# 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]
[1] School of Electrical, Computer, and Energy Engineering, Arizona State University, Tempe, AZ
[2] Department of Electrical and Computer Engineering, University of Central Florida, Orlando, FL
$*$ *Co-first Authors with Equal Contributions*

***Abstract*—Recent advancements in Deep Neural Networks (DNNs) have enabled widespread deployment in multiple security-sensitive domains. The need for resource-intensive training and the use of valuable domain-specific training data have made these models the top intellectual property (IP) for model owners. One of the major threats to DNN privacy is model extraction attacks where adversaries attempt to steal sensitive information in DNN models. In this work, we propose an advanced model extraction framework *DeepSteal* that steals DNN weights *remotely* for the first time with the aid of a 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 fault technique as the information leakage vector. *HammerLeak* leverages several novel system-level techniques tailored 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 the proposed model extraction framework 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

Recent developments in deep learning technologies have made machine learning an integral part of our daily life. This widespread application of Deep Neural Networks (DNNs) includes but 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 shown that adversaries can leverage side channel attacks to gain sensitive information in computing systems [21]–[24]. Particularly, several works have demonstrated successful information leakage manipulation in DNN models by leveraging microarchitecture attacks [25], power/electromagnetic side channels [26], [27] and bus snooping attacks [28], [29]. Hardware-based attacks can be extremely dangerous as they allow adversaries to *directly gain internal knowledge about the victim DNN models*. However, existing hardware-based DNN attacks either only extract high-level model structures (e.g., model architectures) or require physical access to the target machines to gain fine-grained model information, which is not applicable to remote victims (e.g., in the cloud). Recent work reveals that rowhammer attacks can be harnessed as a side channel exploit in a remote setting [7]. However, it remains uncertain whether accurate information about model weights (i.e., the core information of DNN models) can be effectively exfiltrated via rowhammer-based side channels. 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 bolstering adversarial attacks [18], [30]. In this paper, we focus on answering the following question: *Is it possible for an adversary to perform model extractions through remotely stealing weight parameters using rowhammer-based side channels?*

While obtaining model weights can be useful intuitively, there are several major challenges from the attacker's perspective to practically capture and effectively utilize such information. *First*, although fine-grained secret leakage has been widely shown to be plausible in many non-ML applications (e.g., through microarchitecture attacks [22], [31]–[35]), such attacks fail 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 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 [36], [37] have shown that variations on only tens out of millions of weight parameters will completely malfunction a DNN model. In this case, whether partial information of 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 rowhammer-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 first aforementioned challenge, *in the first stage*, we develop a novel rowhammer-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 attack 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 from its neighboring cells [7], [37]. While such a phenomenon was first used in the prior work [7] to successfully steal crypto keys, we note that such a technique is ineffective in stealing secrets in bulk and also cannot be applied in the context of DNNs. Therefore, we 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, the weight search space of a victim model still remains high. For instance, even after recovering 90% of the bits in a large model like VGG-11 [38] (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). To address the 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 that achieves similar accuracy as the victim model with high fidelity. Moreover, the trained substitute model could help mount strong adversarial input attacks on the victim model. We summarize the major contributions of our work as follows:

1) In this work, we investigate an advanced model extraction attack with the exploitation of a remote side channel that *for the first time* can exfiltrate fine-grained information from DNN model weights.
2) We develop *HammerLeak*, a multi-round rowhammer-based information leakage technique that can exfiltrate secretive data from computing systems in bulk with high accuracy. To make the rowhammer side channel applicable and efficient for attacking DNNs, HammerLeak integrates a set of novel system-level techniques such as *the use of more flexible rowhammer memory layouts*, *effective DNN inference anchoring* and *batched victim page releasing*.
3) We propose a novel training algorithm to build a substitute model using the leaked weight bits as constraints. It first conducts a data filtering process of the leaked bits to construct a profile consisting of projected searching space for each weight. 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 attack in real systems and demonstrate its efficacy on four popular DNN architectures (e.g., ResNet-18/34, Wide-ResNet-28, and VGG-11). We evaluate our attack on three standard image classification datasets (e.g., CIFAR-10, CIFAR-100, and GTSRB). For example, the extracted substitute model has successfully achieved more than *90%* test accuracy on deep residual networks for the CIFAR-10 dataset.
5) Finally, we demonstrate the superior performance of leveraging the recovered model weight bits (i.e., MSBs) to build more powerful substitute model (as compared to the use of model architecture information only) for adversarial input attacks. With our DeepSteal, it can generate effective adversarial examples with similar attack efficacy as a white-box attack (e.g., 0.05%).

## II. Background and Related Works

### 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 (e.g., model architecture, weights and biases). An ideal model extraction attack would 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 $M_\theta$ with parameters $\theta$. In this work, we designate this model $M_\theta$ 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 parameters) stealing is practically challenging if not impossible [11].

**Algorithm-based Model Extraction.** Prior works [11], [12], [39] have defined several potential 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], [40] leverage 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], [41] aim to achieve high model accuracy or 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 requirement of excessive 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], [40] | Functionally Equivalent |
| Active Learning/Learning | [11]–[18], [41] | Task Accuracy/Fidelity |
| Side channel & Learning | [25]–[29], [42]–[45] | 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], [46]–[50]. Recent works have demonstrated that such attack vectors can also be applied to exfiltrate sensitive DNN information [25]–[28], [42]–[44]. 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. Typically, the goal of side channel attack is to produce a functionally equivalent model or achieve high fidelity on a dataset. To achieve this, attacker often supplements 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 [28]. Note that these side channel attacks primarily recover model architecture or hyperparameters. However, to date, only a limited number of studies have explored the extraction of model parameters (i.e., weights) in DNN models. Recent studies on physical side channels have exhibited successful exfiltration of model parameter information, including EM side channels [27] and PCI-e bus snooping [29]. These works assume the attacker has physical access to the target machines to enable hardware-based probing or snooping, which may not be practical for remote exploitation of platforms such as cloud services. The goal of this work is to investigate the possibility of exploiting rowhammer-based side channels to perform remote stealing of model weights, and explore ways to generate a 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.

