX events in the enclave. Some approaches [107], [108], [106] depend on Intel Transactional Synchronization Extensions (TSX) features, which enable atomically executing a set of instructions. An interrupt in the middle of the transaction results in an abort, which triggers a fallback mechanism for its inspection. Hyperrace [105] secures a trusted time source inside the enclave, thereby it can periodically check the number of resume routines executed. However, these AEX-awareness approaches introduce significant overhead and carry the risk of false negatives or false positives. Additionally, Intel TSX is not widely adopted.

AEX-Notify [109] proposes a solution to address the root cause of single-stepping attacks by introducing a new instruction, EDECCSSA [110]. It allows developers to secure a trusted handler that preemptively prepares page accesses before the enclave resumes. By doing so, it disturbs deterministic single stepping and can thus mitigate our attack. In practice, AEX-Notify requires both hardware support (i.e., the EDECCSSA instruction) and software support (i.e., the trusted IRQ handler). Although Intel SGX SDK (from version 2.22) includes such a handler, most dominating enclave development frameworks (such as Fortanix EDP [111], Enarx [112], OpenEnclave [113], Gramine [114], Asylo [115], Certifier Framework [116], and Occlum [117]) do not yet support AEX-Notify. Moreover implementing the trusted handler demands expert knowledge of SGX internals, further limiting its adoption in practice.

B. Generality

While confidential DNN applications [118], [119] are actively supported on Intel SGX, such applications are also widely supported across other TEE platforms. As described in §V-F, DLS operates on latency trace as input, making it decoupled from platform-specific mechanisms. Consequently, it is not limited to SGX and can be extended to other TEEs. While the latency traces used in DLS are collected using techniques originally introduced for SGX, recent work [120], [121], [122], [123], [124] has demonstrated that this primitive is not tied to SGX, and can be effectively replicated across a range of TEE architectures, including AMD SEV [120], ARM TrustZone [121], [122], and Intel TDX [123], [124]. For instance, SEV-Step [120] leverages APIC-based interrupts to enable single stepping on AMD SEV, successfully reproducing the Nemesis attack and showing that instruction latency remains measurable and distinguishable. Similarly, the interrupt-based single stepping is used for fine-grained cache attacks on ARM TrustZone [122], [121] and RISC-V [125], and instruction counting attacks on Intel TDX [124]. These studies demonstrate that the single-stepping mechanism is applicable

across various TEE platforms. As long as latency traces can be reliably collected, DLS remains directly reusable. This cross-platform compatibility highlights the generality of our approach.

We show that our attacks are applicable to CNN-based neural networks on well-known libraries. The feasibility of our approach stems from the fact that victim DNN models exhibit distinct execution flows across the entire program, influenced by sensitive information (i.e., network structure and hyperparameters). Second, each function or layer, performing a specific role, is likely to produce a unique latency pattern. Further, certain hyperparameters inherently impact control flow elements like branch decisions and iteration counts. Since these characteristics are generally applicable, DLS can be extended to other deep learning libraries and deep learning model types such as recurrent neural networks (RNN), GAN, and autoencoder. Note that DLS measures the latency of machine instructions and is independent of language or software versions.

IX. Conclusion

DLS demonstrates that a privileged adversary can collect an instruction latency trace of a victim DNN model and identify its distinctive execution behaviors. The behaviors extend beyond simply identifying the sequence of executed functions (i.e., FEF); they also include the sequence of executed basic blocks, achieving instruction-level granularity (i.e., BBEF). By leveraging FEF and BBEF, our approach can successfully reconstruct the network structure and hyperparameters. We demonstrate the attack on the three DNN libraries, which are common in current DNN model protection research. Our results indicate that DLS can recover model architecture with high accuracy. Looking ahead, we recommend that developers of TEE-protected DNN models implement effective countermeasures, such as AEX-Notify, to mitigate these risks.

Ethical Considerations

This study adheres to ethical guidelines throughout the entire process. Our attack is conducted under a controlled experimental setup. We collect our dataset exclusively from this environment, ensuring that no attacks are launched against any external entities. Furthermore, we make our datasets and source code publicly available in our repository [45].

Importantly, our work does not introduce any new vulnerabilities in SGX. Instead, DLS builds upon existing single-stepping attacks [35], [36], which have been documented in prior literature [109], [110]. Our contribution lies in demonstrating the impact of these attacks on DNN libraries and proposing a methodology. DLS is not tailored to the three evaluated libraries or their specific implementations; rather, it demonstrates a general technique applicable to a broader class of enclave applications.

Acknowledgment

We thank the anonymous reviewers for their insightful feedback. This work was supported in part by the Texas A&M Engineering Experiment Station on behalf of its SecureAmerica Institute, National Science Foundation under grant 2229876, Department of Homeland Security, IBM, and Office of Naval Research under grant N00014-23-1-2157. Any opinions, findings, recommendations, and conclusions expressed in this material are those of the authors and do not necessarily reflect the views of the sponsors.

References

[1] M. I. Razzak, S. Naz, and A. Zaib, "Deep Learning for Medical Image Processing: Overview, Challenges and the Future," *Classification in BioApps: Automation of decision making*, pp. 323–350, 2018.

[2] T. Sun, B. Zhou, L. Lai, and J. Pei, "Sequence-based Prediction of Protein Protein Interaction using a Deep-Learning Algorithm," *BMC Bioinformatics*, vol. 18, pp. 1–8, 2017.

