DeepSteal: Advanced Model Extractions Leveraging Efficient Weight Stealing in Memories

Adnan Siraj Rakin*[1], Md Hafizul Islam Chowdhuryy*[2], Fan Yao[2] and Deliang Fan[1]

* *Co-First Authors with Equal Contributions*
[1]Department of Electrical,Computer and Energy Engineering, Arizona State University
[2]Department of Electrical and Computer Engineering, University of Central Florida

***Abstract*—Recent advancements of Deep Neural Networks (DNNs) have seen widespread deployment in multiple security-sensitive domains. The need of resource-intensive training and use of valuable domain-specific training data have made these models a top intellectual property (IP) for model owners. One of the major threats to the DNN privacy is model extraction attacks where adversaries attempt to steal sensitive information in DNN models. Recent studies show hardware-based side channel attacks can reveal internal knowledge about DNN models (e.g., model architectures) However, to date, existing attacks cannot extract detailed model parameters (e.g., weights/biases). In this work, for the first time, we propose an advanced model extraction attack framework *DeepSteal* that effectively steals DNN weights with the aid of memory side-channel attack. Our proposed DeepSteal comprises two key stages. Firstly, we develop a new weight bit information extraction method, called *HammerLeak*, through adopting the rowhammer based hardware fault technique as the information leakage vector. *HammerLeak* leverages several novel system-level techniques tailed for DNN applications to enable fast and efficient weight stealing. Secondly, we propose a novel substitute model training algorithm with *Mean Clustering* weight penalty, which leverages the partial leaked bit information effectively and generates a substitute prototype of the target victim model. We evaluate this substitute model extraction method on three popular image datasets (e.g., CIFAR-10/100/GTSRB) and four DNN architectures (e.g., ResNet-18/34/Wide-ResNet/VGG-11). The extracted substitute model has successfully achieved more than *90 %* test accuracy on deep residual networks for the CIFAR-10 dataset. Moreover, our extracted substitute model could also generate effective adversarial input samples to fool the victim model. Notably, it achieves similar performance (i.e., $\sim$ 1-2 % test accuracy under attack) as white-box adversarial input attack (e.g., PGD/Trades).**

***Index Terms*—rowhammer, model extraction, bit leakage, adversarial attack**

I. Introduction

The recent development of deep learning technologies has made them an integral part of our daily life. This widespread application of Deep Neural Networks (DNNs) includes but is not limited to image classification [1], image detection [2] and speech recognition [3], many of which are deployed in security-sensitive settings [4]. DNN models typically take a tremendous amount of resources to train, and in many cases the training relies on the use of valuable domain-specific data. As a result, DNN models are regarded as the *top intellectual properties* (IP) for machine learning (ML) service providers and model owners [5]. With the rapid development of system and hardware level attack vectors that can compromise and tamper computing systems [6]–[10], understanding and investigating the security of deep learning systems has become imperative.

Model extraction attacks aim to infer or steal critical information from DNN models to achieve certain malicious goals [11]. Particularly, active learning is a popular approach in recovering the performance of a victim DNN model [12]–[18]. These methods primarily use input and output scores to recover the victim model's performance. It is challenging to extract the exact internal decision boundary only with input-output scores, which is especially the case for DNNs with complex and deep structures [11], [19], [20]. These techniques typically come with tremendous training overhead and substantial attack costs because of heavy model queries.

Recent advances in hardware-based exploitation have showed that adversaries can use side channel attacks to gain sensitive information in a target system [21]–[24]. Particularly, several works have shown that DNN model configuration parameters and structure information can be extracted by leveraging microarchitecture attacks [25], physical side channels (e.g., through observing power consumption or EM emanations [26]), and bus snooping attacks [27] with very high accuracy. Hardware-based attacks can be extremely dangerous as they allow adversaries to *directly gain internal knowledge* about the victim DNN models. However, most existing hardware-based DNN attacks focus on inferring high-level model information (i.e., commonly model architectures)[1]. It remains uncertain whether accurate information about model weights—the core information of DNN models, can be effectively exfiltrated via hardware-based side channel exploits. Note that acquisition of such information can potentially further extend DNN adversarial capabilities, including constructing substitute models with high accuracy, reproducing model fidelity (i.e., identical prediction as to the victim model), and bolster transfer adversarial attacks [18]. In this paper, we aim to answer the following question: *Is it possible for an adversary to perform advanced model extractions through stealing weight parameters using hardware-based attacks?*

While obtaining model weights can be useful intuitively, there are a few major challenges from the attacker's perspective to efficiently capture and effectively utilize such infor-

[1]Recently, [28] shows an attack that recovers a whole model by monitoring the plaintext packets transmitted over the PCIe-bus. Such attack is essentially an *overt channel* that can be mitigated via traffic encryption.

mation. *First*, although fine-grained secret leakage has been widely shown to be plausible in many non-ML applications through hardware-based side channels (e.g., microarchitecture attacks in particular [22], [29]–[32]). To date, no effective side channels have been demonstrated to exfiltrate detailed model weights due to the lack of distinguishable control and data-flow dependencies in DNN applications. Second, DNN models are often extremely large (with millions of parameters), even with a hardware-based attack vector that can recover certain model weight information, it is typically impractical to assume that *the entire weights* can be exfiltrated in practical settings. Moreover, prior works [33], [34] have proved that variations on only tens, out of millions, weight parameters will completely malfunction a DNN model. In this case, whether partial information about model weights can be effectively leveraged to build a stronger model extraction attack is uncertain. The final challenge involves how to design highly optimized ML techniques based on the obtained unique and partial model weight knowledge for different attack objectives.

In this paper, we present *DeepSteal*, an advanced model extraction attack framework using efficient model weight stealing with the aid of hardware-based side channels. The objective of our attack is to recover (partial) weight parameters of a target DNN model (i.e., victim model), which will be harnessed to build useful substitute models using novel learning schemes. At a high level, DeepSteal consists of two key stages. To address the aforementioned first challenge, **in the first stage**, we develop a novel hardware-based side channels that can exfiltrate partial bit information of model weight parameters, called *HammerLeak*. Particularly, we leverage the well-known rowhammer fault attack [9] as the information leakage vector. Our exploitation is motivated by prior studies showing that the occurrence of rowhammer-induced fault in a memory cell highly depends on the data pattern of its neighbouring bits [7], [34]. While such a phenomenon was first used by the prior work in [7] to successfully steal crytpo keys, we note that such technique is ineffective in stealing secrets in bulk and also cannot be applied in the context of DNNs. We, therefore, propose a set of system-level rowhammer optimization schemes that enable *fast and efficient exfiltration* of partial model weights tailored for DNN application platforms. After recovering the partial information at stage-1, the weight search space of a victim model still remains high. For example, even after recovering 90 % of the bits in a large model like VGG-11 [35] (i.e., 1056 Million bits for an 8-bit model), the attacker still needs to train the recovered model with limited data to restore the remaining 10 % bits (i.e., 105.6 million bits). Therefore, to address these additional challenges, **in the second stage**, we propose a novel substitute model training algorithm with *Mean Clustering* weight penalty. The purpose of such a loss penalty term is to utilize the recovered partial weight bits for effectively guiding the substitute model training. Subsequently, DeepSteal produces a substitute model which is expected to achieve similar accuracy as the victim model with high fidelity. Moreover, the trained substitute model could help to mount strong adversarial input attacks on the victim model as well. We summarize the major contributions of our work as follows:

1) For the first time, we investigate a *new model extraction attack* with the assistance of state-of-the-art side channels in memories that can exfiltrate detailed but partial information of DNN model weight parameters.
2) We develop HammerLeak, a system attack technique that utilizes fault-based information leakage through rowhammer to efficiently steal partial model weight parameters at scale. To make rowhammer-based side channel applicable and efficient for attacking DNNs, we propose several novel system-level techniques including: ❶ using new memory layout for hammering, ❷ implementing efficient DNN weight pages massaging through *inference activity anchoring* and *batched page releasing*, and ❸ reverse-engineering backend model memory management (in PyTorch) to enable mapping between physical location of leaked bit to logic location in the model.
3) Leveraging the leaked bit profile from HammerLeak, we propose a novel training algorithm for the substitute model with *Mean Clustering* weight penalty. It first conducts a data filtering process of the leaked bits to construct a profile consisting of projected searching space for each weight, which helps the optimization of substitute model training with reduced optimization space compared with full scale (i.e., no leaked weight info). Then, a Mean Clustering penalty term is added to the cross-entropy loss during training, penalizing each weight to converge near the mean of the projected range, for achieving a substitute prototype of the victim DNN model with comparable accuracy and high fidelity.
4) We build an end-to-end DeepSteal framework and demonstrate its efficacy on four popular DNN architectures (e.g., ResNet-18/34, Wide-ResNet-28, VGG-11). We also evaluate our attack on three standard image classification datasets (e.g., CIFAR-10, CIFAR-100, GTSRB). For example, the extracted substitute model has successfully achieved more than *90* % test accuracy on deep residual networks for the CIFAR-10 dataset. Moreover, DeepSteal attack could achieve similar as white-box adversarial input attack performance (i.e., degrading the model accuracy to 1-2 % under attack).
5) Finally, for the first time, we demonstrate the effectiveness and efficiency of recovering Most Significant Bit (MSB) only from a victim model. With our DeepSteal, it can generate effective adversarial examples with similar attack efficacy as a white-box attack (i.e., 0.16 %).

II. Background

A. Model Extraction

Model extraction is an emerging class of attacks in Deep Learning applications. It jeopardizes the privacy of the deployed victim model by leaking confidential information (i.e., model architecture, weights, biases, etc.). An ideal model extraction attack would expect to extract the exact copy of the victim model. For a task, the input and output pair data $(X, Y) \in \mathbb{R}$ can be drawn from the true distribution D_A to train a DNN model M_θ with parameters of θ. In this work, we designate this model M_θ as the *victim model*. To extract the

exact model, the attacker will attempt to recover a theft model $\hat{M}_\theta$, such that $M_\theta = \hat{M}_\theta$. However, such an identical model (i.e., same architecture and parameter) stealing is practically challenging if not impossible [11].