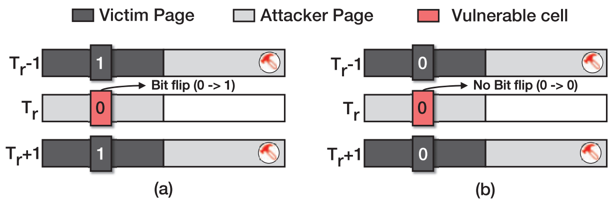


Fig. 1: Data dependency for inducing a rowhammer fault. Here, based on the presence of bit flip in the *attacker-controlled* vulnerable bit in the target row ($T_r$), data from adjacent row from victim program can be inferred.

## B. Rowhammer Attacks

Rowhammer is a software-induced fault attack that exploits DRAM disturbance errors via user-space applications [9]. Specifically, it has been shown that accesses (i.e., activations) to certain DRAM rows can introduce electrical disturbance to the DRAM cells in the neighboring rows, which accelerates the leakage of their charges in the capacitors [9], [51]. An attacker can intentionally activate particular DRAM rows (whose data belongs to the attacker) frequently enough (i.e., hammering) to eventually cause bit flips in a victim's address space. Such attacks have been successfully demonstrated on commercial-off-the-shelf DRAM modules even with the presence of ECC features [52], [53]. There are mainly three hammering techniques proposed in the literature: a) double-sided hammering [8], [9], [51], [53], [54]: where two aggressor rows are frequently activated to induce fault in the middle row b) single-sided hammering [54]: where one adjacent row to the target row and another random row are activated repeatedly; and c) one-sided hammering [52]: where one periodically-accessed row causes repetitive row activations under the *close-page* DRAM policy. Among all techniques, double-sided hammering is the most effective in inducing DRAM faults since it introduces the strongest disturbance.

There is a large body of works demonstrating many variants of rowhammer attacks. Most of them focus on tampering the integrity of systems, including privilege escalations [54], system denial of service [55] and more recently faulting DNN model parameters [37], [56]. Recently, RAMBleed [7] reveals that the rowhammer fault characteristic can be leveraged to carry out *information leakage* attacks that directly infer victim secrets in memory. 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 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 cells 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 places his own page in the middle row that has the vulnerable cell, and manages to trigger the placement of two copies of a victim's page in the corresponding aggressor rows. By observing whether a bit flip occurs in attacker's pages after hammering, the adversary can infer the secretive bit (i.e., RSA keys). We note that exploitation of rowhammer in the domain of information leakage 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. We assume that the deep learning system is deployed in 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 [57]. The attacker can control a user-space unprivileged process that runs on the machine where the victim DNN service is deployed. Our proposed framework manifest as a *semi-black box* attack where the adversary does not have any prior knowledge of the model parameters. 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., via caches [5], memory bus [28] and EM [26]).

In this work, we leverage the rowhammer fault attack vector that commonly exists in today's DRAM-based memory systems as the side channel [7]. Specifically, the attacker takes advantage of the fact that bit flip in *vulnerable DRAM cells* only occurs when the column-wise bit striping pattern exists in double-sided rowhammering. By leveraging such data dependency, the attacker can infer bits in the aggressor rows by observing if a bit flip occurs in *his own address space*. In other words, the attacker does not directly tamper with the victim's memory (as shown in most traditional rowhammer attacks). 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 [58]. We assume that proper confinement mechanism is put in place to disallow direct access to data across processes. We further assume that the operating system and the hypervisor are benign, and appropriate kernel-space protection mechanisms are deployed to avoid direct tampering of kernel structures [59].

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. Meanwhile, similar to recent related works [60], [61], we assume the attacker has access to a publicly available portion (e.g., $\leq$ 10%) of the labeled training dataset.

TABLE II: *List of information accessible to the attacker for substitute model training.*

| 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 ($\leq$ 10%) | ✓ |

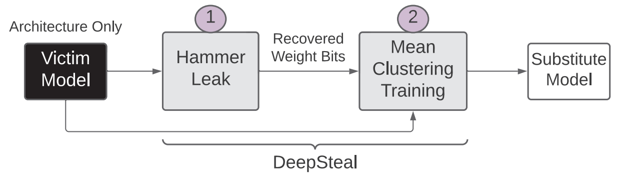


Fig. 2: Overview of the *DeepSteal* attack framework. *Stage-1:* exfiltrating DNN partial weight bits efficiently through exploiting memory fault vulnerabilities (HammerLeak). *Stage-2:* With the recovered bits, training a substitute model using *Mean Clustering* weight penalty.

## IV. Overview of DeepSteal

In this work, we propose an advanced model extraction attack framework through efficient weight bits stealing in memories. An overview of our proposed attack framework, *DeepSteal*, is shown in Figure 2. It has two key components: i) an efficient rowhammer-based weight-stealing side channel module *HammerLeak*, and ii) a substitute model training mechanism with novel *Mean Clustering* loss penalty. At stage-1 in Figure 2, we mount the HammerLeak attack on inference infrastructure (i.e., a remote machine running the target DNN inference service) to recover partial weight bits. We continue the HammerLeak for many rounds until the desired portion of weight bits is recovered. Once HammerLeak completes, at stage-2, our goal is to use the leaked weight bit information and generate a substitute prototype of the victim model. To achieve this, we propose a novel neural network training algorithm that constrains the trained substitute model weight parameters to be as close as possible to the recovered partial weight info, as well as minimize the accuracy loss. The learned substitute model will pose the following properties: i) having comparable test accuracy as the victim model; ii) exhibiting high fidelity and iii) can be used to generate extremely strong

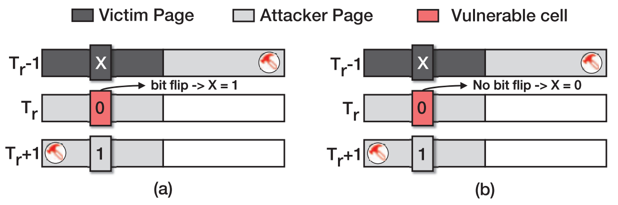


Fig. 3: HammerLeak attack leaking victim secrets using single victim page. (a) Bit flip observed when victim's bit is `1`, (b) Bit flip not observed when victim's bit is `0`.