[3] L. Deng, G. Hinton, and B. Kingsbury, "New Types of Deep Neural Network Learning for Speech Recognition and Related Applications: An Overview," in *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*. IEEE, 2013, pp. 8599–8603.

[4] 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," 2020. [Online]. Available: https://arxiv.org/abs/1810.03487

[5] M. Yan, C. W. Fletcher, and J. Torrellas, "Cache Telepathy: Leveraging Shared Resource Attacks to Learn DNN Architectures," in *29th USENIX Security Symposium (USENIX Security 20)*, Aug. 2020, pp. 2003–2020.

[6] Z. Liu, Y. Yuan, Y. Chen, S. Hu, T. Li, and S. Wang, "DeepCache: Revisiting Cache Side-Channel Attacks in Deep Neural Networks Executables," in *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2024.

[7] Y. Liu and A. Srivastava, "GANRED: GAN-based Reverse Engineering of DNNs via Cache Side-Channel," in *Proceedings of the 2020 ACM SIGSAC Conference on Cloud Computing Security Workshop*, 2020, pp. 41–52.

[8] 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 *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020, pp. 385–399.

[9] Y. Zhu, Y. Cheng, H. Zhou, and Y. Lu, "Hermes Attack: Steal DNN Models with Lossless Inference Accuracy," in *30th USENIX Security Symposium (USENIX Security 21)*, Aug. 2021, pp. 1973–1988.

[10] X. Zhang, A. A. Ding, and Y. Fei, "Deep-Learning Model Extraction Through Software-Based Power Side-Channel," in *2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD)*. IEEE, 2023, pp. 1–9.

[11] A. S. Rakin, M. H. I. Chowdhuryy, F. Yao, and D. Fan, "Deepsteal: Advanced model extractions leveraging efficient weight stealing in memories," in *2022 IEEE symposium on security and privacy (SP)*. IEEE, 2022, pp. 1157–1174.

[12] R. Joud, P.-A. Moëllic, S. Pontié, and J.-B. Rigaud, "A Practical Introduction to Side-Channel Extraction of Deep Neural Network Parameters," in *International Conference on Smart Card Research and Advanced Applications*. Springer, 2022, pp. 45–65.

[13] R. Shokri, M. Stronati, C. Song, and V. Shmatikov, "Membership Inference Attacks against Machine Learning Models," in *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2017, pp. 3–18.

[14] Y. Long, L. Wang, D. Bu, V. Bindschaedler, X. Wang, H. Tang, C. A. Gunter, and K. Chen, "A Pragmatic Approach to Membership Inferences on Machine Learning Models," in *2020 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 2020, pp. 521–534.

[15] H. Hu, Z. Salcic, L. Sun, G. Dobbie, P. S. Yu, and X. Zhang, "Membership Inference Attacks on Machine Learning: A Survey," *ACM Computing Surveys (CSUR)*, vol. 54, no. 11s, pp. 1–37, 2022.

[16] Y. Liu, X. Chen, C. Liu, and D. Song, "Delving into Transferable Adversarial Examples and Black-box Attacks," *arXiv preprint arXiv:1611.02770*, 2016.

[17] N. Papernot, P. McDaniel, and I. Goodfellow, "Transferability in Machine Learning: From Phenomena to Black-box Attacks using Adversarial Samples," *arXiv preprint arXiv:1605.07277*, 2016.

[18] D. L. Quoc, F. Gregor, S. Arnautov, R. Kunkel, P. Bhatotia, and C. Fetzer, "secureTF: A Secure TensorFlow Framework," in *Proceedings of the 21st International Middleware Conference*, ser. Middleware '20. New York, NY, USA: Association for Computing Machinery, 2020.

[19] K. Kim, C. H. Kim, J. J. Rhee, X. Yu, H. Chen, D. J. Tian, and B. Lee, "Vessels: Efficient and Scalable Deep Learning Prediction on Trusted Processors," in *Proceedings of the 11th ACM Symposium on Cloud Computing (SOCC)*, 2020.

[20] P. Yuhala, P. Felber, V. Schiavoni, and A. Tchana, "PLINIUS: Secure and Persistent Machine Learning Model Training," in *Proceedings of the 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2021.

[21] Y. Li, D. Zeng, L. Gu, Q. Chen, S. Guo, A. Zomaya, and M. Guo, "Lasagna: Accelerating Secure Deep Learning Inference in SGX-enabled Edge Cloud," in *Proceedings of the 12th ACM Symposium on Cloud Computing (SOCC)*, 2021.

[22] F. Tramer and D. Boneh, "Slalom: Fast, Verifiable and Private Execution of Neural Networks in Trusted Hardware," *arXiv preprint arXiv:1806.03287*, 2018.

[23] Z. Gu, H. Huang, J. Zhang, D. Su, H. Jamjoom, A. Lamba, D. Pendarakis, and I. Molloy, "Confidential Inference via Ternary Model Partitioning," *arXiv preprint arXiv:1807.00969*, 2018.

[24] J. Choi, J. Kim, C. Lim, S. Lee, J. Lee, D. Song, and Y. Kim, "GuardiaNN: Fast and Secure On-Device Inference in TrustZone Using Embedded SRAM and Cryptographic Hardware," in *Proceedings of the 23rd ACM/IFIP International Middleware Conference*, 2022, pp. 15–28.