Algorithm-based Model Extraction. To overcome the challenge of stealing an identical DNN model, prior works [11], [12], [36] have defined some possible realistic approaches to extract DNN model information. In Table I, we summarize the prior DNN model extraction works into three major categories. First, in *direct recovery* method, the attacker attempts to reconstruct the victim DNN model using DNN output scores and gradient information. These works [11], [19], [20] have solved layer-wise mathematical formulation and internal functional representation to recover weights. In this setting, the goal of the attacker is to create a functionally equivalent model which is given an input $x \in X$, the recovered model $\hat{M}_\theta$ should follow: $M_\theta(x) = \hat{M}_\theta(x)$. This objective is a weaker version of the exact model extraction method. But it remains a difficult route to succeed in model extraction, as prior works [11], [19], [20] have failed to show a successful attack for over 2-layer neural network.

In the second approach (i.e., *learning*), papernot et. al [18] first proposed substitute model neural network training using input and output pairs of a victim DNN model to mount transferable adversarial input attack. In contrast, recent works [11]–[17], [37] aim to achieve high accuracy or high fidelity on a task using active learning methods. If the attacker prioritizes task accuracy, then the goal is to construct $\hat{M}_\theta$ such that the probability of $[\arg\max \hat{M}(x) == y]$ (i.e., true label) is being maximized. As for fidelity extraction, given a similarity function S(.), the goal is to construct a model $\hat{M}_\theta$ such that the similarity index $S(\hat{M}_\theta(x), M_\theta(x))$ between the output of the victim and substitute model is maximized. One of the major drawbacks of the learning-based model extraction approach is the high requirement of input query and access to the victim model's output score/predictions.

TABLE I: Summary of the existing model extraction methods.

Type	Attack	Goal
Direct/Mathematical Recovery	[11], [19], [20]	Functionally Equivalent
Active Learning/Learning	[11]–[18], [37]	Task Accuracy/Fidelity
Side channel & Learning	[26], [27], [38]–[43]	Functionally Equivalent/Fidelity

Side Channel Attacks on DNNs. There has been a large body of studies on hardware/microarchitecture side channel exploitation where attackers can leak confidential system information through power, EM and timing information on various platforms [21], [44]–[47]. Recent works have demonstrated that such attack vectors can also be applied to exfiltrate sensitive DNN information [25]–[27], [38]–[41]. Among the existing techniques, side channel attack is a more practical strategy to steal sensitive information about a deeper (i.e., many layers) victim model. Usually, the goal of side channel attack is to produce a functionally equivalent model or achieve high fidelity on a dataset. To achieve this, they often supplement side channel attacks with a learning scheme to train a substitute model using the leaked parameter information. This substitute model can later generate adversarial input samples with high transferable properties to attack the victim model more efficiently [27]. Note that these side channel attacks can only recover model architecture information accurately, but to date, no attacks can steal information about model weights.

Fig. 1: Data dependency for inducing a rowhammer fault. Here, based on the presence of bit flip in the *attacker-controlled* vulnerable bit in target row (T_r), data from adjacent row from victim program can be inferred.

In summary, among the three popular directions of model extraction attack, our proposed attack surface falls into the category of side channel attack, leveraging the memory-based fault injection attack rowhammer [9]. Our goal is to use our proposed DeepSteal framework to reproduce a copy of the victim model (i.e., substitute model) with high accuracy and high fidelity on a task. Finally, such a substitute model can later generate adversarial samples with high transferable properties to the victim model.

B. Rowhammer Attacks

Rowhammer is a software-based fault-injection attack that exploits DRAM disturbance errors via user-space applications [9]. Particularly, it has been shown that accesses (e.g., activations) to certain DRAM row can introduce electrical disturbance to the DRAM cells in its neighboring rows, which accelerates the leakage of their charges in the capacitors [9], [48]. An attacker can intentionally activate one DRAM row (whose data belongs to the attacker) frequently enough (i.e., hammering) that will eventually cause certain cells in neighboring rows to lose sufficient charge, leading to bit flips in memory. Such attacks have been successfully demonstrated on commercial-off-the-shelf DRAM modules even with the presence of ECC features [49], [50]. There are mainly three hammering techniques proposed in the literature: a) double-sided hammering [8], [9], [48], [50], [51]: where two adjacent aggressor rows to the target row creates a sandwich pattern with target row as inner layer and the aggressors rows repeatedly activated; b) single-sided hammering [51]: where one adjacent row to the target row and another random row are activated repeatedly; and c) one-sided hammering [49]: where one adjacent row is repeatedly activated maintaining a fixed frequency to exploit *close-page* DRAM policy (i.e., system policy that automatically deactivates the activated row if not accessed within a determined time frame). Among all those techniques, double-sided hammering is the most effective in inducing DRAM faults since it introduces the strongest disturbance.

There are a large body of works demonstrating many variants of Rowhammer attacks. Most of them focus on

tampering with the integrity of victims, including privilege escalations [51], system denial of service [52] and more recently faulting DNN model parameters [34], [53]. Several works also use rowhammer as the fault injection attack vector to recover crypto-keys through differential fault analysis [54]. Until recently, RAMBleed [7] reveals that the rowhammer fault characteristic can be leveraged to carry out *information leakage* attacks that directly infer information about certain victim secret. This attack leverages the fact that a column-wise data dependence is required to successfully flip a bit for a *known vulnerable memory cell*. Notably, under the most effective double-sided hammering, a bit flip for a vulnerable cell can succeed with high confidence if its upper and lower bits in the same column (e.g., in the aggressor rows) store the opposite bits (`1-0-1` or `0-1-0`), or fail if such pattern is not in place. Figure 1 illustrates such a data dependency for cell with bit flip vulnerability (in the $0 \rightarrow 1$ direction). As we can see, for the victim page in the middle with a vulnerable bit set to '0', a bit flip would only occur if the direct top and bottom bits are set to '1s', thus achieving the column-wise striped pattern. In RAMBleed, the attacker puts his own page in the middle row that has the vulnerable cell, and manages to trigger the placement of two copies of victim's secret in the corresponding aggressor rows. By observing whether a bit flip occur in his own page after hammering, the attacker can infer the secretive bit (i.e., RSA keys) from the victim stored in the aggressor rows. We note that exploitation of rowhammer in the domain of information opens new direction in terms of information security beyond the scope of integrity tampering.

III. Threat Model

The attacker targets on exfiltrating internal information (i.e., model weights) from deep learning systems by exploiting the underlying hardware fault vulnerabilities in modern computing systems. The attacker can control a user-space un-privileged process that runs on the same machine as the victim DNN service. We assume that the deep learning system is deployed on a resource-sharing environment to offer ML inference service. Such application paradigm is becoming popular due to the prevalence of machine-learning-as-a-service (MLaaS) platforms [55]. Our proposed framework manifest as a *semi-black box* attack where the adversary does not have any prior knowledge of the model weights. However, the attacker is aware of some key model architecture information, including model topology and layer sizes. We note that such an assumption is legitimate, as prior works demonstrate many practical ways to recover model architecture information through various side channel exploitation (e.g., through caches [5], memory bus [27] and EM [39]).

In this work, we leverage the rowhammer fault attack vector that commonly exists in today's DRAM-based memory systems (i.e., DDR3, DDR4) as the side channel [7]. Specifically, the attacker takes advantage of the fact that bit flip in *vulnerable DRAM cells* only occur when the column-wise bit striping pattern exists. By leveraging such data dependency, the attacker can infer bits in the aggressor rows by observing if a bit flip occur in *his/her own address space* (i.e., in the middle row). In other words, the attacker do not directly tamper with the victim's memory (as shown in most traditional rowhammer attacks), therefore it remains stealthy as it does not incur system-level alarming event (e.g., crashes). The attacker may share certain read-only memory together with the victim DNN (e.g., ML platform binaries) either through library sharing or advanced memory deduplication feature supported in modern OS [56]. Our attack mainly utilizes double-side rowhammer since fault occurrence in such setup exhibits stable data dependency. We assume that proper confinement mechanism is put in place to disallow direct access of data across processes. We further assume that the operating system and the hypervisor are benign and appropriate kernel-space protection mechanism are deployed to avoid direct tampering of kernel structures [57].

For substitute model training, as depicted in Table II, we assume the attacker has no knowledge of gradients, and is denied access to DNN output scores/predictions, which is more strict than prior works [18], [58]. Meanwhile, following most recent related works [59], [60], the attacker has access to a publicly available portion (e.g., ≤ 10 %) of the labeled training dataset.

TABLE II: *A list of information accessible to the attacker for substitute model training at stage-2 (Figure 2).*

Attacker Information	Accessible
1. DNN Architecture	✓
2. HammerLeak recovered weight bits	✓
3. Gradient Computation	✗
4. Train/Test Data	✗
4. Victim model Output	✗
5. A portion of publicly available data (≤ 10 %)	✓

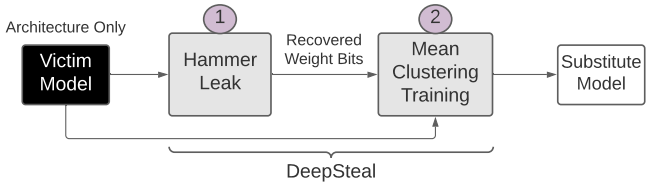


Fig. 2: **Overview** of our *DeepSteal* attack framework. *stage-1: HammerLeak* is a side channel attack to leak sensitive weight bit information through exploiting hardware fault vulnerabilities using rowhammer. *stage-2:* To use the recovered bit information efficiently, we train a substitute model using *Mean Clustering* weight penalty and recover a substitute prototype of the victim model

IV. Overview of DeepSteal

In this work, we propose a novel adversarial model stealing mechanism for Deep Neural Networks through rowhammer. An overview of our proposed attack framework, *DeepSteal*, is presented in Figure 2. It has two key components: i) an

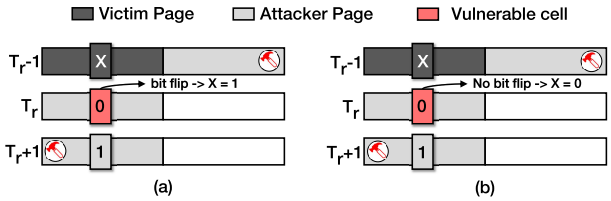


Fig. 3: HammerLeak attack leaking victim secret using single victim page. (a) Bit flip observed when victim's bit is `1`, (b) Bit flip not observed when victim's bit is `0`.