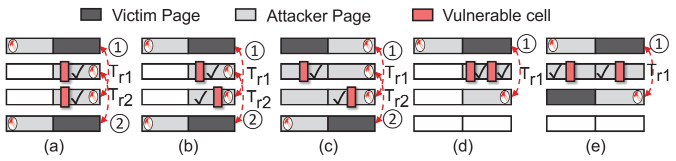


Fig. 4: Efficient utilization of multiple target rows holding flippable bits. HammerLeak can utilize all $V_c$ in (a) adjacent target rows ($T_{r1}$, $T_{r2}$) with the same $V_c$ offset, (b) adjacent $T_{r1}$, $T_{r2}$ with different $V_c$ offsets, (c) adjacent $T_{r1}$, $T_{r2}$ with different $V_c$ offsets (alternate page), (d) one $T_r$ with different $V_c$ offsets (same page) and (e) same $T_r$ with different $V_c$ offsets (different page).

adversarial input samples to the victim model. We describe the details of our DeepSteal framework in the following sections.

## V. HammerLeak: Efficient Data Stealing in Memories

In this section, we present *HammerLeak*, an efficient rowhammer-based information leakage attack which can steal victim's secretive data in bulk. HammerLeak is a multi-round attack framework. In each round, it manages to relocate victim's model weight pages to leakable DRAM locations and perform rowhammer-based side channel to steal weight bits. Such operations are iterated over multiple rounds until sufficient bits are leaked for model extractions.

### A. Leveraging Rowhammer in HammerLeak

Although the rowhammer attack in RAMBleed [7] demonstrates information stealing from the aggressor rows adjacent to a vulnerable cell, it requires the same secret page to be present in both aggressor rows. While it is possible that certain applications may launch multiple threads with each allocating memories to store a duplicated copy of shared pages (e.g., as in the case of OpenSSH), we find that such duplication does not commonly exist in many applications (e.g., ML framework in PyTorch [62]). Therefore, this type of manifestation does not work in a more general setting. HammerLeak augments rowhammer by exploiting the same observation of data dependency, but with added capability of leaking victim bits by utilizing *only one copy of victim page*. This makes the rowhammer-based side channel potentially applicable to any victim application. As shown in Figure 3, instead of requiring two duplicated copies of the victim page, the attacker substitutes one victim page with his own page while still being able to leak secret bits from the victim page. Specifically, the attacker first places the victim page to one of the adjacent aggressor rows (e.g., top aggressor row, $T_r-1$ shown in example Figure 3) and then places attacker's own page in the other aggressor row (e.g., bottom aggressor row, $T_r+1$). Note that the target row ($T_r$) contains the vulnerable DRAM cell ($V_c$), which can have `0`$\rightarrow$`1` flip as an illustration. The content of the two attacker rows is controlled such that it creates a bit layout of `X-0-1` with the aggressor row containing the victim page, where `X` is the secretive victim bit, the middle bit (`0`) is $V_c$ and the last bit (`1`) is the other attacker-controllable bit. When the two aggressor rows are frequently accessed, a bit flip will be observed for $V_c$ if `X` is `1` as shown in Figure 3a (thus creating a `1-0-1` pattern), otherwise there will be no bit flip in $V_c$ (Figure 3b). The same technique can be applied with the `X-1-0` pattern if flip direction of $V_c$ is `1`$\rightarrow$`0`.

**Efficient Utilization of Vulnerable Rows.** Typically, vulnerable DRAM cells are uniformly distributed across the DRAM DIMM [63]. As a result, it is possible that vulnerable cells are present in pages at adjacent rows as well as different pages in the same row. Carefully designing the rowhammer memory layout to maximize the utilization of vulnerable bits is critical for efficient bit leakage. Figure 4 enumerates the possible cases for multiple vulnerable cell locations. In case of multiple vulnerable cells in adjacent rows (i.e., Figure 4a, Figure 4b and Figure 4c), we can setup the memory layout so that HammerLeak first leaks secret from $T_{r1}$ (①) by utilizing $T_{r2}$ as an aggressor, then leaks from $T_{r2}$ by using $T_{r1}$ as an aggressor (②). When more than one $V_c$ exist on the same page (Figure 4d), we can leak multiple secret bits at the same time by setting proper memory layout in $T_r$ and the attacker-controlled aggressor row. Finally, when different $V_c$ are in different pages of the same $T_r$, HammerLeak places two victim pages, one to each aggressor row as shown in Figure 4e. Such a configuration allows the attacker to leak bits from both pages in one iteration of hammering. HammerLeak judiciously setup the memory layout based on the bit flip profile to maximize bit leakage in each round. Note that while this demonstration is considering a system with a *single-channel* memory with two pages per physical row, it remains valid for *multi-channel* memory systems.

### B. Bulk Secret Leakage using HammerLeak

To leak bits with rowhammer as discussed in Section V, victim pages containing secrets must be placed in a row adjacent to a $V_c$ to be leakable (i.e., memory massaging [37], [52]). Memory massaging of victim pages is challenging in HammerLeak due to several reasons: i) Generally, the target victim pages are anonymous pages and their allocations are directly managed by the operating system. Tweaking system software to relocate these pages as desired itself is a major challenge in rowhammer; ii) Typically only one (or a few) bit can be leaked from one victim page under attack in one round. In order to steal sufficient amount of bits, it is essential for HammerLeak to perform multiple rounds of attack, where

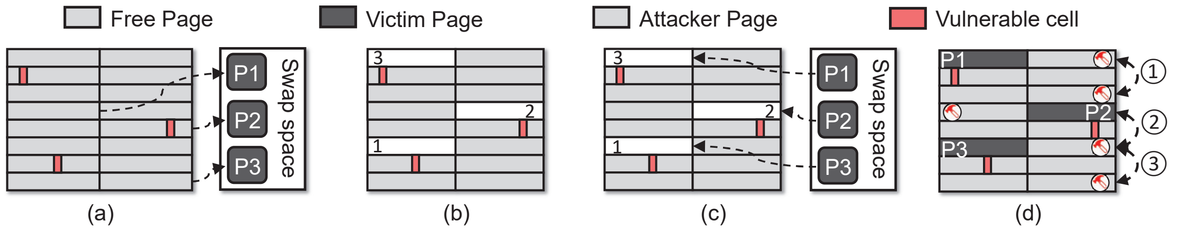


Fig. 5: Overview of the operations of HammerLeak.

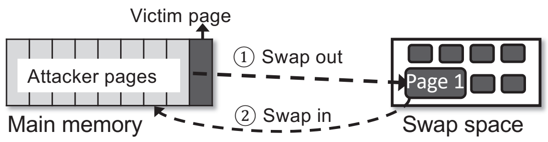


Fig. 6: Memory eviction technique occupying large memory space by attacker, forcing victim data to *swap out* (①). When victim accesses the data, its *swap in* (②) to a different location.

memory massaging is needed at the beginning of each round in order to leak *new bits*. Note that the prior memory massaging technique leveraging page deduplication (e.g., [64]) is not applicable as the attacker is not aware of the victim's page content. Furthermore, page cache evictions as proposed in [52] do not work well for relocating anonymous pages. We propose a four-step HammerLeak framework to augment a practical rowhammer-based side channel tackling these challenges.