[25] F. Mo, A. S. Shamsabadi, K. Katevas, S. Demetriou, I. Leontiadis, A. Cavallaro, and H. Haddadi, "DarkneTZ: Towards Model Privacy at the Edge using Trusted Execution Environments," in *Proceedings of the 18th International Conference on Mobile Systems, Applications, and Services*, 2020, pp. 161–174.

[26] A. Gangal, M. Ye, and S. Wei, "HybridTEE: Secure Mobile DNN Execution Using Hybrid Trusted Execution Environment," in *2020 Asian Hardware Oriented Security and Trust Symposium (AsianHOST)*. IEEE, 2020, pp. 1–6.

[27] M. Misono, D. Stavrakakis, N. Santos, and P. Bhatotia, "Confidential VMs Explained: An Empirical Analysis of AMD SEV-SNP and Intel TDX," *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, vol. 8, no. 3, pp. 1–42, 2024.

[28] Y. Gao, H. Qiu, Z. Zhang, B. Wang, H. Ma, A. Abuadbba, M. Xue, A. Fu, and S. Nepal, "DeepTheft: Stealing DNN Model Architectures through Power Side Channel," in *2024 IEEE Symposium on Security and Privacy (SP)*, Los Alamitos, CA, USA, 2024, pp. 3311–3326.

[29] Y. Yuan, Z. Liu, S. Deng, Y. Chen, S. Wang, Y. Zhang, and Z. Su, "HyperTheft: Thieving Model Weights from TEE-Shielded Neural Networks via Ciphertext Side Channels," in *Proceedings of the 2024 ACM SIGSAC Conference on Computer and Communications Security*, 2024.

[30] E. Tromer, D. A. Osvik, and A. Shamir, "Efficient Cache Attacks on AES, and Countermeasures," *Journal of Cryptology*, vol. 23, pp. 37–71, 2010.

[31] Y. Yarom and K. Falkner, "FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack," in *23rd USENIX security symposium (USENIX security 14)*, 2014, pp. 719–732.

[32] M. Lipp, A. Kogler, D. Oswald, M. Schwarz, C. Easdon, C. Canella, and D. Gruss, "PLATYPUS: Software-based Power Side-Channel Attacks on x86," in *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2021, pp. 355–371.

[33] M. Li, L. Wilke, J. Wichelmann, T. Eisenbarth, R. Teodorescu, and Y. Zhang, "A Systematic Look at Ciphertext Side Channels on AMD SEV-SNP," in *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2022, pp. 337–351.

[34] Intel, "What Is the Difference in Cache Memory Between CPUs for Intel Xeon Scalable Processors," https://www.intel.com/content/www/us/en/support/articles/000027820/processors/intel-xeon-processors.html, [Online; accessed April 21, 2025].

[35] J. Van Bulck, F. Piessens, and R. Strackx, "SGX-Step: A Practical Attack Framework for Precise Enclave Execution Control," in *Proceedings of the 2nd Workshop on System Software for Trusted Execution*, 2017, pp. 1–6.

[36] ——, "Nemesis: Studying Microarchitectural Timing Leaks in Rudimentary CPU Interrupt Logic," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 178–195.

[37] H. Yavarzadeh, A. Agarwal, M. Christman, C. Garman, D. Genkin, A. Kwong, D. Moghimi, D. Stefan, K. Taram, and D. Tullsen, "Pathfinder: High-Resolution Control-Flow Attacks Exploiting the Conditional Branch Predictor," in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, 2024, pp. 770–784.

[38] S. Lee, M.-W. Shih, P. Gera, T. Kim, H. Kim, and M. Peinado, "Inferring Fine-grained Control Flow Inside SGX Enclaves with Branch Shadowing," in *26th USENIX Security Symposium (USENIX Security 17)*. Vancouver, BC: USENIX Association, 2017.

[39] I. Puddu, M. Schneider, M. Haller, and S. Čapkun, "Frontal Attack: Leaking Control-Flow in SGX via the CPU Frontend," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021, pp. 663–680.

[40] A. Moghimi, G. Irazoqui, and T. Eisenbarth, "CacheZoom: How SGX Amplifies the Power of Cache Attacks," in *Proceedings of the 19th International Conference on Cryptographic Hardware and Embedded Systems (CHES)*. Springer, 2017.

[41] A. Moghimi, J. Wichelmann, T. Eisenbarth, and B. Sunar, "MemJam: A False Dependency Attack Against Constant-Time Crypto Implementations," *International Journal of Parallel Programming*, 2018.

[42] J. Redmon, "Darknet: Open Source Neural Networks in C," http://pjreddie.com/darknet/, [Online; accessed April 21, 2025].

[43] Google, "TensorFlow Lite," https://www.tensorflow.org/lite, [Online; accessed April 21, 2025].

[44] Microsoft, "ONNX Runtime," https://github.com/microsoft/onnxruntime-openenclave.git, [Online; accessed July 20, 2025].

[45] "Public GitLab Repository for DLS," https://gitlab.com/s3lab-code/public/dls, 2025.

[46] F. Tramèr, F. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart, "Stealing machine learning models via prediction apis," in *25th USENIX Security Symposium (USENIX Security 16)*, 2016, pp. 601–618.