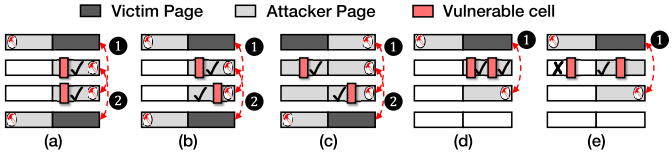


Fig. 4: Efficient utilization of target row holding vulnerable bits. HammerLeak can utilize all V_c in (a) Adjacent T_r with same V_c location, (b) Adjacent T_r with different V_c location (mirror page), (c) Adjacent T_r with different V_c location (alternate page), and (d) Same T_r with different V_c location (same page). Only in the case of (e) Same T_r with different V_c location (different page), one page in T_r can be used.

efficient rowhammer-based weight-stealing module *Hammer-Leak* & ii) a substitute model training mechanism with novel *Mean Clustering* loss penalty. At stage-1 in Figure 2, we mount HammerLeak attack on inference infrastructure (i.e., a remote machine running on-demand inference using the DNN model) to recover weight bits corresponding to each layer of the model (more explanation about HammerLeak is discussed in Section V and case study in Section VIII-A. We continue the HammerLeak for many rounds until the desired portion of weight bits are recovered.

Once a portion of the neural network weight bits are leaked using HammerLeak, at stage-2, our goal is to use these leaked weight bit information and generate a substitute prototype of the victim model. To achieve this, we propose a novel neural network training algorithm with Mean Clustering loss penalty on the weights to constrain the trained substitute model weight parameters to be as close as possible to the recovered partial weight info, as well as minimizing the accuracy loss. The learned substitute model will pose the following properties: i) it has a similar test accuracy as the victim's model; ii) high fidelity; & iii) the attacker can use this substitute model to generate strong adversarial input samples to attack the victim model with higher efficacy. Next, we will describe the details of stage-1 and stage-2.

V. HammerLeak: Efficient rowhammer based Data Stealing

In this section, we present the *HammerLeak* framework, which is built on advanced rowhammer-based side channels to leak secretive data (model weigt bits) from the memory of victim DNN applications in bulk.

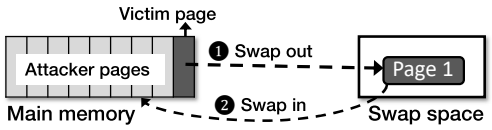


Fig. 5: Memory exhaustion technique to occupy large memory area by attacker, forcing victim data to *swap out* (❶). When victim accesses the data, its *swap in* (❷) to a different memory location

A. Rowhammer Information Leakage for DNNs

Though the rowhammer based attack in RAMBleed ([7]) demonstrates to steal information from the aggressor rows adjacent to the vulnerable cell, it requires the same page content in both aggressor rows. While it is possible that certain applications may launch multiple threads with each allocating memories to store a copy of secrets (e.g., as in the case of OpenSSH shown in [7]), we find that such duplication does not exist in modern machine learning frameworks (e.g., pytorch [61]). As such, the RAMBleed-style leakage manifest does not work for DNNs. To address this issue, we augment rowhammer by exploiting the same observation of data depen-decy, but with added capability of leaking victim bits with only one copy of victim page, which enables the rowhammer based information leakage applicable to any victim application. Figure 3 illustrates the major design of augmented rowhammer leakage. Unlike typical rowhammer based information leakage attack, the improved rowhammer leakage does not require two duplicated copy of the victim page, instead it substitutes one victim page with attackers page while still being able to leak secret bits from the victim page. To carry out the attack, the attacker first places the victim page to one of the adjacent aggressor rows (e.g., top aggressor row, T_{r-1}) and then places attacker's own page in the other aggressor row (e.g., bottom aggressor row, T_{r+1}) and the target row (T_r), as shown in Figure 3. The content of the attacker rows are controlled such that it creates a bit layout of `X-0-1` where `X` is the secretive victim bit, the middle bit is the V_c and the last one is another attacker controllable bit. Now, if the two aggressor rows are hammered, a bit flip will be observed in V_c if `X` is `1` as shown in Figure 3a (thus creating a `1-0-1` pattern), otherwise there will be no bit flips in V_c (Figure 3b). This same attack can also be carried out considering `X-1-0` bit layout, using the same technique.

Efficient Utilization of Vulnerable Rows. As the vulnerable DRAM cells are typically randomly distributed across a DRAM DIMM [62], it is possible that multiple vulnerable cells are present in the same row or in adjacent rows. In that case, the improved rowhammer leakage can create smart memory layout to increase the utilization of vulnerable bits in leaking secrets. Figure 4 enumerates all possible cases of multiple V_c in same or adjacent rows. In case of multiple V_c in adjacent rows (i.e., Figure 4a, Figure 4b and Figure 4c), we can setup the memory layout in such a way that it first leaks secret from first T_r (❶) by using the second T_r as an aggressor, then leaks from the second T_r by using the first T_r

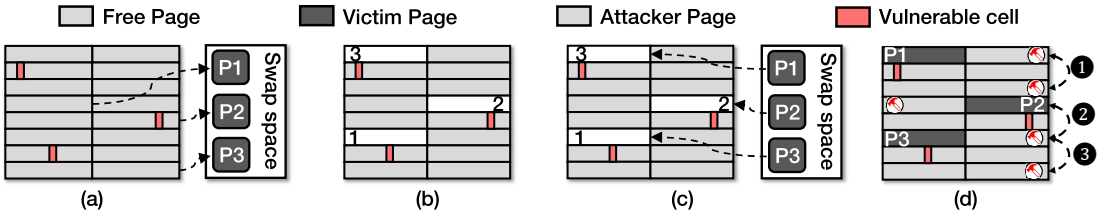


Fig. 6: Overview of the operations in the HammerLeak fraemwork.

as an aggressor (❷). In case of multiple V_c in the same page (i.e., same T_r as in Figure 4d), we can leak multiple secret bits at the same time by setting proper bit layout in the T_r and attacker-controller aggressor. Only in case of different V_c is different page of the same T_r, we can use only one page at a time since the attacker needs to control a page in the aggressor row containing victim page in order to carry-out the hammering (Figure 4e). In general, in a DRAM row with n individual pages, we can leak from upto $n-1$ pages while keeping one page as row activation page used for hammering. Note that this is considering a system with a *single-channel* memory which contains two 4KB pages per physical row. In systems with *dual-channels*, even more bits could be leaked per physical page.

B. Victim Secret Leakage using `HammerLeak`

At a high level, HammerLeak is a multi-round attack framework. In each round the attacker first tweaks the placement of victim pages in appropriate location (i.e., adjacent to T_r) using memory exhaustion and deterministic victim relocation via Bitflip-aware page release and then recovering the secret bits using improved rowhammer leakage. We now discuss each of these steps in details.

As rowhammer bitflips are highly dependent on specific DRAM locations, the victim pages containing secret must be placed appropriately to create desired memory layout with vulnerable cells (i.e., V_c). This is challenging because: i) typically V_c locations are sparse and only a few bits can be leaked in each round of hammering. Hence, to leak *new* bits, the victim page must relocate in different physical region which is not in direct control by the the attacker. Note that prior memory massaging techniques such as page deduplication [54] or page cache evictions [49] do not work as we find out DNN platforms such as pytorch allocate anonymous pages for DNN model weights; and ii) the physical address of victim page is not accessible by an attacker running as user-space process. We propose a four-step HammerLeak framework tackling each of these challenges to augment a practical rowhammer based side channel.

Step 1: Anonymous Page Swapping. During this step, the attacker's goal is to evict victim application pages from main memory to swap space so that they later get relocated by the operating system the next time they are accessed by the victim. This procedure is illustrated in Figure 5 where the victim page *Page 1* is *swap out* of main memory to the swap space because of attacker occupying the physical memory (❶). Once the victim application accesses this page again, the OS *swaps in Page 1* to main memory in a new location (❷). This is the first step towards the attacker's goal of relocating the victim page in a new location so that he can leak new bits. To achieve this, the attack allocates large memory using `mmap` with `MAP_POPULATE` flag. This triggers the OS to move other data (including victim application pages) out of main memory to swap space. After the end of this phase (Figure 6a), the attacker occupies most of the physical memory space with victim pages stored in swap space.

Step 2: Bitflip-aware Page Release. After the memory exhaustion where the attacker process occupies the majority of the system memory, the attacker systematically releases pages for victim pages to relocate into. Given the knowledge of virtual to physical page frame mapping of attacker process (using the same memory massaging technique as memory template), the attacker first creates a list of potential pages for the victim to occupy as *aggressor* during rowhammer attack (i.e., pages that are adjacent to a T_r holding one or more V_c). During each round of HammerLeak, the attacker randomly chooses a predetermined number of pages from the list. Finally, the attacker releases the selected pages (i.e., by calling `munmap`) to release the pages (Figure 6b). Note that the ordering of page release (i.e., `1,2` and `3` in that order as illustrated in Figure 6c) is critical in this step as we discuss next.

Step 3: Deterministic Victim Relocation. The main goal of this step is to place victim pages in a predetermined location which is optimized to create appropriate memory layout for rowhammer leakage. In addition, this also ensures that the victim page location is known to the attacker so that the attacker can correlate leaked secret with exact data from the victim domain. For this purpose, we utilize the per-core page-frame cache structure called `per-cpu pageset` [63]. This structure is maintained by the Linux kernel and is used to hold recently released pages by that CPU core. This is a stack-like *last-in-first-out* (LIFO) structure that holds the recently freed pages by processes running on that core. When the OS needs to allocate a page for a process, it first checks the pageset corresponding to the physical core to obtain a free memory location. Note that this structure is shared between hyperthreads running on the same CPU core, hence an attacker can exploit the LIFO policy of pageset to deterministically place victim pages into the desired memory location by running on the same CPU core (either in round-robin scheduler or in simultaneous multithreading). In particular, based on the order of release of pre-selected pages by the attacker during *Step 2*, the location of victim page placements can be determined with extremely

high accuracy [34] as illustrated in Figure 6c. Here `P1` (page 1), `P2` and `P3` are allocated for the victim process in that order and these pages are placed in memory location `3,2` and `1` respectively following the reverse order of page release during Step 2.

Step 4: Recovering Secrets Using Rowhammer. After completing Step 3 in which the victim pages are placed in appropriate location, the attacker mounts our rowhammer-based side channel (as discussed in Section V-A) to steal victim data. Based on the flip direction of a specific V_c (i.e., flip in either `0`→`1` or `1`→`0` direction), we preset the V_c and adjacent attacker controlled aggressor row bit respectively to `0-1` (for `0`→`1`) or `1-0` (for `1`→`0`). After hammering, the attacker reads the value of V_c and examines for bit flips. An observed bit flip indicates that the adjacent bit in victim controlled aggressor row is the same as the preset bit in the attacker's controlled aggressor row. This way, the attacker can recover secret bits from all of the selected T_r iteratively to maximize the data leakage in one round of HammerLeak.