**Step 1: Anonymous Page Swapping.** In the first step, the attacker aims to evict the victim's pages from main memory to swap space so that they later get relocated by the operating system at the next time these pages are accessed by the victim. During this step, the attacker first allocates a large chunk of memory using `mmap` with `MAP_POPULATE` flag. This triggers the OS to evict other data (including victim's pages) from the main memory to the swap space. At the end of this step (Figure 5a), the attacker occupies most of the physical memory space with victim pages stored in swap space. This procedure is illustrated in Figure 6 where the victim page (*Page 1*) is swapped out of main memory to the swap space (①). Once the victim accesses this page again, the OS *swaps in* Page 1 to the main memory at a new location (②).

**Step 2: Bit Flip-aware Page Release.** In this step, the attacker process systematically releases his own pages during victim's execution to enforce desired relocations of victim's pages. By obtaining the knowledge of virtual to physical page frame mapping of attacker's process [7], [52], the attacker first creates a list of potential pages for the victim to occupy as *aggressors* during the rowhammer-based leakage attack (i.e., pages that are adjacent to a $T_r$ holding one or more $V_c$). At each round of HammerLeak, the attacker chooses a predetermined number of pages from the list. Finally, the attacker releases the selected pages (i.e., by calling `munmap`) as illustrated in Figure 5b.

**Step 3: Deterministic Victim Relocation.** In this step, the attacker manages to place victim pages in predetermined locations to create an appropriate memory layout for rowhammer. Crucially, this also ensures that the victim page location is known to the attacker so that the attacker can correlate leaked bits with exact data in the victim domain. We exploit the per-core page-frame cache structure (i.e., `per-cpu pageset` [65]) to manipulate the OS page allocation, which allows the attacker to control where the victim pages are relocated. Specifically, `per-core pageset` is a *last-in-first-out* (LIFO) structure maintained by the Linux kernel 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 core to obtain a free memory location. We exploit the LIFO policy to deterministically place victim pages into the desired memory locations by running the attacker process on the same core. In particular, based on the order of pages released by the attacker during *Step 2*, the relocation of victim pages can be performed with extremely high accuracy [37] as illustrated in Figure 5c. Here `P1` (page 1), `P2` and `P3` are allocated for the victim in 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 Step 3, the victim pages are placed in the appropriate locations. The attacker mounts our rowhammer-based information leakage attack (as discussed in Section V-A) to steal victim's data. Based on the flip direction of a specific $V_c$ (i.e., flip in either `0`$\rightarrow$`1` or `1`$\rightarrow$`0` direction), we preset the $V_c$ and adjacent attacker controlled aggressor row bit respectively to `0-1` (for `0`$\rightarrow$`1`) or `1-0` (for `1`$\rightarrow$`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 recovers secret bits from all of the selected $T_r$ iteratively to maximize the data leakage in one round of HammerLeak.

## C. Batched Victim Page Massaging

While the one-time *bit flip-aware page release* during one round of HammerLeak is sufficient for victim programs with small memory footprints, this is not the case for applications with large memory allocations since the per-cpu pageset has a fixed size. For applications that have a large working set (i.e., larger than 2MB), deterministic victim page relocation cannot be guaranteed if all targeted pages in the victim in one round are released at once during Step 2. This is because if the required number of pages exceeds the size of pageset, the pageset will be overflown and the system will start to release some of the free pages to the global memory pool. At that point, the attacker can no longer leverage the LIFO policy of `pageset` to perform deterministic page relocation.

We propose *batched victim page massaging* to tackle this challenge. Specifically, HammerLeak periodically releases a small number of pages at specific points (i.e., anchor points) 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 non-leakable locations. In order to determine *when* to release leakable pages, the attacker needs to monitor victim's activities that lead to secretive page access. We built a cache side channel based tracking tool to set up function *anchor point* for this monitoring purpose. Specifically, we assume that the attacker and the victim application share the same library (which results in shared read-only physical pages [58]). For applications using open-source libraries (common for many crypto algorithm implementations and ML applications), the attacker analyzes and determines the execution points that access the targeted secret (e.g., DNN model weights) through source/binary code inspection (e.g., specific functions performing computation using secretive data). The attacker then launches the Flush+Reload attack [31] at certain *anchor points* in those functions to synchronize with the victim's execution. Through inspection of victim's execution flow, the attacker can determine the victim's memory access pattern (e.g., whether there are non-secretive page accesses between a chunk for secret page access) and derive the number of non-secretive pages the victim allocates after each anchor point, which guides the page releasing in batch.

Putting everything together, *prior to the attack*, the attacker determines anchor points 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 access to the anchor points. Once the victim access is detected, the attacker releases $P_b + P_s + P_i = P$ physical pages in reverse order of victim's page accesses (i.e., LIFO access). Out of these pages, only $P_s$ pages need to be in the adjacent rows to $V_c$ for information leakage. If $P_s$ is larger than the size of per-core pageset, the attacker will choose a different anchor point, which effectively divides the victim secret page access into additional smaller chunks. By chaining several page batches for memory massaging, HammerLeak can eventually steal bits spawning *hundreds to thousands of pages* in each round, enabling bit leakage in bulk.

## VI. Substitute model training with Mean Clustering

At Stage-2 of DeepSteal, we leverage the bit information leaked by HammerLeak to learn a substitute prototype of the victim DNN model. To fully leverage those leaked partial bit-wise data, we propose a novel substitute model training algorithm to reconstruct a neural network model, targeting high accuracy and high fidelity. Moreover, this learned substitute model will help the attacker generate highly effective adversarial input samples to fool the victim model successfully.