[47] L. Wei, B. Luo, Y. Li, Y. Liu, and Q. Xu, "I Know What You See: Power Side-Channel Attack on Convolutional Neural Network Accelerators," in *Proceedings of the 34th Annual Computer Security Applications Conference (ACSAC)*, 2018, pp. 393–406.

[48] K. Goto and R. A. V. D. Geijn, "Anatomy of High-Performance Matrix Multiplication," *ACM Transactions on Mathematical Software (TOMS)*, vol. 34, no. 3, pp. 1–25, 2008.

[49] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, H. Shen, M. Cowan, L. Wang, Y. Hu, L. Ceze *et al.*, "TVM: An Automated End-to-End Optimizing Compiler for Deep Learning," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, 2018, pp. 578–594.

[50] N. Rotem, J. Fix, S. Abdulrasool, G. Catron, S. Deng, R. Dzhabarov, N. Gibson, J. Hegeman, M. Lele, R. Levenstein *et al.*, "Glow: Graph Lowering Compiler Techniques for Neural Networks," *arXiv preprint arXiv:1805.00907*, 2018.

[51] M. Li, Y. Zhang, H. Wang, K. Li, and Y. Cheng, "Cipherleaks: Breaking constant-time cryptography on amd sev via the ciphertext side channel," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021, pp. 717–732.

[52] S. Gueron, "A Memory Encryption Engine Suitable for General Purpose Processors," Cryptology ePrint Archive, Paper 2016/204, 2016. [Online]. Available: https://eprint.iacr.org/2016/204

[53] V. Costan, "Intel SGX Explained," *IACR Cryptol, EPrint Arch*, 2016.

[54] M. El-Hindi, T. Ziegler, M. Heinrich, A. Lutsch, Z. Zhao, and C. Binnig, "Benchmarking the Second Generation of Intel SGX Hardware," in *Proceedings of the 18th International Workshop on Data Management on New Hardware*, 2022, pp. 1–8.

[55] P.-C. Cheng, W. Ozga, E. Valdez, S. Ahmed, Z. Gu, H. Jamjoom, H. Franke, and J. Bottomley, "Intel TDX Demystified: A Top-Down Approach," *ACM Computing Surveys*, vol. 56, no. 9, pp. 1–33, 2024.

[56] D. Lee, D. Jung, I. T. Fang, C.-C. Tsai, and R. A. Popa, "An off-chip attack on hardware enclaves via the memory bus," in *29th USENIX Security Symposium (USENIX Security 20)*, 2020.

[57] P. Simmons, "Security through amnesia: a software-based solution to the cold boot attack on disk encryption," in *Proceedings of the 27th Annual Computer Security Applications Conference*, 2011, pp. 73–82.

[58] S. U. Hassan, I. Gridin, I. M. Delgado-Lozano, C. P. García, J.-J. Chi-Domínguez, A. C. Aldaya, and B. B. Brumley, "Déjà Vu: Side-Channel Analysis of Mozilla's NSS," in *Proceedings of the 2020 ACM*

SIGSAC Conference on Computer and Communications Security, 2020, pp. 1887–1902.

[59] D. Moghimi, "Downfall: Exploiting Speculative Data Gathering," in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023, pp. 7179–7193.

[60] C. Easdon, M. Schwarz, M. Schwarzl, and D. Gruss, "Rapid Prototyping for Microarchitectural Attacks," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 3861–3877.

[61] D. Moghimi, J. Van Bulck, N. Heninger, F. Piessens, and B. Sunar, "CopyCat: Controlled Instruction-Level Attacks on Enclaves," in *29th USENIX Security Symposium (USENIX Security 20)*, 2020, pp. 469–486.

[62] M. Hähnel, W. Cui, and M. Peinado, "High-Resolution Side Channels for Untrusted Operating Systems," in *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, 2017, pp. 299–312.

[63] A. Moghimi, G. Irazoqui, and T. Eisenbarth, "CacheZoom: How SGX Amplifies The Power of Cache Attacks," in *Proceedings of the 19th International Conference on Cryptographic Hardware and Embedded Systems (CHES)*. Springer, 2017, pp. 69–90.

[64] T. Sato and Y. Kameya, "New Advances in Logic-based Probabilistic Modeling by PRISM," in *Probabilistic Inductive Logic Programming: Theory and Applications*. Springer, 2008, pp. 118–155.

[65] G. D. Forney, "The Viterbi Algorithm," *Proceedings of the IEEE*, vol. 61, no. 3, pp. 268–278, 1973.

[66] S. P. Bayerl, T. Frassetto, P. Jauernig, K. Riedhammer, A.-R. Sadeghi, T. Schneider, E. Stapf, and C. Weinert, "Offline Model Guard: Secure and Private ML on Mobile Devices," in *Proceedings of the 23rd Conference on Design, Automation and Test in Europe*, ser. DATE '20, San Jose, CA, USA, 2020.

[67] F. Li, B. Liu, X. Wang, B. Zhang, and J. Yan, "Ternary Weight Networks," *arXiv preprint arXiv:1605.04711*, 2016.

[68] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-Based Learning Applied to Document Recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[69] I. J. Goodfellow, J. Shlens, and C. Szegedy, "Explaining and Harnessing Adversarial Examples," 2015. [Online]. Available: https://arxiv.org/abs/1412.6572

[70] H. Xiao, K. Rasul, and R. Vollgraf, "Fashion-MNIST: a Novel Image Dataset for Benchmarking Machine Learning Algorithms," *arXiv preprint arXiv:1708.07747*, 2017.