These four steps correspond to one round of HammerLeak attack on the victim application. HammerLeak is a chained operation where the attacker resumes the attack again to leak different bits and this chained operation is continued until the attack goal (e.g., a specific page recovery) is reached. Note that the attacker must choose different sets or different orders of targeted page (Bitflip-aware) for release in Step 2 in order to put victim pages into locations where it is possible to leak more bits. Since the attacker has fine-grained controlled over the placement of victim pages, he can optionally optimize the pages to release in *Step 2* depending on which data is yet to recover, instead of relying of randomly releasing pages. For example, if the attacker needs to recover bit offset `x` from a victim page, he can chose to release a page which is adjacent to a T_r with V_c in bit offset `x`. This greatly increases the efficiency of HammerLeak by reducing the number of rounds to execute to recover certain number of bits.

C. Batched Victim Page Massaging

While the one-time *bitflip-aware page release* for one round of HammerLeak is sufficient for victim applications with small memory footprint (i.e., the size of working set is less than 512 pages), this is not true for larger applications since per-cpu pageset has a fixed size. For applications that have large working set (i.e., larger than 2MB), deterministic victim page relocation cannot be guaranteed if all required pages by victim in one round is released at once during Step 2. This is because if the required number of pages exceeds the size of pageset, the pageset is overflown and it starts to release some of the free pages to global memory pool. At that point, we can no longer maintain the LIFO ordering of pageset, which could lead to the failure of deterministic victim page relocation.

We propose *batched victim page massaging* to address this issue, which periodically releases a small number of pages at specific point(s) of victim execution so that the pageset does not get overflown. Additionally, this allows the attacker to have better control over filtering non-secretive victim pages by placing them at unleakable locations. To determine *when* to release leakable pages, the attacker needs to monitor victim's activities that lead to secretive page accesses. To simplify this process, we assume that both the attacker and the victim application use the same library (which results in shared read-only physical pages [56]). For applications using open-source library (which is the case for majority of the cryptographic applications and machine learning applications), the attacker can analyze and determine the secret accessing execution flow through source/binary code examination (e.g., a specific function which performs computation using secretive data), otherwise the attacker needs to perform binary analysis. Then the attacker instantiates FLUSH+RELOAD at cerntern *anchor point* in that function [29][2]. We build a cache side channel based monitoring tool to setup function anchor point for tracking purpose. If library sharing is not possible, we can alternatively use PRIME+PROBE based anchors. Additionally, through experimentation or source code analysis, the attacker pre-determines how many non-secretive pages the victim allocates after accessing the anchored function and what is the victim access pattern (e.g., if there are non-secretive page access between a chunk for secret page access) and uses that information in releasing pages.

Putting everything together, *prior to the attack*, the attacker determines an anchor point in victim's execution path which symbolizes the victim accessing certain secret. The attacker also determines secret page access pattern (i.e., number of non-secretive pages before accessing the secret, P_b, secret page accesses, P_s, and intermediate non-secretive pages access between secrets, P_i). *During the attack phase*, the adversary monitors accesses to the anchor points. Once the victim access is detected, the attacker releases $P_b + P_s + P_i = P$ physical pages considering the reverses order of victim accesses (i.e., LIFO access). Out of these pages, only P_s pages need to be the victim's pages for information leakage. If P is larger than the size of per-core pageset, the attacker will choose a different anchor point, which effectively divides the victim secret page accesses to additional smaller chunks. By chaining several page batches for memory massaging, HammerLeak can eventually steal bits spanning *hundreds to thousands pages* in each individual round, which enables weight leakage for DNN applications with large memory footprint.

VI. Substitute Model Training with Mean Clustering

At Stage-2 of DeepSteal, we will leverage the bit information leaked by HammerLeak to learn a substitute prototype of the victim DNN model. The challenge is that the recovered weight bits are not complete, with a mixture of different significant bits. To fully leverage those leaked partial bit-wise data, we propose a novel substitute model training algorithm with a mean clustering loss penalty to reconstruct a neural network model, targeting for high accuracy & high fidelity. Moreover, this learnt substitute model will help the attacker

[2]The attacker repeatedly flushes the cacheline containing the function code and times the subsequent access to the same function. A short access time means victims has accessed this function, otherwise the victim did not access.

generate highly effective adversarial input samples to fool the victim model successfully.

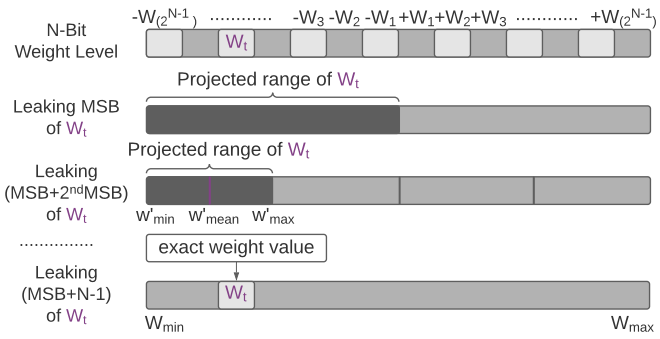


Fig. 7: *First row:* N-Bit quantized weight level; *Second row:* Once the MSB of a weight W_t in the victim model is leaked, we can narrow down the projected range of W_t in the substitute model; *Last row:* Leaking all the bits can track down the exact value of W_t for the substitute model training.

A. Hammer Leaked Data Filtering

At stage-1, HammerLeak recovers a portion of the neural network weight bit information scattered across different significant bits (i.e., from LSB to MSB) for each weight. However, not all of those recovered bits will be used for substitute model training, since it is a mixture of significant bits for each weight. As shown in Fig.7, for each weight, it is preferred to recover MSB first. Thus it forms a smaller and closed searching space (i.e., either positive or negative), rather than full-scale space, for this weight during substitute model training to minimize loss. With the knowledge of MSB, the 2^{nd} MSB or more following bits will further reduce the closed searching space. Otherwise, only recovering lower significant bits, without higher significant bits, does not provide much useful information about this weight's potential range. As a consequence, to use the leaked bit information effectively from stage-1 of HammerLeak, the attacker must filter and reorganize the leaked bits in a sequence from MSB to LSB. Therefore, in this work, before substitute model training, we sort out the leaked weight bits in the following sequence: MSB leaked, MSB + 2^{nd} MSB leaked, MSB + (2^{nd} & 3^{rd} MSB) leaked, and so on, to develop a profile for each weight with projected range, as described in Fig.7. Note that, if no MSB is recovered, the projected weight value range will be treated as full scale.

In Figure 7, we visualize the relationship between i) filtered bits (leaked by HammerLeak) information of weight from a victim model and ii) the expected range of that corresponding weight during the training of the substitute model. It shows gradually leaking more bit information (i.e., MSB, MSB+2^{nd} MSB,..) of one target weight W_t can help an attacker reduce the searching space of W_t during model training. We define this expected range as *projected range* of each weight in the substitute model.

B. Mean Clustering Optimization

Leveraging the profile of such projected range of each weight, we propose a novel training algorithm for the substitute model using *Mean Clustering* weight penalty. It applies an additional loss penalty to the cross-entropy loss during the training process. The Mean Clustering penalty term aims to penalize each weight to converge near the mean of the projected range.

To formally define the problem, let's consider the weight matrix of the victim DNN model at layer l to be $\hat{\boldsymbol{W}}^l$. Based on the leaked weight bits at this layer, the attacker can compute the projected range of each weight in the substitute model $\boldsymbol{W}^l$. The projected range can be represented as: $\boldsymbol{W}^l_{min}$ & $\boldsymbol{W}^l_{max}$ matrix; the minimum and maximum projected value matrix corresponding to each weight in $\boldsymbol{W}^l$. Using this closed range, the projected mean matrix $\boldsymbol{W}^l_{mean}$ is computed as: ($\boldsymbol{W}^l_{max}$ + $\boldsymbol{W}^l_{min}$)/2. Next, leveraging this mean matrix, we propose to design a Mean Clustering loss penalty as highlighted in Equation (1). This loss term is added to the inference loss $\mathcal{L}$ and the optimization process can be formulated as:

$$\min_{\{\mathbf{W}_l\}_{l=1}^{L}} \mathbb{E}_{\boldsymbol{x}}\ \mathcal{L}(f(\boldsymbol{x}, \{\mathbf{W}^l\}_{l=1}^{L}), \boldsymbol{y}) + \underbrace{\lambda \cdot \sum_{l=1}^{L}(||\mathbf{W}^l - \mathbf{W}^l_{mean}||)}_{\text{loss penalty for Mean Clustering}} \tag{1}$$

Here, λ is a hyper-parameter that controls the strength of the loss penalty and f(.) denotes the inference function of the DNN model for an input-label pair $(\boldsymbol{x}, \boldsymbol{y})$. The first term of the loss function in Equation (1) is a typical cross-entropy loss for neural network training using gradient descent. The purpose of the additional Mean Clustering loss penalty is to penalize each weight to converge near $\{\boldsymbol{W}^l_{mean}\}_{l=1}^{L}$.

C. Overall Training Algorithm.

We summarize our proposed training algorithm in Algorithm 1. After the filtering step, we divide the weights into three categories: *Weight Set-1:* Full 8-Bit recovered, *Weight Set-2:* Partial bit recovered (i.e., MSB + n; n= 0,...,6) & *Weight Set-3:* No bit recovered. For set-1, the attacker knows the exact weight value in the victim model. Hence, we will use the exact recovered value for the substitute model by freezing (i.e., set gradient to zero) them during training. The second set of weights is trained using the proposed loss function in Equation (1). And for set-3, we do not apply the Mean Clustering loss penalty (i.e., λ=0). Both set-2 & set-3 weights are trained using standard gradient descent optimization. During training, each time before computing the loss function in Equation (1), we update the projected mean matrix $\{\boldsymbol{W}^l_{mean}\}_{l=1}^{L}$ using the weights of current iteration. If any weight value exceeds the projected range, it will be clipped. Finally, for the last few iterations (e.g., 40), the model will be fine-tuned (i.e., $\lambda = 0$, no clipping & low learning rate) to generate the final substitute model.