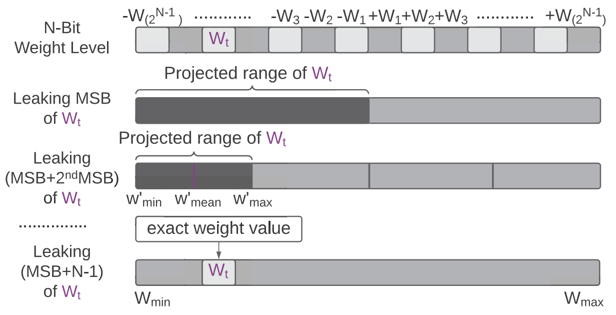


Fig. 7: *First row:* N-Bit quantized weight level; *Second row:* Once the MSB of weight $W_t$ in the victim model is leaked, we can narrow down the projected range of $W_t$ in the substitute model; *Last row:* Leaking all the bits can track down the exact value of $W_t$ for the substitute model training.

### 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 Figure 7, for each weight parameter, 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, the attacker must filter and reorganize the leaked bits in a sequence from MSB to LSB. Therefore, 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 a projected range, as described in Figure 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 weights 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, $\lambda$ is a hyper-parameter that controls the strength of the loss penalty and f($\cdot$) 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 purposed 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., $\lambda$=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, in the last few iterations (e.g., 40), the model will be fine-tuned ($\lambda = 0$, no clipping & low learning rate) to generate the final substitute model.