[71] tensorflow-lite sgx, "tensorflow-lite-sgx," https://github.com/Jumpst3r/tensorflow-lite-sgx, [Online; accessed April 21, 2025].

[72] L. Batina, S. Bhasin, D. Jap, and S. Picek, "CSI NN: Reverse Engineering of Neural Network Architectures Through Electromagnetic Side Channel," in *28th USENIX Security Symposium (USENIX Security 19)*, Santa Clara, CA, Aug. 2019, pp. 515–532.

[73] W. Hua, Z. Zhang, and G. E. Suh, "Reverse Engineering Convolutional Neural Networks Through Side-channel Information Leaks," in *Proceedings of the 55th Annual Design Automation Conference*, 2018, pp. 1–6.

[74] Y. Zhang, R. Yasaei, H. Chen, Z. Li, and M. A. Al Faruque, "Stealing Neural Network Structure Through Remote FPGA Side-Channel Analysis," *IEEE Transactions on Information Forensics and Security*, vol. 16, pp. 4377–4388, 2021.

[75] H. Yu, H. Ma, K. Yang, Y. Zhao, and Y. Jin, "DeepEM: Deep Neural Networks Model Recovery through EM Side-Channel Information Leakage," in *2020 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*. IEEE, 2020, pp. 209–218.

[76] D. Yang, P. J. Nair, and M. Lis, "HuffDuff: Stealing Pruned DNNs from Sparse Accelerators," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2023, pp. 385–399.

[77] J. Wei, Y. Zhang, Z. Zhou, Z. Li, and M. A. Al Faruque, "Leaky DNN: Stealing Deep-learning Model Secret with GPU Context-switching Side-channel," in *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2020, pp. 125–137.

[78] H. Naghibijouybari, A. Neupane, Z. Qian, and N. Abu-Ghazaleh, "Rendered Insecure: GPU Side Channel Attacks are Practical," in *Proceedings of the 2018 ACM SIGSAC conference on computer and communications security*, 2018, pp. 2139–2153.

[79] Z. Liu, Y. Yuan, S. Wang, X. Xie, and L. Ma, "Decompiling x86 Deep Neural Network Executables," in *32nd USENIX Security Symposium (USENIX Security 23)*. Anaheim, CA: USENIX Association, 2023, pp. 7357–7374.

[80] J. Zhang, P. Wang, and D. Wu, "LibSteal: Model Extraction Attack Towards Deep Learning Compilers by Reversing DNN Binary Library," in *Proceedings of the 18th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE)*, 2023.

[81] F. Tramèr, F. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart, "Stealing Machine Learning Models via Prediction APIs," in *Proceedings of the 25th USENIX Conference on Security Symposium (USENIX Security 16)*, 2016.

[82] B. Wang and N. Z. Gong, "Stealing Hyperparameters in Machine Learning," in *2018 IEEE Symposium on Security and Privacy (SP)*, 2018, pp. 36–52.

[83] M. Jagielski, N. Carlini, D. Berthelot, A. Kurakin, and N. Papernot, "High Accuracy and High Fidelity Extraction of Neural Networks," in *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, 2020, pp. 1345–1362.

[84] J. Seo, B. Lee, S. M. Kim, M.-W. Shih, I. Shin, D. Han, and T. Kim, "SGX-Shield: Enabling Address Space Layout Randomization for SGX Programs," in *Proceedings of the 24th Network and Distributed System Security Symposium (NDSS)*, 2017.

[85] O. Oleksenko, B. Trach, R. Krahn, A. Martin, C. Fetzer, and M. Silberstein, "Varys: Protecting SGX Enclaves from Practical Side-Channel Attacks," in *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference*, ser. USENIX ATC '18. USENIX Association, 2018.

[86] W. Wang, G. Chen, X. Pan, Y. Zhang, X. Wang, V. Bindschaedler, H. Tang, and C. A. Gunter, "Leaky Cauldron on the Dark Land: Understanding Memory Side-Channel Hazards in SGX," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '17, New York, NY, USA, 2017.

[87] F. Brasser, U. Müller, A. Dmitrienko, K. Kostiainen, S. Capkun, and A.-R. Sadeghi, "Software Grand Exposure: SGX Cache Attacks Are Practical," in *Proceedings of the 11th USENIX Conference on Offensive Technologies*, 2017.

[88] Y. Xu, W. Cui, and M. Peinado, "Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems," in *2015 IEEE Symposium on Security and Privacy*. IEEE, 2015, pp. 640–656.

[89] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher *et al.*, "Spectre Attacks: Exploiting Speculative Execution," *Communications of the ACM*, vol. 63, no. 7, pp. 93–101, 2020.

[90] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown," *arXiv preprint arXiv:1801.01207*, 2018.

[91] J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx, "Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution," in *27th USENIX Security Symposium (USENIX Security 18)*, 2018, pp. 991–1008.

[92] G. Chen, S. Chen, Y. Xiao, Y. Zhang, Z. Lin, and T. H. Lai, "SgxPectre: Stealing Intel Secrets from SGX Enclaves Via Speculative Execution," in *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 2019, pp. 142–157.

[93] P. Qiu, Q. Gao, C. Liu, D. Wang, Y. Lyu, X. Li, C. Wang, and G. Qu, "PMU-Spill: A New Side Channel for Transient Execution Attacks," *IEEE Transactions on Circuits and Systems I: Regular Papers*, 2023.