Algorithm 1 *Substitute Model Training with Mean Clustering*

procedure TRAIN SUBSTITUTE MODEL (M_θ,$\hat{M}_\theta$,$\hat{\theta}^L_{l=1}$)
  Takes victim model architecture $\hat{M}_\theta$ as input
  Takes the leaked parameter list $\hat{\theta}^L_{l=1}$ as input.
  Randomly Initialize Substitute model M_θ.
  Perform step-1, data filtering of the leaked bits.
  for Each layer $l = 1, \ldots, L$ **do**
    Compute Initial $\boldsymbol{W}^l_{min}$ & $\boldsymbol{W}^l_{max}$ using $\hat{\theta}_l$.
    Estimate $\boldsymbol{W}^l_{mean}$ = ($\boldsymbol{W}^l_{min}$+ $\boldsymbol{W}^l_{max}$)/2.
  end for

  Re-Initialize model M_θ using following rules:
  Weight Set-1 (Full 8-bit recovered): Initialize the weights at the exact recovered value. This weight set will not be trained (i.e., set gradients to zero).
  Weight Set-2 (Partial bit recovered (i.e., MSB+n; n = 0, ..., 6)): Initialize the weights using the $\{\boldsymbol{W}^l_{mean}\}^L_{l=1}$ matrix & set λ suitably.
  Weight Set-3 (0 bit recovered): Random Initialization & set λ in Equation (1) to zero.

  for each training iteration **do**
    for each training batch $(\boldsymbol{x}, \boldsymbol{y})$ **do**
      Compute Loss using Equation (1).
      Perform a gradient descent step to update θ.
      Update $\boldsymbol{W}^l_{min}$, $\boldsymbol{W}^l_{max}$ & $\boldsymbol{W}^l_{mean}$.
    end for
    Clip $\{\boldsymbol{W}^l\}^L_{l=1}$ within the projected range.
  end for
  Return: Trained Substitute Model M_θ.
end procedure

VII. EXPERIMENTAL SETUP

A. Attack Evaluation Metrics

To evaluate the efficacy of our DeepSteal attack, we adopt three different evaluation matrices, i.e., accuracy of the substitute model, fidelity of the substitute model, and accuracy of the victim model under adversarial input attack.

a) Accuracy (%): It is the measurement of the percentage of test samples being correctly classified by the substitute model for a given test dataset. Note that, this is the same test data used for victim model. For an ideal successful model extraction attack, we expect the accuracy of the victim and substitute model to be almost identical.

b) Fidelity (%): We measure the *fidelity* as the percentage of test samples with identical output prediction label between the victim model and substitute model. This follows the definition of [11], where two models with high fidelity should agree on their label prediction for any given input sample. Ideally, an attacker should achieve 100 % fidelity, where the substitute and victim model agree on all the prediction output.

c) Accuracy Under Attack (%): It is defined as the percentage of adversarial test samples generated from substitute model being correctly classified by the victim model. It indicates the transferability of the adversarial examples as explained in prior [64]. Ideally, if the substitute model and victim models are identical, then adversarial samples transferred from the substitute model should achieve similar efficacy (i.e., accuracy under attack) as a white-box attack (i.e., the attacker knows everything about the victim model). In this evaluation, we use the popular projected gradient descent (PGD) [65] attack to generate adversarial samples on the substitute model. The PGD attack uses L_∞ norm, $\epsilon = 0.031$ and an attack iteration step of 7 for all three dataset.

B. Hardware Configuration

We train our DNN models using GeForce GTX 1080 Ti GPU platform operating at 1481MHz and deploy the trained models in an inference testbed. The HammerLeak attack is evaluated on the inference testbed equipped with Intel Haswell series processor (i5-4570) with AVX2 instruction set support. We collect bitflip profile of the memory modules (i.e., templating) used in target system to identify potential vulnerable locations in DRAM. Note that memory templating is considered as a standard processor for rowhammer attack and we leverage existing techniques as described in [7], [49], [50], [54]. The system is configured with a dual channel memory subsystem with one 4GB DDR3 DIMM (Hynix) in each channel. Our tested DIMMs have 71% of the pages containing at least one V_c and in total 0.017% of memory cells are vulnerable to bit flip. Compared to bit flip profiles computed in prior work showing multiple DRAM modules with more than 98% of all rows being vulnerable [62], our system has moderate level of vulnerability in rowhammer-induced bit flips.

C. Dataset and Architecture

We take three visual datasets: CIFAR-10 [66], CIFAR-100 [67] and German Traffic Sign Recognition Benchmark (GTSRB) [68] for object classification task. CIFAR-10 contains 60K RGB images in size of 32×32. Following the standard practice, 50K examples are used for training and the remaining 10K for testing. On the other hand, CIFAR-100 has 100 classes with 600 images in each class. Both CIFAR-10 and CIFAR-100 has same image size and test-train data split. For GTSRB dataset, we use 40k labelled images and split them 85-15 ratio for training and evaluation purposes. Each traffic sign image has a size of 112x112 and is distributed in 42 different class. For CIFAR-10 experiments, we used residual networks (e.g., ResNet-18/34,Wide-ReNet-28) [1], [69] and VGG-11 [35] architectures. For GTSRB and CIFAR-100, we adopted ResNet-18 and ResNext-50-32x4d [70] as the evaluation architecture respectively. For all experiments, the weights of each victim model are quantized into 8 bit.

Following the standard practice in [59], [60], we assume the attacker has access to a publicly available portion (i.e., $\sim$8%) of the train dataset. In our experimental setting, we used 4096 ($\sim$8%) train samples to train the substitute model for both CIFAR-10 & CIFAR-100 dataset. For GTRSB, we only used 2656 ($\sim$8%) train samples to train the substitute model based on our proposed Mean Clustering training method. We follow similar data distribution and experimental setting for the

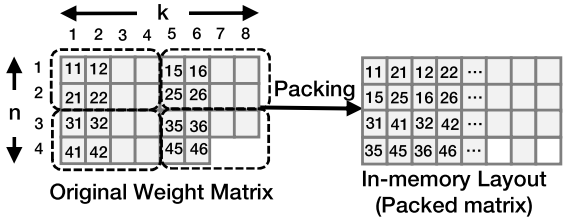


Fig. 8: Illustration of PyTorch weight packing (example using 4×2 size chunks) and organization of weights in memory. Each chunk is represented by dashed block and the small white blocks represents uninitialized location (no weight in this location).

baseline method (i.e., architecture only case) as well. During the experiments, to train the substitute model for all different cases, we train three individual models independently and report the average (e.g., accuracy,fidelity) of three rounds.

VIII. Evaluation

A. DNN Weight Recovery using HammerLeak

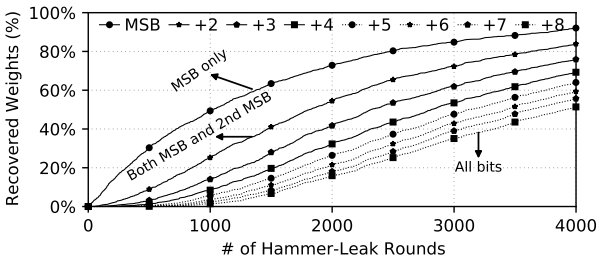
As a practical example of HammerLeak, we choose an inference server as victim application running inference on different DNN models using **PyTorch** framework. Depending on the hardware instruction set supported on the host system, PyTorch performs extensive optimization of core computational kernel of DNN. For example, PyTorch has direct compatibility with FBGEMM [71] backend for x86 platform and QNNPACK [72] backend for ARM platform to accelerate matrix computation process using vector instructions (i.e., AVX-2 or AVX-512). Since there are multiple platform- and hardware-specific optimization for each specific type of DNN model following different execution path, for the rest of this discussion, we use 8-bit quantized DNN models running inference using FBGEMM backend as a case-study. We use PyTorch `v1.7.1-rc3` with FBGEMM commit `1d71039` running on Ubuntu 20.04 as the platform of our investigation. Note that the similar inference execution flow can be obtained for other cases as well.

In-Memory Organization of DNN Weights. The accelerated vector instructions use single instruction multiple data (SIMD) for its internal calculation, this requires the operands used in these computation to be arranged in a specific layout (called *packed* layout which is essentially reorganization of matrix data into a data array format optimized for sequential access by computational kernel [71]). PyTorch hence creates an optimized packed layout for both operands of these computation (i.e., weights and inputs). Since DNN model weights remain unchanged throughout an inference operation, PyTorch creates a packed copy for DNN weights of each linear and convolutional layers during model initialization and uses that pre-packed structure (stored in main memory) for each inference operation. Whenever a new inference is requested, PyTorch uses the pre-packed weights from main memory to carry out the inference. This pre-packing operation is instantiated by `PackedLinearWeight::prepack` and `PackedConvWeight::prepack` functions for linear and convolutional layer respectively and the packing operation is handled by `fbgemm::PackBMatrix` (FBGEMM uses `PackB` to represent weight matrices internally). FBGEMM divides the weights of each layer into smaller chunks, and then each chunk is stored sequentially into memory, which is useful for both accelerating computation performance and optimizing cache accesses. For AVX2 systems, each of these smaller chunks has a size of 512×8, indicating all weights in a specific layer are divided into several 512×8 weight chunks and weights in one chunk are laid sequentially into memory using *column-major* format (data in columns are stored sequentially) forming a data array type layout which is different from regular memory layout of matrix that uses *row-major* format [73]. Figure 8 illustrates the packing and memory layout of the stored weights. Given the a specific memory byte in the in-memory layout, the actual weight location in original weight matrix can be easily determined. Then within each layer, the matrix computational kernel performs vector multiplication and accumulation to produce the output of that layer.

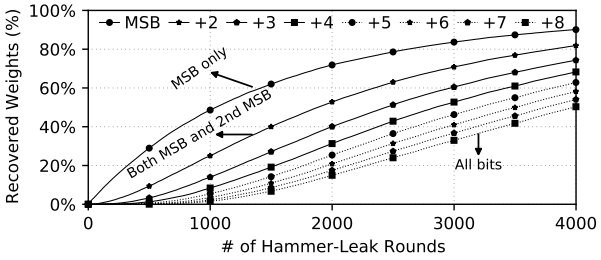
Mounting HammerLeak on PyTorch. We first use the `forward(<input>).toTensor()` method of DNN model as an anchor to monitor the beginning of an inference. This anchor will trigger the monitoring of subsequent anchors. During the inference operation, PyTorch uses the `apply_impl` function (corresponding to linear or convolutional layer) to instantiate the FBGEMM computation for each layer. We setup two anchor points (using FLUSH+RELOAD based monitoring) in each of the `apply_impl`. Since the computation of each layer is sequential, the first call to `apply_impl` represents the beginning of the first layer computation, while the second call denotes the second layer and so on. This is how we can distinguish between different layer computations and determine which layer is currently being executed. This passes a pointer to input and packed weights to the FBGEMM backend using `fbgemmPacked`. The `ExecuteKernel::execute` coordinates the generation of JIT code (`jit_micro_kernel_fp`) for computation by executing the `getOrCreate` function. We setup another anchor point monitoring the access to `getOrCreate` function. We use batched victim page massaging by releasing vulnerable pages once this anchor is triggered to release pages of small number so that per-cpu pageset does not get overflown, but still have sufficient pages released for all the ML pages accessed in that chunk computation to occupy.