**Algorithm 1** *Substitute Model Training with Mean Clustering*

```
1:  procedure TRAIN SUBSTITUTE MODEL ($M_\theta$,$\hat{M}_\theta$,$\hat{\theta}^L_{l=1}$)
2:      Takes victim model architecture $\hat{M}_\theta$ as input
3:      Takes the leaked parameter list $\hat{\theta}^L_{l=1}$ as input.
4:      Randomly Initialize Substitute model $M_\theta$.
5:      Perform step-1, data filtering of the leaked bits.
6:      for Each layer $l = 1, \ldots, L$ do
7:          Compute Initial $\boldsymbol{W}^l_{min}$ & $\boldsymbol{W}^l_{max}$ using $\hat{\theta}_l$.
8:          Estimate $\boldsymbol{W}^l_{mean}$ = ($\boldsymbol{W}^l_{min}$+ $\boldsymbol{W}^l_{max}$)/2.
9:      end for
10:
11:     Re-Initialize model $M_\theta$ using following rules:
12:     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).
13:     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 $\lambda$ suitably.
14:     Weight Set-3 (0 bit recovered): Random Initialization
        & set $\lambda$ in Equation (1) to zero.
15:
16:     for each training iteration do
17:         for each training batch $(\boldsymbol{x}, \boldsymbol{y})$ do
18:             Compute Loss using Equation (1).
19:             Perform a gradient descent step to update $\theta$.
20:             Update $\boldsymbol{W}^l_{min}$, $\boldsymbol{W}^l_{max}$ & $\boldsymbol{W}^l_{mean}$.
21:         end for
22:         Clip $\{\boldsymbol{W}^l\}^L_{l=1}$ within the projected range.
23:     end for
24:     Return: Trained Substitute Model $M_\theta$.
25: 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 labels 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 the substitute model being correctly classified by the victim model. It indicates the transferability of the adversarial examples as explained in prior [66]. 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) [67] attack to generate adversarial samples on the substitute model. The PGD attack uses $L_\infty$ 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 AVX-2 instruction set support. We collect bit flip profile of the memory modules (i.e., templating) used in target system to identify potentially vulnerable locations in DRAM. Note that memory templating is considered as a standard process for rowhammer. We leverage existing techniques as described in [7], [52], [53], [64]. The system is configured with memory subsystem with 4GB DDR3 DIMMs in either single- or dual-channel settings. 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 observed in prior work showing multiple DRAM modules with more than 98% of all rows being vulnerable [63], our system has moderate level of vulnerability in rowhammer-induced bit flips. Finally, we profile the vulnerable DRAM cells and empirically categorize the $V_c$ into two classes based on flip repeatability: *Strongly-leakable* cells and *Weakly-leakable* cells. Our HammerLeak only leverages *Strongly-leakable* cells for the side channel to maximize the bit stealing accuracy.

## C. Dataset and Architecture

A detailed description of the model architecture and dataset is provided in the Appendix (XI-F).

## VIII. Evaluation

## A. DNN Weight Recovery using HammerLeak

We instantiate our HammerLeak attacks against DNN models running in the popular **PyTorch** framework. Depending on the hardware instruction set supported on the host system, PyTorch performs extensive optimization of core computational kernel for DNN. Particularly, PyTorch is directly compatible with FBGEMM [68] backend for x86 platform and QNNPACK [69] backend for ARM platform. These are used to accelerate matrix computations with platform specific instruction-set (i.e., AVX-2 or AVX-512). There are multiple platform- and hardware-specific optimization for each specific type of DNN model following different execution paths. For the rest of this discussion, we use 8-bit quantized DNN models running inference using the FBGEMM backend. We use PyTorch `v1.7.1-rc3` with FBGEMM commit `1d71039` running as the platform of our investigation. Note that a similar inference execution flow can be obtained for other configurations and models as well.

**In-Memory Organization of DNN Weights.** In order to use vector instructions to accelerate DNN inference, instruction

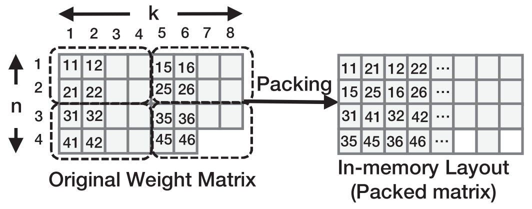


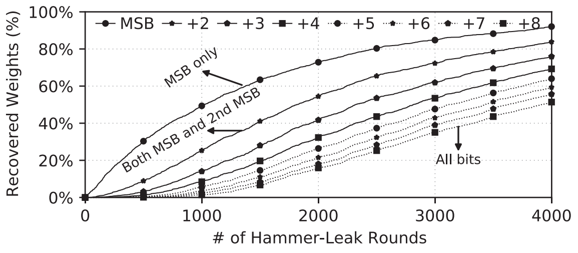
Fig. 8: Illustration of PyTorch weight packing (example using $4 \times 2$ size chunks) and organization of weights in memory. Each chunk is represented by dashed block and the small white blocks represents uninitialized location (no weight in this location).

operands used in the computation need to be arranged in a *packed layout* (i.e., a reorganization of matrix data into a data array optimized for sequential access by kernels [68]). PyTorch creates an optimized packed layout for both operands of these computations (i.e., weights and inputs). Since DNN model weights remain unchanged throughout an inference operation, PyTorch creates a packed copy of DNN weights of each linear and convolutional layer during model initialization and uses that pre-packed structure (stored in main memory) for all inference operations. The weight 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` symbol to represent weight matrices internally). FBGEMM divides the weights of each layer into smaller chunks, and then each chunk is stored sequentially in memory, which is useful for both accelerating computation performance and optimizing cache accesses. For AVX-2 enabled systems, each of these smaller chunks has a size of $512 \times 8$, indicating all weights in a specific layer are divided into several such weight chunks. Weights in one chunk are laid sequentially into memory using the *column-major* format, which is different from the regular memory *row-major* format [70]. Within each layer, the matrix computation kernel performs vector multiplication and accumulation to produce the output of that layer. Figure 8 illustrates the packing and memory layout of the stored weights. Given a specific memory byte in the in-memory layout, the actual weight location in original weight matrix can be determined with the knowledge of packed layout. Note that model weights are typically loaded only once to the main memory and prepared for inference (via packing) of streaming input data. Every single inference operation will require a complete traversal of these model weight pages. As such, weight pages are not de-allocated after each inference. The packed weight pages are kept in memory as long as the inference server is active.

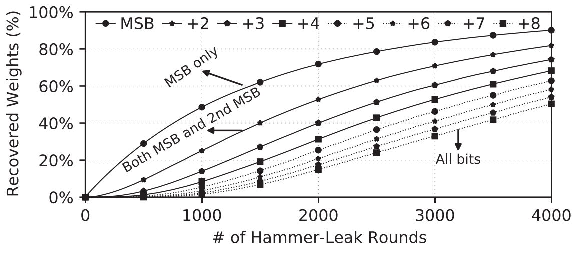
**Mounting HammerLeak on PyTorch.** To mount HammerLeak in PyTorch, we use batched victim page massaging (as described in Section V-C) with four anchor points. We first use the `forward(<input>).toTensor()` method of DNN model as the first anchor point to monitor the beginning of an inference. This anchor will trigger the monitoring of subsequent anchors. During the inference operation, PyTorch uses the `apply_impl` function (corresponding to the 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 allows HammerLeak to distinguish between different layer computations and determine which layer is currently being executed. The `apply_impl` passes pointers 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 has sufficient pages released for all the ML pages accessed in that chunk computation to occupy. Note that for each round, HammerLeak only needs to relocate the victim model's weight pages to the target DRAM locations once. While the model weight pages are placed in the aggressor rows, HammerLeak does not require PyTorch's inference to be performed to enable double-sided rowhammer for the leakage of certain model weight bit. As shown in Figure 3, HammerLeak achieves the memory layout where the attacker controls at least a page (or half a page in the case of dual-channel setting) in each aggressor row. Subsequently, the attacker launches the rowhammer-based side channel by accessing its own pages in the two aggressor rows alternatively.

### *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 about 90% of the MSB bits for model weights 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 MSB bits, the recovery rate for additional weight bits is also very high, with 55%-63% weights across all layers have the complete weight recovered. This shows the high efficiency of the proposed HammerLeak attack. Figure 10 illustrates the distribution of weight percentages that have at least MSB bit recovered for all 21 layers. Depending on the attacker's goal (in our case, high percentage of weights with MSB exfiltrated), HammerLeak can be completed sooner than 4000 rounds. We observe most



(a) Layer 1