[94] M. Schwarz, S. Weiser, D. Gruss, C. Maurice, and S. Mangard, "Malware Guard Extension: Using SGX to Conceal Cache Attacks," in *Detection of Intrusions and Malware, and Vulnerability Assessment: 14th International Conference, DIMVA 2017, Bonn, Germany, July 6-7, 2017, Proceedings 14*. Springer, 2017, pp. 3–24.

[95] S. Hosseinzadeh, H. Liljestrand, V. Leppänen, and A. Paverd, "Mitigating Branch-Shadowing Attacks on Intel SGX using Control Flow Randomization," in *Proceedings of the 3rd Workshop on System Software for Trusted Execution*, 2018, pp. 42–47.

[96] A. Rane, C. Lin, and M. Tiwari, "Raccoon: Closing Digital Side-Channels through Obfuscated Execution," in *24th USENIX Security Symposium (USENIX Security 15)*, 2015, pp. 431–446.

[97] B. Coppens, I. Verbauwhede, K. De Bosschere, and B. De Sutter, "Practical Mitigations for Timing-Based Side-Channel Attacks on Modern x86 Processors," in *2009 30th IEEE Symposium on Security and Privacy (SP)*. IEEE, 2009, pp. 45–60.

[98] M. Salehi, G. De Borger, D. Hughes, and B. Crispo, "NemesisGuard: Mitigating Interrupt Latency Side Channel Attacks with Static Binary Rewriting," *Computer Networks*, vol. 205, p. 108744, 2022.
[99] J. Li, Z. He, A. S. Rakin, D. Fan, and C. Chakrabarti, "Neurobfuscator: A full-stack obfuscation tool to mitigate neural architecture stealing," in *2021 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*. IEEE, 2021, pp. 248–258.
[100] Y. Che and R. Wang, "Dnncloak: Secure dnn models against memory side-channel based reverse engineering attacks," in *2022 IEEE 40th International Conference on Computer Design (ICCD)*. IEEE, 2022, pp. 89–96.
[101] M. Zhou, X. Gao, J. Wu, J. Grundy, X. Chen, C. Chen, and L. Li, "Modelobfuscator: Obfuscating model information to protect deployed ml-based systems," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2023, pp. 1005–1017.
[102] T. Zhou, S. Ren, and X. Xu, "Obfunas: A neural architecture search-based dnn obfuscation approach," in *Proceedings of the 41st IEEE/ACM International Conference on Computer-Aided Design*, 2022, pp. 1–9.
[103] Y. Luo, S. Duan, C. Gongye, Y. Fei, and X. Xu, "Nnrearch: A tensor program scheduling framework against neural network architecture reverse engineering," in *2022 IEEE 30th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 2022, pp. 1–9.
[104] T. Nayan, Q. Guo, M. Al Duniawi, M. Botacin, S. Uluagac, and R. Sun, "Sok: All you need to know about on-device ml model extraction-the gap between research and practice," in *33rd USENIX Security Symposium (USENIX Security 24)*, 2024, pp. 5233–5250.
[105] G. Chen, W. Wang, T. Chen, S. Chen, Y. Zhang, X. Wang, T.-H. Lai, and D. Lin, "Racing in Hyperspace: Closing Hyper-Threading Side Channels on SGX with Contrived Data Races," in *2018 IEEE Symposium on Security and Privacy (SP)*, 2018.
[106] M.-W. Shih, S. Lee, T. Kim, and M. Peinado, "T-SGX: Eradicating Controlled-Channel Attacks Against Enclave Programs," in *Proceedings of the 24th Network and Distributed System Security Symposium (NDSS)*, 2017.
[107] S. Chen, X. Zhang, M. K. Reiter, and Y. Zhang, "Detecting Privileged Side-Channel Attacks in Shielded Execution with Déjá Vu," in *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, 2017, pp. 7–18.
[108] D. Gruss, J. Lettner, F. Schuster, O. Ohrimenko, I. Haller, and M. Costa, "Strong and Efficient Cache Side-Channel Protection using Hardware Transactional Memory," in *26th USENIX Security Symposium (USENIX Security 17)*, 2017, pp. 217–233.
[109] S. Constable, J. Van Bulck, X. Cheng, Y. Xiao, C. Xing, I. Alexandrovich, T. Kim, F. Piessens, M. Vij, and M. Silberstein, "AEX-Notify: Thwarting Precise Single-Stepping Attacks through Interrupt Awareness for Intel SGX Enclaves," in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023, pp. 4051–4068.
[110] Intel, "Asynchronous Enclave Exit Notify and the EDECCSSA User Leaf Function," https://cdrdv2-public.intel.com/736463/aex-notify-white-paper-public.pdf, 2022, white Paper.
[111] Fortanix, "Fortanix EDP," https://edp.fortanix.com/, [Online; accessed April 21, 2025].
[112] Enarx, "Enarx," https://enarx.dev/, [Online; accessed April 21, 2025].
[113] O. Enclave, "Open Enclave," https://openenclave.io/sdk/, [Online; accessed April 21, 2025].
[114] Gramine, "Gramine," https://gramineproject.io/, [Online; accessed April 21, 2025].
[115] Asylo, "Asylo," https://asylo.dev/, [Online; accessed April 21, 2025].
[116] C. F. for Confidential Computing, "Certifier Framework for Confidential Computing," https://github.com/ccc-certifier-framework/certifier-framework-for-confidential-computing, [Online; accessed April 21, 2025].
[117] Occlum, "Occlum," https://occlum.io/, [Online; accessed April 21, 2025].
[118] Intel, "Reference Architecture for Privacy Preserving Machine Learning with Intel SGX and TensorFlow Serving," https://www.intel.com/content/www/us/en/developer/articles/technical/privacy-preserving-ml-with-sgx-and-tensorflow.html, [Online; accessed April 20, 2025].
[119] ——, "Rising to the Challenge - Data Security with Intel Confidential Computing," https://community.intel.com/t5/Blogs/Products-and-Solutions/Security/Rising-to-the-Challenge-Data-Security-with-Intel-Confidential/post/1353141, [Online; accessed April 20, 2025].
[120] L. Wilke, J. Wichelmann, A. Rabich, and T. Eisenbarth, "SEV-Step: A Single-Stepping Framework for AMD-SEV," *arXiv preprint arXiv:2307.14757*, 2023.
[121] K. Ryan, "Hardware-Backed Heist: Extracting ECDSA Keys from Qualcomm's TrustZone," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 181–194.
[122] Z. Kou, W. He, S. Sinha, and W. Zhang, "Load-Step: A Precise TrustZone Execution Control Framework for Exploring New Side-channel Attacks Like Flush+Evict," in *2021 58th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2021, pp. 979–984.
[123] E. Aktas, C. Cohen, J. Eads, J. Forshaw, and F. Wilhelm, "Intel Trust Domain Extensions (TDX) Security Review," Google Technical Report, Tech. Rep., 2023.
[124] L. Wilke, F. Sieck, and T. Eisenbarth, "TDXdown: Single-Stepping and Instruction Counting Attacks against Intel TDX," in *ACM CCS 2024*, 2024.
[125] L. Gerlach, D. Weber, R. Zhang, and M. Schwarz, "A Security RISC: Microarchitectural Attacks on Hardware RISC-V CPUs," in *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2023, pp. 2321–2338.
[126] "Artifacts for DLS on Zenodo," https://doi.org/10.5281/zenodo.17038976, 2025.