B. HammerLeak Performance Analysis

In this section, we use ResNet-18 as one representative DNN model for HammerLeak analysis. ResNet-18 has 21 layers with 11 million weight parameters. We perform HammerLeak on this model to investigate the efficiency of our attack in recovering model parameters. We observe that at about 4000 HammerLeak rounds, HammerLeak can steal more than 90% of the MSB bits for model weights almost across all layers (with the lowest per layer recovery rate to be 88%). Figure 9 shows the percentage of weights with leaked `{MSB}` bits as well as percentage of weights with other bits simultaneously leaked together with the MSB bits (e.g., `{MSB+`2^{nd} `MSB}`) for two different layers. We observe that along with



(a) Layer 1



(b) Picking a random layer (Layer 5 as an example)

Fig. 9: Percentage of weights with `MSB` and `MSB+x` bits recovered where `+x` represents `x` number of consecutive higher order bits recovery (i.e., `+3` represents weights with all three `MSB` bits recovered).

MSB bits, the recovery rate for additional weight bits are also very high, with 55%-63% weights across all layers have the complete weight recovered. This shows the high efficiency of the proposed targeted HammerLeak attack when deployed with the augmented rowhammer leakage.

Figure 10 illustrates the distribution of weight percentages that has at least MSB bit recovered for all 21 layers. Depending on the attacker's goal (in our case, high percentage of weights with MSB exfilrated), HammerLeak can be completed much sooner than 4000 rounds. We observe most of the layers have half of the weights with MSB bit recovered within 1000 rounds of attack.

We further investigate the time spent during each of the steps of a HammerLeak round to determine the attack cost factors. We experimentally find that memory exhaustion (Step 1) and Bitflip-aware page release (Step 2) requires 12 seconds and 21 seconds respectively. The actual inference operation (which is Step 3 when the inference application accesses the ML pages) takes less than 1 seconds. Finally, by far the most expensive portion (in terms of latency overhead) is the actual bit leakage step (Step 4) where HammerLeak steals the model bits through rowhammer-based side channel. When considering bit leakage from pages that have at least one MSB bit offset in V_c (i.e., *MSB* configuration in Table III), each HammerLeak round will hammer about 11K rows on average, which takes on average aout 239 seconds. In contrast, without constraining MSB pages, recovering all bits (i.e., the *All Bits* configuration in Table III) requires hammering 17K rows on average (i.e., about 375 seconds). Note that only Step 2 and Step 3 have to be done simultaneously with the

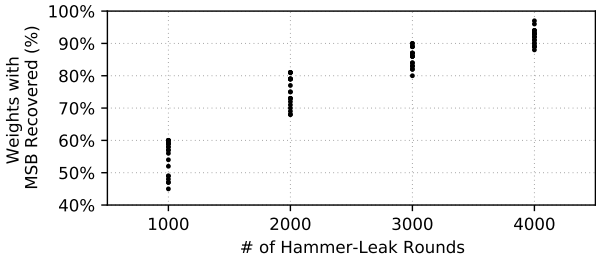


Fig. 10: Distribution of weights with `MSB` recovered across 21 individual layers of ResNet-18.

inference operation, the bulk of the operation (Step 4) can be done in-between multiple inferences by limiting the number of vulnerable pages to release in each round. In addition, instead of considering each vulnerable page while releasing, we can only release the physical pages that have at least one MSB offset leakable (i.e., MSB prioritization scheme). This way we can substantially improve the attack efficiency while leaking most of the influential bits. We experimentally found that using MSB prioritization scheme, HammerLeak can still recover 68% other bits along with recovering 92% of MSB bits.

C. DeepSteal Experimental Results: CIFAR-10

In Table III, we evaluate the performance of DeepSteal attack on CIFAR-10 dataset for three different architectures. Further, we show an ablation study showing the impact of using several rounds of HammerLeak attack information for DeepSteal attack. As a baseline method, we only compare with the architecture only case (i.e., 0-bit information leaked). For the baseline case, we assume the attacker only knows the victim model architecture. Then, a substitute model with the same architecture is trained using a similar setting (i.e., less than *8 %* available data). On the other hand, we treat the white-box case as the best-case scenario where the attacker knows every information (i.e., weights, biases and architecture) of the victim model. In summary, with more and more recovered weight bits, our DeepSteal achieves much improved accuracy, fidelity and adversarial example attack efficacy. For example, our substitute model can generate effective transferable adversarial example with similar efficacy (i.e., $\sim$*0 %*) as white-box attack for both ResNet-18 and ResNet-34.

In our evaluation, the residual victim models ResNet-18 & ResNet-34 have 93.16 % & 93.11 % inference accuracy, respectively. As shown in Figure 9, after 4000 rounds of HammerLeak attack, the attacker could recover 90 % of the MSB bits ($\sim$11.52 % of total bits). Next, by only using the leaked MSB bit information, the attacker can recover up to 88.65/89.42 % test accuracy for the ResNet (18/34) models. However, for All Bits case in Table III, even after paying an additional time cost (i.e., *1.66* $\times$), the performance of DeepSteal attack can hardly improve on residual models. In contrast, a larger (i.e., 132 Million parameters) model, such as VGG-11 with different architecture topology, highly benefits from the additional information of all the bits. For VGG, we observe a $\sim$*5 %* improvement by using all the filtered bits

TABLE III: *Summary of CIFAR-10 results for three different DNN architectures. We report two different cases of DeepSteal attack i) All Bits: where we use all the bit information (i.e., all 8 plots) plotted in Figure 9. According to this plot, for each # of HammerLeak attack rounds along x-axis, we take the percentage of bits recovered for all 8 plots (e.g., MSB,MSB+2nd MSB & so on). ii) MSB: We only use the MSB bit information labelled as MSB curve in Figure 9.*

Method			*ResNet-18*				*ResNet-34*			*VGG-11*		
# of Hammer-Leak Rounds	Method	Case	Time (days)	Accuracy (%)	Fidelity (%)	Accuracy under Attack (%)	Accuracy (%)	Fidelity (%)	Accuracy Under Attack (%)	Accuracy (%)	Fidelity (%)	Accuracy Under Attack (%)
Baseline	Architecture Only	-	-	72.68	73.59	62.58	72.59	73.24	63.53	70.9	71.52	62.08
1500	DeepSteal	All Bits	7.5	75.9	77.13	54.49	75.08	75.65	56.23	72.35	73.24	63.53
		MSB	3.9	77.0	78.24	54.8	76.97	77.89	55.14	72.81	74.04	58.27
3000	DeepSteal	All Bits	14.9	86.28	87.54	5.69	85.15	86.08	5.28	81.57	82.74	35.85
		MSB	7.8	85.39	86.97	12.49	87.03	88.2	7.61	77.24	78.48	37.6
4000	***DeepSteal***	All Bits	***17.3***	***89.15***	***90.64***	***1.89***	***88.02***	***89.22***	***1.69***	***84.67***	***86.55***	***17.2***
		MSB	***10.4***	***88.65***	***90.6***	***2.21***	***89.42***	***90.58***	***2.12***	***79.84***	***80.42***	***26.44***
Best-Case	White-box	-	-	93.16	100.0	0.0	93.11	100.0	0.0	89.96	100.0	4.63

in comparison to using MSB only. Similar to the 4000 round case, we observe a similar pattern in accuracy recovery at 3000 rounds of HammerLeak attack as well.

Next, we evaluate the adversarial attack performance for both 3000 and 4000 rounds of HammerLeak attack. In Table III, we show that our substitute model can transfer effective adversarial samples to the victim model across all three architectures. In particular, for ResNet models (e.g., 18/34), our substitute model generated adversarial examples demonstrate close to white-box attack efficacy (i.e., within *2-5 %* of the best case result). As for VGG model, which is already known as a robust architecture [65], our substitute model generated adversary reaches within *∼13-22 %* of an ideal white-box attack. Nevertheless, our attack efficacy still shows an improvement of about *∼45-60 %* across all three architectures in comparison to the baseline (i.e., architecture only) technique.

Finally, we consider an attack scenario where the attacker has a strict time budget. In this scenario, let's assume he/she can only afford to run 1500 rounds of HammerLeak attack while prioritizing MSBs (e.g., only *3.9* days of attack time). As summarized in Table III, even such a restricted attack can generate effective adversarial examples to lower accuracy under attack by *8%* for ResNet models and by *4%* for VGG-11 compared to baseline. One key observation for this low budget (i.e., 1500 round attack) attack is that attacker can generate a much more effective substitute model by only using MSB information rather than all the bits. The reason being with limited bit information (e.g., 50 % MSB only) putting strict penalization (i.e., mean clustering) on the weights during training does not help the substitute model accuracy. In fact, for VGG-11, it becomes worse than the baseline method. As a result, for DeepSteal attack with limited partial bit information, using the relaxation of the weight constraints (i.e., MSB only) can be more effective than using all the available filtered bits.

D. DeepSteal Experimental Results: CIFAR-100 & GTSRB

We evaluate DeepSteal attack on two relatively larger datasets as well. We chose CIFAR-100, which has *10×* larger output class size than CIFAR-10, and the GTSRB dataset has *12×* larger input dimension than CIFAR-10. In Table IV, we summarize the results of these two datasets.

For CIFAR-100, we attack a ResNext-50-32x4d victim DNN model. Here, the baseline (i.e., architecture only) method can recover the model accuracy only up to *32.81 %* which is *∼36 %* lower than the ideal case (i.e., white-box). Our attack can improve this baseline recovered accuracy by *∼26 %* and also the fidelity by *∼31 %*. Again, for GTSRB, our attack can recover the exact baseline accuracy with a high fidelity rate *98.77 %*. It shows the dataset with a larger output class (e.g., CIFAR-100) poses a tough challenge to recover the test accuracy of the victim model.