(b) Another random layer (Layer 5 under illustration)

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

of the layers have half of the weights with MSB bit recovered within 1000 rounds.

We further investigate the time spent during each of the steps of a HammerLeak round to determine the system attack cost. We empirically observe that memory exhaustion (Step 1) and bit flip-aware page release (Step 2) requires 12 seconds and 21 seconds respectively. The most time-consuming operation (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 around 3K rows on average, which takes on average about 200 seconds. Note that only Step 2 and Step 3 have to be done simultaneously with the inference operation, the bulk of the operation (Step 4) can be done asynchronously by the attacker in each round as the weight pages are already placed as desired. Due to the importance of MSB bits, we choose to release the physical pages with at least one MSB offset leakable (i.e., MSB prioritization scheme). This way, we substantially improve the attack efficiency while leaking most of the influential bits. We experimentally found that using MSB prioritization, 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 compare with the architecture only case (i.e., 0-bit information leaked). For the

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+$2^{nd}$ MSB & so on). ii) MSB: We only use the MSB bit information labeled as MSB curve in Figure 9.*

| *Method* | | | *ResNet-18* | | | | *ResNet-34* | | | | *VGG-11* | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| # of HammerLeak Rounds | Method | Case | Time (days) | Accuracy (%) | Fidelity (%) | Accuracy under Attack (%) | Time (days) | Accuracy (%) | Fidelity (%) | Accuracy Under Attack (%) | Time (days) | Accuracy (%) | Fidelity (%) | Accuracy Under Attack (%) |
| Baseline | Arch. Only | - | - | 73.18 | 74.29 | 61.33 | - | 72.22 | 72.85 | 62.69 | - | 70.76 | 72.06 | 61.19 |
| 1500 | DeepSteal | All Bits | 4.5 | 74.33 | 75.38 | 53.64 | 7.6 | 74.43 | 75.2 | 55.99 | 3.9 | 72.3 | 73.34 | 62.24 |
| | | MSB | 3.9 | 76.61 | 77.56 | 50.4 | 6.5 | 76.77 | 77.53 | 53.47 | 3.4 | 72.67 | 73.89 | 58.19 |
| 3000 | DeepSteal | All Bits | 8.9 | 86.32 | 87.86 | 5.24 | 15.3 | 85.62 | 86.72 | 3.93 | 7.8 | 81.03 | 82.88 | 36.45 |
| | | MSB | 7.8 | 86.93 | 88.51 | 8.13 | 12.9 | 87.19 | 88.39 | 4.61 | 6.7 | 80.15 | 81.52 | 26.85 |
| 4000 | **DeepSteal** | All Bits | ***11.9*** | ***89.05*** | ***90.74*** | ***1.94*** | ***20.4*** | ***88.17*** | ***89.27*** | ***1.44*** | ***10.4*** | ***84.59*** | ***86.24*** | ***16.87*** |
| | | MSB | ***10.4*** | ***89.59*** | ***91.6*** | ***1.61*** | ***17.4*** | ***90.16*** | ***91.8*** | ***1.03*** | ***8.9*** | ***81.56*** | ***83.33*** | ***18.55*** |
| Best-Case | White-box | - | - | 93.16 | 100.0 | 0.0 | - | 93.11 | 100.0 | 0.0 | - | 89.96 | 100.0 | 4.63 |

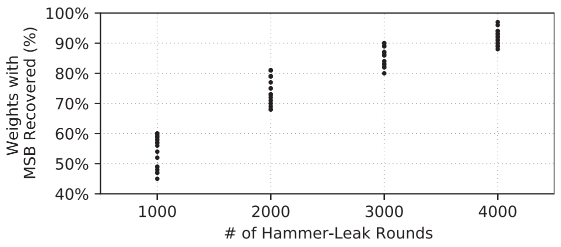


Fig. 10: Distribution of weights with `MSB` recovered across 21 individual layers of ResNet-18.

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 all information (i.e., weights, biases and architecture) of the victim model. In summary, with more recovered weight bits, our DeepSteal achieves better accuracy, fidelity and adversarial example attack efficacy. For instance, our substitute model can generate effective transferable adversarial example with similar efficacy (i.e., *∼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 adversary could recover 90% of the MSB bits (∼11.52% of total bits). By only utilizing the leaked MSB bits, the attacker can recover up to 89.05/88.17% test accuracy for the ResNet (18/34) models. Additionally, for the *All Bits* case in Table III, after paying an additional time cost (i.e., *1.66*×), the performance of DeepSteal attack only exhibits marginal improvement on residual models. In contrast, the larger model with a different architecture topology (i.e., VGG-11 with 132 Million parameters) highly benefits from the additional information of all the bits. For VGG, we observe a *∼3%* improvement by using all the filtered bits in comparison to using MSB only. 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, our substitute model generates adversarial examples demonstrate close to the white-box attack efficacy (i.e., within 2% of the best case result) with 4000 rounds of HammerLeak attack. As for VGG model, which is already known as a robust architecture [67], our substitute model generated adversary reaches within ∼12%-14% of an ideal white-box attack. Nevertheless, our attack efficacy still shows an improvement of about ∼25%-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 us 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 *7%-11%* for ResNet models and by *3%* 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. Comparison to State-of-the-art Techniques*

In Table IV, we summarize the standing of our DeepSteal attack compared with existing model recovery methods for three different domains of applications. We can see existing model regularization [11], [41] and data augmentation

TABLE IV: *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 onpar with the SOTA methods across all three evaluation metrics.*

| *Case Study* | *Method* | *Objective* | *Model* | *Accuracy (%)* | *Fidelity (%)* | *Accuracy Under Attack (%)* |
|---|---|---|---|---|---|---|
| *Regularization & Data Augmentation* | Fully-Supervised [41] | Accuracy/Fidelity | WideResNet-28 | 86.51 | 87.37 | - |
| | Rand-Augment [60] | Accuracy | WideResNet-28 | 87.4 | - | - |
| | Auto-Augment [61] | Accuracy | WideResNet-28 | 87.7 | - | - |
| | ***DeepSteal (ours)*** | ***Accuracy/Fidelity/Attack*** | ***WideResNet-28*** | ***91.93*** | ***93.45*** | ***0.05*** |
| *Model Extraction* | Side Channel [25], [28], [43], [45] | 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) [71] | Adversarial Attack | ResNet-18 | - | - | 20.47 |
| | White-Box (PGD/Trades) [67], [72] | Adversarial Attack | ResNet-18 | - | - | 0.0 |
| | ***DeepSteal (ours)*** | ***Adversarial Attack*** | ***ResNet-18*** | **-** | **-** | ***1.2*** |

techniques [60], [61] are useful in training deep models with limited data. However, our substitute model achieves a much higher accuracy (i.e., ∼*3%*). Other existing side channel attacks [25], [26], [28], [43], [45] fall into a similar attack category as DeepSteal. Among them, [26] 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 [25], [28], [43], [45] 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. Our DeepSteal can leverage 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 ∼18% improvement in accuracy and ∼61% improvement in degrading the accuracy under adversarial attack. Note that while DeepSteal is a semi-blackbox attack, DeepSteal actually can achieve 1.2% accuracy under attack, which is extremely close to a white-box attack performance (i.,e., 0%). We observe a 19% improvement in attack performance compared to a powerful black-box substitute model (e.g., Inception-V1) attack.

### *E. Impact of Bit Stealing Errors*

Bit stealing accuracy can be influenced in case expected bit flips do not occur. In this section, we analyze the bit errors in the rowhammer-based side channel. Specifically, we profile bit errors on 4 different vulnerable DIMMs with random bits set in the victim's pages. Note that HammerLeak only leverages *Strongly-leakable cells* that exhibit consistent bit flips in double-sided rowhammer. Our analysis reveals that about 70% of the flippable DRAM cells fall into this category. Our results show very high and stable bit stealing accuracy – on average 95.7% – across all tested DIMMs.