APPENDIX A
ARTIFACT APPENDIX

A. Description & Requirements

DLS is a model extraction attack framework that analyzes latency traces to recover layer types and hyperparameters of DNN models. This artifact provides detailed instructions for accessing the source code and dataset used in our evaluation, which applies DLS to models implemented in the Darknet library, as well as for running and evaluating the attack.

1) How to access: The artifact is available in our repository [45] and on Zenodo [126].

2) Hardware dependencies: A commodity desktop or laptop is enough to run the artifact; NVIDIA GPU is optional.

3) Software dependencies: A Python environment is required to run the scripts. CUDA is optional. We tested the artifact on an Intel Zeon server running Ubuntu 22.04.

4) Benchmarks: Since DLS is specific to each deep learning library, this artifact targets the Darknet library in particular. The attack takes as input instruction latency traces collected during the execution of Darknet DNN models.

B. Artifact Installation & Configuration

There are four steps to install the artifact: (i) We first require that the repository is cloned to a local directory via `git clone`, and then follow the instructions in `README.md` from the repository. (ii) Navigate to the repository directory and setup Python virtual environment by executing `python3 -m venv .venv`; (iii) Install all the dependencies by executing `python3 -m pip install -r requirements.txt`; (iv) Extract the dataset into the directory using `tar xvfz dls_darknet_dataset.tar.gz`.

C. Experiment Workflow

This artifact test DLS with Darknet. The scripts are based on analysis of the Darknet library, as described in §VI.

DLS consists of four main steps, as illustrated in Figure 1: (i) Preprocessing, (ii) Function Identification, (iii) Basic Block Identification, and (iv) Execution Flow Mapping. Since preprocessing (Step (i)) and learning the L2F and L2BB models (part of Step (ii) and (iii)) take time, we include a preprocessed dataset and pretrained models in the artifact.

The Function Identification and Basic Block Identification steps aim to recover function-level execution flows (FEFs) and basic block-level execution flows (BBEFs), respectively. The scripts then check the accuracy of FEFs, BBEFs, and the final execution flow mapping. There are two experiments: the first (script: `fef_identification.ipynb`) shows the accuracy of FEFs and the related mapping (layer types and some hyperparameters); the second (script: `bbef_identification.ipynb`) shows the accuracy of BBEFs and their mapping (hyperparameters).

D. Major Claims

- (C1): DLS can identify FEFs that can subsequently be used to recover the model architecture (layer types and hyperparameters). The L2F model classifies the function corresponding to each latency value. The identified functions are then assembled into FEFs. These recovered FEFs enable accurate inference of the model's layer types and several hyperparameters. The results obtained from running this experiment correspond to those shown in Figure 4(a) and Figure 5(a) (E1).
- (C2): DLS can identify BBEFs that can subsequently be used to recover the model architecture, specifically hyperparameters. The L2BB model reconstructs the BBEF for a given function. The recovered BBEFs enable accurate inference of hyperparameters listed in Table II. The results of this experiment correspond to those presented in Figure 4(b) and Figure 5(b).

E. Evaluation

It consists of two experiments: FEF identification and mapping, and BBEF identification and mapping.