As for adversarial input attack, our substitute model can achieve *6.6 %* accuracy under attack on CIFAR-100. It shows almost similar attack efficacy (i.e., *0-6 %*) as a white-box attack while gaining a *36 %* improvement compared to baseline. For GTSRB, our substitute model can lower the accuracy under attack by *∼24 %* compared to the baseline. However, the attack efficacy still lags by *39 %* from the ideal case (i.e., white-box) mainly because of the input image dimension (112×112). As an adversarial image generated by the substitute model is less likely to succeed in attacking the victim model for a larger input dimension/search space.

E. Comparison to State-of-the-art Techniques

In Table V, we summarize the standing of our DeepSteal attack compared with existing model recovery methods for three different domains of applications. First, existing model regularization [11], [37] and data augmentation techniques [59], [60] are useful in training deep models with limited data. Compared to them, our substitute model achieves a much higher accuracy(i.e., *∼3%*). The only exception is Mix-Match [11], as it performs slightly better (i.e., *∼2%*) than our proposed technique. However, Mix-Match has adopted a combination of existing data augmentation and regularization techniques, which could also be integrated with our method to boost the performance. Hence, neither regularization nor data augmentation methods is a competing method to our attack. In

TABLE IV: *Summary of DeepSteal attack performance on large-scale dataset (i.e., CIFAR-100: 100 output class & GTSRB: 112x112 input dimension). Our proposed DeepSteal outperforms the baseline (i.e., architecture only) case across all three evaluation metrics even on large-scale image dataset. The improvements are shown in comparison to the baseline method.*

Dataset	*CIFAR-100*			*GTSRB*		
Method	Accuracy (%)	Fidelity (%)	Accuracy under Attack (%)	Accuracy (%)	Fidelity (%)	Accuracy Under Attack (%)
Baseline	32.81	33.97	42.7	98.67	98.01	68.28
DeepSteal	***59.8 (↑ 26)***	***64.11(↑ 31)***	***6.61***	***99.57***	***98.77***	***43.5(↓ 24)***
White-Box	69.7	100.0	0.0	99.05	100.0	4.32

TABLE V: *We evaluate DeepSteal attack against state-of-the-art techniques across three different domains as case studies. In each of the cases, only our attack performs on-par with the SOTA methods across all three evaluation metrics.*

Case Study	*Method*	*Objective*	*Model*	*Accuracy (%)*	*Fidelity (%)*	*Accuracy Under Attack (%)*
Regularization & Data Augmentation	Fully-Supervised [37]	Accuracy/Fidelity	WideResNet-28	86.51	87.37	-
	Rand-Augment [59]	Accuracy	WideResNet-28	87.4	-	-
	Auto-Augment [60]	Accuracy	WideResNet-8	87.7	-	-
	Mix-Match [11]	Accuracy/Fidelity	WideResNet-28	93.2	93.9	-
	DeepSteal (ours)	***Accuracy/Fidelity/Attack***	***WideResNet-28***	***90.35***	***91.82***	***0.16***
Model Extraction	Side Channel [27], [40], [42], [43]	Accuracy/Fidelity/Attack	ResNet-18	72.68	73.59	62.58
	DeepSteal (ours)	***Accuracy/Fidelity/Attack***	***ResNet-18***	***90.02***	***91.67***	***1.2***
Input Attack	Black-Box (Inception-V1) [74]	Adversarial Attack	ResNet-18	-	-	20.47
	White-Box (PGD/Trades) [65], [75]	Adversarial Attack	ResNet-18	-	-	0.0
	DeepSteal (ours)	***Adversarial Attack***	***ResNet-18***	**-**	**-**	***1.2***

addition, the mix-match recovery method requires significant modification to make it functionally equivalent only for a two-layer neural network, as discussed in [11].

Other existing side channel attacks [27], [39], [40], [42], [43] fall into a similar attack surface category as DeepSteal. Among them, [39] is only applicable to binary neural networks. On the contrary, our attack is a more general version of the attack applicable to any bit-width. Other side channel attacks [27], [40], [42], [43] focus on recovering the architecture and then training the model with limited data. To compare with them, we assume the attacker knows the exact model architecture, which is on top of those attacks. Then, our DeepSteal leverages the leaked weight bits to further improve the attack efficacy. From this point, our attack certainly outperforms such prior architecture only model extraction attacks with $\sim$18 % improvement in accuracy and $\sim$61 % improvement in degrading the accuracy under adversarial attack.

Finally, DeepSteal attack falls into a grey zone between black-box and white-box adversarial input attack. In Table V, we show DeepSteal can achieve 1.2 % accuracy under attack, which is more closer to a white-box attack performance (i.,e., 0 %). We observe a 17 % improvement in attack performance compared to a powerful black-box substitute model (e.g., Inception-V1) attack.

IX. Countermeasures for DeepSteal

A. Hardware-Assisted Protection for Secret Pages

One possible way to mitigate DeepSteal information leakage is to protect the secretive pages through support of trusted execution environments such as Intel SGX [76]. With Intel SGX, pages that belongs to a protected enclave is encrypted using processor-side memory encryption engine (MEE) before leaving the processor die. This means that the enclave pages stored in the main memory are encrypted by design and hence using this attack, the attacker cannot retrieve any information about the actual data. However, applications running on enclave have several times of overhead in runtime [77] compared to unsecure non-enclave version because of the nature of protection. Additionally, the maximum effective size of encrypted data size in memory is 96MB in Microsoft Windows [78], which creates non-trivial overhead for applications requiring larger memory than this (i.e., large DNN models), due to the necessarily complex mechanism of page-fault handling.

B. Software-based Protection of DNN Weights

Alternative to the hardware-assisted protection schemes, application developer can use software based encryption-decryption scheme for streaming applications. For example, using software base en-/decryption, the ML framework can perform on-demand weight decryption during computation and does not store the decrypted weight at any time. Since the unencrypted weights are not stored in memory, Hammer-Leak will unable to any secret on this. Note that while the runtime overhead of such software-based protection scheme needs evaluation, one block encryption (128-bit) using AES-NI instruction set (advanced instruction set designed specifically to accelerate AES encryption/decryption) still requires 8-cycle latency [79]. This can accumulate substantially and result in

potentially high run-time overhead, limiting the deployment of such implementation in DNN applications.

X. Conclusion

The training of deep neural networks requires heavy computational resources and sensitive private data. Thus, any potential breach in model privacy through leakage of sensitive model parameters may cost the service provided a heavy penalty. Our proposed DeepSteal attack reveals the serious threat of an effective model extraction attack. In particular, our weight bit extraction method HammerLeak showed it could recover a significant portion of the weight bits. Our substitute model training algorithm can leverage this information to effectively launch a strong adversarial input attack to the victim model. The success of DeepSteal attack demands more defense work in protecting the IP of DNN models.

References

[1] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[2] He *et al.*, "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification," in *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*, December 2015.

[3] W. Xiong, J. Droppo, X. Huang, F. Seide, M. Seltzer, A. Stolcke, D. Yu, and G. Zweig, "Achieving human parity in conversational speech recognition," *arXiv preprint arXiv:1610.05256*, 2016.

[4] V. Chandrasekaran, K. Chaudhuri, I. Giacomelli, S. Jha, and S. Yan, "Exploring connections between active learning and model extraction," in *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Aug. 2020, pp. 1309–1326.

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

[6] K. Murdock, D. Oswald, F. D. Garcia, J. Van Bulck, D. Gruss, and F. Piessens, "Plundervolt: Software-based fault injection attacks against intel sgx," in *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2020, pp. 1466–1482.

[7] A. Kwong, D. Genkin, D. Gruss, and Y. Yarom, "Rambleed: Reading bits in memory without accessing them," in *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2020, pp. 695–711.

[8] D. Gruss, C. Maurice, and S. Mangard, "Rowhammer. js: A remote software-induced fault attack in javascript," in *International conference on detection of intrusions and malware, and vulnerability assessment*. Springer, 2016, pp. 300–321.

[9] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu, "Flipping bits in memory without accessing them: An experimental study of dram disturbance errors," *ACM SIGARCH Computer Architecture News*, vol. 42, no. 3, pp. 361–372, 2014.

[10] Z. Kenjar, T. Frassetto, D. Gens, M. Franz, and A.-R. Sadeghi, "V0ltpwn: Attacking x86 processor integrity from software," in *29th {USENIX} Security Symposium ({USENIX} Security 20)*, 2020, pp. 1445–1461.

[11] 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)*, 2020, pp. 1345–1362.

[12] V. Chandrasekaran, K. Chaudhuri, I. Giacomelli, S. Jha, and S. Yan, "Exploring connections between active learning and model extraction," in *29th {USENIX} Security Symposium ({USENIX} Security 20)*, 2020, pp. 1309–1326.

[13] T. Orekondy, B. Schiele, and M. Fritz, "Knockoff nets: Stealing functionality of black-box models," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2019, pp. 4954–4963.

[14] A. Barbalau, A. Cosma, R. T. Ionescu, and M. Popescu, "Black-box ripper: Copying black-box models using generative evolutionary algorithms," *arXiv preprint arXiv:2010.11158*, 2020.

[15] G. K. Nayak, K. R. Mopuri, V. Shaj, V. B. Radhakrishnan, and A. Chakraborty, "Zero-shot knowledge distillation in deep networks," in *International Conference on Machine Learning*. PMLR, 2019, pp. 4743–4751.

[16] J. R. Correia-Silva, R. F. Berriel, C. Badue, A. F. de Souza, and T. Oliveira-Santos, "Copycat cnn: Stealing knowledge by persuading confession with random non-labeled data," in *2018 International Joint Conference on Neural Networks (IJCNN)*. IEEE, 2018, pp. 1–8.

[17] S. Pal, Y. Gupta, A. Shukla, A. Kanade, S. Shevade, and V. Ganapathy, "A framework for the extraction of deep neural networks by leveraging public data," *arXiv preprint arXiv:1905.09165*, 2019.

[18] N. Papernot, P. McDaniel, I. Goodfellow, S. Jha, Z. B. Celik, and A. Swami, "Practical black-box attacks against machine learning," in *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*. ACM, 2017, pp. 506–519.

[19] S. Milli, L. Schmidt, A. D. Dragan, and M. Hardt, "Model reconstruction from model explanations," in *Proceedings of the Conference on Fairness, Accountability, and Transparency*, 2019, pp. 1–9.

[20] D. Rolnick and K. Kording, "Reverse-engineering deep relu networks," in *International Conference on Machine Learning*. PMLR, 2020, pp. 8178–8187.

[21] 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.

[22] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, "Last-level cache side-channel attacks are practical," in *2015 IEEE symposium on security and privacy*. IEEE, 2015, pp. 605–622.