1) Experiment (E1): [FEF Identification and Mapping] [2 compute-hour]: This experiment runs the L2F model to identify FEFs and evaluates the accuracy of the recovered DNN architecture (layer types and hyperparameters).

[Preparation] Use the `fef_identification.ipynb` script with the pretrained L2F model in the `saved_model` directory and the input test traces from the `labeled_data` directory. FEF mapping information is provided in the `seq_data` and `cfg` directories. The script automatically detects and loads the necessary files.

The input trace contains four columns:

- `rip`: The collected program counter value, used for accuracy verification. It can be mapped to the binary `enclave.signed.so.darknet`.
- `mean`: The averaged instruction latency, which is used for FEF identification.
- `label`: The ground truth function group label, used in the identification.
- `func_label`: The ground truth function label, used only for preprocessing.

[Execution] Execute each cell in the script sequentially. Each cell produces output indicating whether the stage completed successfully. The script is structured into seven stages as follows:

- Stage 1: It loads the L2F model (in the `saved_model` directory) and 100 input trace files (in the `labeled_data` directory). For each trace file `tracename`, it writes the classification output to `log/logid_tmp_dir/tracename` and displays classification accuracy. This is the most time-consuming step in this experiment.
- Stage 2: It loads the L2F mapping data and saves cleaned traces as `log/logid_tmp_dir/stripped_tracename`.

- Stage 3 and 4: It identifies FEFs from the cleaned traces and performs FEF mapping. Intermediate data is stored in the `log/logid_tmp_dir/seq_data` and final mapping results are logged to `log/logid_tmp_dir/result/fef_result.log`.
- Stage 5: It parses the mapping result log and calculates architecture identification accuracy.
- Stage 6: It computes the error rate of the FEF sequence.
- Stage 7: It adds function labels to `log/logid_tmp_dir/stripped_tracename` based on the recovered FEFs for use in the BBEF identification experiment.

The output produced by the cell is as follows:

```
[INFO] Stage1: Identifying Function...
Total training, validation, and testing data:0 0 100
Loaded fef_darknet model from disk
[INFO] Pre-trained Model loaded
[INFO] Inference Step...
[LOG] testing model...  100
  ...
```

[Results] The result file `fef_result.log` includes the number of identified layers for each DNN model trace (`identified sequence`), the recovered FEF sequence (`recovered seq`), and the ground truth FEF sequence (`groundtruth seq`). In sequence, one character represents one function name, and the mapping between characters and original function names is provided in the `dictionary.log` file.

Stage 5 outputs accuracy statistics across 100 traces, including the total number of layers (per layer type), the number of layers whose types and associated hyperparameters are correctly identified with zero edit distance, those correctly identified with a non-zero edit distance, *etc*. Also, it reports the same statistics for each individual file (DNN model). At the end of the output, the accuracy is computed and displayed; that corresponds to Figure 4(a). Stage 6 reports the FEF sequence error rates, corresponding to Figure 5(a).

An example output is shown below. In this case, the accuracy for the fully connected layer type is 99%:

```
...
convolutional 0.9818181818181818
avgpool 1.0
maxpool 1.0
connected 0.98989898989899
```

For reference, the output files from the FEF identification experiment are provided in `log/cached_tmp_dir`.

2) Experiment (E2): [BBEF Identification and BBEF Mapping] [12 compute-hour]: This experiment performs the L2BB model to identify BBEFs and evaluate the accuracy of the reconstructed DNN architecture.

[Preparation] Use the `bbef_identification.ipynb` script with the pretrained L2BB model in the `hmm_query` directory and the post-processed files `log/logid_tmp_dir/stripped_tracename` generated from the previous experiment (E1). The script automatically loads all required files. The BBEF mapping logic, as shown in Table II, is hard-coded in the notebook cell.

[Execution] Run each cell in the script sequentially. Each cell reports whether the stage completes successfully or not. The script is composed of four stages:

- Stage 1: It loads the L2BB models (from the `hmm_query` directory) and 100 input trace files (from the `log/logid_tmp_dir/` directory). Each function uses a separate L2BB model.
- Stage 2: It performs basic block identification. This is the most time-consuming stage. It saves the ground truth and identified BBEFs in the `log/logid_tmp_dir/result/bbef_data/`. If these results already exist, they will be reused. If interrupted, the process resumes from the last checkpoint. To skip this stage, use the preprocessed results in the `log/cached_tmp_dir/result/bbef_data/` directory.
- Stage 3: It parses the result files and computes the architecture recovery accuracy.
- Stage 4: It parses the result files and computes the error rate of the recovered BBEF sequence.

[Results] The generated BBEF sequence files in `log/logid_tmp_dir/result/bbef_data/` contain two types of data: the list of basic block states and their corresponding indices. Each state name is a concatenation of program counter (`rip`) values.

Stage 3 computes the accuracy of the recovered hyperparameters across 100 traces, corresponding to Figure 4(b). Stage 4 computes the sequence error rates shown in Figure 5(b).

For example, the result of Stage 3 is shown below. In this case, the hyperparameter accuracy for the number of filter $\#\text{filter}_{\text{Conv}}$ is approximately 96%:

```
...
Hyperparameter accuracy
#filter_{Conv}   : 0.9636363636363636
fsize_{Conv}     : 0.9939393939393939
...
```

For reference, the result files from the BBEF identification experiment are available in `log/cached_tmp_dir`.