[23] F. Yao, G. Venkataramani, and M. Doroslovački, "Covert timing channels exploiting non-uniform memory access based architectures," in *Proceedings of the on Great Lakes Symposium on VLSI 2017*, 2017, pp. 155–160.

[24] M. H. I. Chowdhuryy, H. Liu, and F. Yao, "Branchspec: Information leakage attacks exploiting speculative branch instruction executions," in *2020 IEEE 38th International Conference on Computer Design (ICCD)*. IEEE, 2020, pp. 529–536.

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

[26] L. Batina, S. Bhasin, D. Jap, and S. Picek, "Csi neural network: Using side-channels to recover your artificial neural network information," *arXiv preprint arXiv:1810.09076*, 2018.

[27] 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.

[28] Y. Zhu, Y. Cheng, H. Zhou, and Y. Lu, "Hermes attack: Steal {DNN} models with lossless inference accuracy," in *{USENIX} Security Symposium*, 2021.

[29] Y. Yarom and K. Falkner, "Flush+ reload: A high resolution, low noise, l3 cache side-channel attack," in *{USENIX} Security Symposium*, 2014, pp. 719–732.

[30] D. Evtyushkin, R. Riley, N. C. Abu-Ghazaleh, ECE, and D. Ponomarev, "Branchscope: A new side-channel attack on directional branch predictor," *ACM SIGPLAN Notices*, vol. 53, no. 2, pp. 693–707, 2018.

[31] M. Yan, R. Sprabery, B. Gopireddy, C. Fletcher, R. Campbell, and J. Torrellas, "Attack directories, not caches: Side channel attacks in a non-inclusive world," in *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019, pp. 888–904.

[32] M. H. I. Chowdhuryy and F. Yao, "Leaking secrets through modern branch predictors in the speculative world," *IEEE Transactions on Computers (TC)*, 2021.

[33] A. S. Rakin, Z. He, and D. Fan, "Bit-flip attack: Crushing neural network with progressive bit search," in *The IEEE International Conference on Computer Vision (ICCV)*, October 2019.

[34] F. Yao, A. S. Rakin, and D. Fan, "Deephammer: Depleting the intelligence of deep neural networks through targeted chain of bit flips," in *USENIX Security Symposium*, 2020, pp. 1463–1480.

[35] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[36] S. Addepalli, G. K. Nayak, A. Chakraborty, and V. B. Radhakrishnan, "Degan: Data-enriching gan for retrieving representative samples from a trained classifier," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 34, no. 04, 2020, pp. 3130–3137.

[37] 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.

[38] Y. Zhang, R. Yasaei, H. Chen, Z. Li, and M. A. Al Faruque, "Stealing neural network structure through remote fpga side-channel analysis," in *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2021, pp. 225–225.

[39] 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.

[40] 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.

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

[42] 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)*. USENIX Association, Aug. 2020, pp. 2003–2020. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/yan

[43] Y. Xiang, Z. Chen, Z. Chen, Z. Fang, H. Hao, J. Chen, Y. Liu, Z. Wu, Q. Xuan, and X. Yang, "Open dnn box by power side-channel attack," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 67, no. 11, pp. 2717–2721, 2020.

[44] R. Callan, A. Zajić, and M. Prvulovic, "Fase: Finding amplitude-modulated side-channel emanations," in *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2015, pp. 592–603.

[45] Z. Zhang, S. Liang, F. Yao, and X. Gao, "Red alert for power leakage: Exploiting intel rapl-induced side channels," in *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security*, 2021, pp. 162–175.

[46] F. Yao, H. Fang, M. Doroslovački, and G. Venkataramani, "Cotsknight: Practical defense against cache timing channel attacks using cache monitoring and partitioning technologies," in *2019 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*. IEEE, 2019, pp. 121–130.

[47] H. Fang, S. S. Dayapule, F. Yao, M. Doroslovački, and G. Venkataramani, "Prodact: Prefetch-obfuscator to defend against cache timing channels," *International Journal of Parallel Programming*, vol. 47, no. 4, pp. 571–594, 2019.

[48] O. Mutlu and J. S. Kim, "Rowhammer: A retrospective," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 8, pp. 1555–1571, 2019.

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

[50] L. Cojocar, K. Razavi, C. Giuffrida, and H. Bos, "Exploiting correcting codes: On the effectiveness of ecc memory against rowhammer attacks," in *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019, pp. 55–71.

[51] M. Seaborn and T. Dullien, "Exploiting the dram rowhammer bug to gain kernel privileges," *Black Hat*, vol. 15, p. 71, 2015.

[52] Y. Jang, J. Lee, S. Lee, and T. Kim, "Sgx-bomb: Locking down the processor via rowhammer attack," in *Proceedings of the 2nd Workshop on System Software for Trusted Execution*, 2017, pp. 1–6.

[53] K. Cai, M. H. I. Chowdhuryy, Z. Zhang, and F. Yao, "Seeds of seed: Nmt-stroke: Diverting neural machine translation through hardware-based faults," 2021.

[54] K. Razavi, B. Gras, E. Bosman, B. Preneel, C. Giuffrida, and H. Bos, "Flip feng shui: Hammering a needle in the software stack," in *USENIX Security Symposium*. Austin, TX: USENIX Association, Aug. 2016, pp. 1–18. [Online]. Available: https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/razavi

[55] M. Ribeiro, K. Grolinger, and M. A. Capretz, "Mlaas: Machine learning as a service," in *2015 IEEE 14th International Conference on Machine Learning and Applications (ICMLA)*. IEEE, 2015, pp. 896–902.

[56] F. Yao, M. Doroslovacki, and G. Venkataramani, "Are coherence protocol states vulnerable to information leakage?" in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2018, pp. 168–179.

[57] R. K. Konoth, M. Oliverio, A. Tatar, D. Andriesse, H. Bos, C. Giuffrida, and K. Razavi, "Zebram: comprehensive and compatible software protection against rowhammer attacks," in *OSDI*, 2018, pp. 697–710.

[58] P.-Y. Chen, H. Zhang, Y. Sharma, J. Yi, and C.-J. Hsieh, "Zoo: Zeroth order optimization based black-box attacks to deep neural networks without training substitute models," in *Proceedings of the 10th ACM Workshop on Artificial Intelligence and Security*. ACM, 2017, pp. 15–26.

[59] E. D. Cubuk, B. Zoph, J. Shlens, and Q. V. Le, "Randaugment: Practical automated data augmentation with a reduced search space," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops*, 2020, pp. 702–703.

[60] E. D. Cubuk, B. Zoph, D. Mane, V. Vasudevan, and Q. V. Le, "Autoaugment: Learning augmentation strategies from data," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2019, pp. 113–123.

[61] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga *et al.*, "Pytorch: An imperative style, high-performance deep learning library," *Advances in neural information processing systems*, vol. 32, pp. 8026–8037, 2019.

[62] A. Tatar, C. Giuffrida, H. Bos, and K. Razavi, "Defeating software mitigations against rowhammer: a surgical precision hammer," in *International Symposium on Research in Attacks, Intrusions, and Defenses*. Springer, 2018, pp. 47–66.

[63] M. Gorman, *Understanding the Linux virtual memory manager*. Prentice Hall Upper Saddle River, 2004.

[64] 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.

[65] A. Madry, A. Makelov, L. Schmidt, D. Tsipras, and A. Vladu, "Towards deep learning models resistant to adversarial attacks," in *International Conference on Learning Representations*, 2018. [Online]. Available: https://openreview.net/forum?id=rJzIBfZAb

[66] A. Krizhevsky, V. Nair, and G. Hinton, "Cifar-10 (canadian institute for advanced research)," *URL http://www. cs. toronto. edu/kriz/cifar. html*, 2010.

[67] A. Krizhevsky and G. Hinton, "Learning multiple layers of features from tiny images," Citeseer, Tech. Rep., 2009.

[68] J. Stallkamp, M. Schlipsing, J. Salmen, and C. Igel, "Man vs. computer: Benchmarking machine learning algorithms for traffic sign recognition," *Neural Networks*, no. 0, pp. –, 2012. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0893608012000457

[69] S. Zagoruyko and N. Komodakis, "Wide residual networks," *arXiv preprint arXiv:1605.07146*, 2016.

[70] S. Xie, R. B. Girshick, P. Dollár, Z. Tu, and K. He, "Aggregated residual transformations for deep neural networks," *CoRR*, vol. abs/1611.05431, 2016. [Online]. Available: http://arxiv.org/abs/1611.05431

[71] D. Khudia, J. Huang, P. Basu, S. Deng, H. Liu, J. Park, and M. Smelyanskiy, "Fbgemm: Enabling high-performance low-precision deep learning inference," *arXiv preprint arXiv:2101.05615*, 2021.

[72] M. Dukhan, Y. Wu, and H. Lu, "QNNPACK: Open source library for optimized mobile deep learning." [Online]. Available: https://engineering.fb.com/2018/10/29/ml-applications/qnnpack/

[73] Y. Langsam, M. Augenstein, and A. M. Tenenbaum, *Data Structures using C and C++*. Prentice Hall New Jersey, 1996, vol. 2.

[74] W. Cui, X. Li, J. Huang, W. Wang, S. Wang, and J. Chen, "Substitute model generation for black-box adversarial attack based on knowledge distillation," in *2020 IEEE International Conference on Image Processing (ICIP)*. IEEE, 2020, pp. 648–652.

[75] H. Zhang, Y. Yu, J. Jiao, E. P. Xing, L. E. Ghaoui, and M. I. Jordan, "Theoretically principled trade-off between robustness and accuracy," in *International Conference on Machine Learning*, 2019.

[76] V. Costan and S. Devadas, "Intel sgx explained." *IACR Cryptol. ePrint Arch.*, vol. 2016, no. 86, pp. 1–118, 2016.

[77] M. Orenbach, P. Lifshits, M. Minkin, and M. Silberstein, "Eleos: Exitless os services for sgx enclaves," in *Proceedings of the Twelfth European Conference on Computer Systems*, 2017, pp. 238–253.

[78] "Enclave Memory Measurement Tool for Intel® Software Guard Extensions (Intel® SGX) Enclaves." [Online]. Available: https://software.intel.com/content/dam/develop/external/us/en/documents/enclave-measurement-tool-intel-sgx-737361.pdf

[79] R. Benadjila, O. Billet, S. Gueron, and M. J. Robshaw, "The intel aes instructions set and the sha-3 candidates," in *International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 2009, pp. 162–178.