To quantify the impact of bit errors, we perform an analysis by evaluating the effectiveness of DeepSteal under a range of bit error rates. Specifically, under the setting where 90% of raw MSB bits are exfiltrated by HammerLeak, we inject random errors at certain rate across each model layer into the recovered bits. Figure 11 shows the performance of the substitute model (in terms of Accuracy and Accuracy Under Attack) when the bit error ranges from 0% to 10% for ResNet-18. The results demonstrate that *low bit error (0-5%) has negligible effect on the performance of the substitute model attack.* Moreover, the accuracy of the substitute model stays stable even as the error rate reaches 10%. On the other hand, the increase in error rate (5-10%) in the recovered bits causes the DeepSteal performance in Accuracy Under Attack to degrade gradually. Nevertheless, we can still observe much higher attack efficiency of DeepSteal compared to the baseline approaches as shown in Table IV.

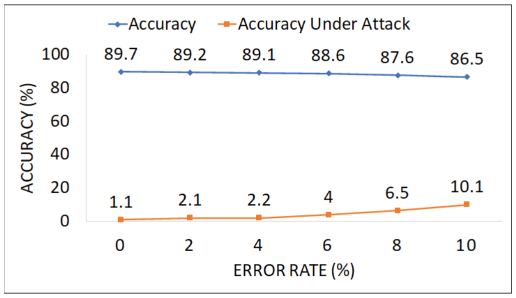


Fig. 11: *Analysis of the impact of the recovered Bit Error Rate (%) on DeepSteal attack performance for ResNet-18.*

## IX. Discussion

### *A. Applicability of DeepSteal in DDR4 Memory*

While our evaluation focuses on DDR3 memories, there is no fundamental constraint in the rowhammer vulnerability that limits DeepSteal from manifesting in DDR4-based systems. This is because the root cause of the information leakage is the *data-pattern dependent bit flip*, which has been shown to exists in DDR4 DIMMs in recent studies [73]–[75]. In fact, it has been observed that due to device scaling, the effective number of hammering needed to induce bit flip in DDR4 devices is significantly reduced, hence making DDR4 DIMMs potentially more vulnerable to rowhammer. Due to the deployment of Targeted Row Refresh (TRR), rowhammer attacks on DDR4 DIMMs need to integrate certain *hammer fuzzing patterns* with the activation of additional rows at a distance [76]. Such additional rows are accessed for the purpose of bypassing TRR while the target bit flip is still induced by the two primary aggressor rows, which is essentially identical to the double-sided rowhammering effect. Finally, same as DDR3-based attacks, for each bit flip, the more complex hammering pattern in DDR4 would still need to complete within one refresh cycle. Therefore, for the same amount of bit leakage, we expect very similar exploitation time of DeepSteal in DDR4-based systems.

### B. Countermeasures for DeepSteal

*1) Algorithm-level Mitigation:* Effect of adversarial training on transferred adversarial sample. One potential approach in defending against adversarial samples is to train the model using the attacked samples popularly known as *adversarial training [67]*. In table V, we evaluate the target victim model which is defended with adversarial training. Naturally, a model trained with adversarial examples becomes more resistant to both white-box & black-box adversarial attacks. We observe a similar pattern in our experiments. Still, our proposed DeepSteal could achieve *4% & 6%* improvement in attacking the target model trained with the adversarial samples compared to the baseline (i.e., architecture only + learning) for ResNet-18 & VGG-11, respectively. We conclude that the existing white-box adversarial defense may lower the transferability of adversarial samples from our substitute model, but fails to prevent the accuracy and fidelity extraction.

In addition, our proposed DeepSteal follows a more strict threat model which does not require access to output logits/predictions. In contrast, prior strong transferable adversarial attacks [77]–[80] require model queries as well as access to output logits/prediction. Even in this extremely limited setting, our attack DeepSteal outperforms the existing model stealing attacks (i.e., architecture only + learning) [28] in attacking a well-defended (i.e., adversarial training) target model. Finally, it is worth noting that current adversarial defenses (e.g., adversarial training) comes with additional training cost and inference accuracy degradation.

TABLE V: *After adversarial training, the white-box accuracy under attack improved to 43.12% & 35.71% for ResNet-18 and VGG-11 models respectively . Here we report the performance of DeepSteal attack using recovered bit information after 4000 rounds of Hammer-Leak attack.*

| *Training Data (%)* | *Accuracy (%)* | *Fidelity (%)* | *Accuracy Under Attack (%)* |
|---|---|---|---|
| | ***ResNet-18 (83.62%)*** | | |
| *Baseline* | 72.87 | 72.66 | 80.42 |
| *DeepSteal* | 82.84(↑ 10) | 81.67 (↑ 9) | 76.78 (↓ 4) |
| | ***VGG-11 (80.21%)*** | | |
| *Baseline* | 70.71 | 71.3 | 76.63 |
| *DeepSteal* | 80.65 (↑ 10) | 81.13 (↑ 10) | 70.28 (↓ 6) |

*2) System and Architecture Defense Approaches:* One possible way to mitigate the DeepSteal attack is to use software-based encryption schemes for streaming applications. For example, with software encryption, the ML framework can perform on-demand weight decryption during computation without storing the plaintext weight parameters in memory at any time. 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 [81]. This can accumulate substantially and result in potentially high run-time overhead, limiting the deployment of such implementation in DNN applications.

Alternative to the system-level protection schemes, DeepSteal can be potentially defeated by protecting the confidentiality and integrity of secretive pages through trusted execution environments (TEE) [82]. With TEE, pages that belong to a protected enclave are encrypted using processor-side memory encryption engine before leaving the processor chip. Hence, even with the rowhammer-based side channel leakage, the attacker cannot retrieve any information about the actual data. However, DNNs typically come with large memory footprints that exceed TEE's pre-allocated secure memory region. As a result, ML applications in contemporary TEE are subject to considerable runtime overhead [83] due to the expensive operations of page swapping between secure and unsecure memory regions. Therefore, we believe designing TEE architectures tailored for ML applications is critical in the future.

## X. Conclusion

The training of deep neural networks requires heavy computational resources and sensitive domain-specific private user data. Thus, any potential breach in model privacy through leakage of sensitive model parameters may cost the service provider a heavy financial penalty. Consequently, the IP of a pre-trained DNN model is critical to protect against adversarial threats (i.e., model extraction). In this work, our proposed DeepSteal attack exposes this threat of an effective model extraction attack in practical settings. In particular, our novel system-level weight bit extraction method HammerLeak enables fast and efficient weight stealing for large scale DNN applications. It can recover a significant portion of the weight bits of a DNN model with millions of wight parameters. On top of that, our proposed Mean Clustering training algorithm can leverage this information to effectively launch a strong adversarial input attack on the victim model. The efficacy of the proposed attack algorithm is validated through extensive experimental evaluation. Such a model extraction threat should encourage future work in this direction to protect the IP of large-scale DNN models.

## Acknowledgement

This work is supported in part by the National Science Foundation under Grant No. 2019548, No. 2019536, No. 1931871, and No. 2144751.

## Availability

Tools and useful code for DeepSteal system exploit are released in https://github.com/casrl/DeepSteal-exploit. The detailed models and dataset for our substitute model training can be accessed at https://github.com/ASU-ESIC-FAN-Lab/DeepStealSP2022.

## References

[1] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *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 *IEEE International Conference on Computer Vision*, 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 *USENIX Security Symposium*, 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 *IEEE Symposium on Security and Privacy*, 2020, pp. 1466–1482.

[7] A. Kwong, D. Genkin, D. Gruss, and Y. Yarom, "Rambleed: Reading bits in memory without accessing them," in *IEEE Symposium on Security and Privacy*, 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," in *ACM SIGARCH Computer Architecture News*, vol. 42, no. 3, 2014, pp. 361–372.

[10] Z. Kenjar, T. Frassetto, D. Gens, M. Franz, and A.-R. Sadeghi, "V0ltpwn: Attacking x86 processor integrity from software," in *USENIX Security Symposium*, 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 *USENIX Security Symposium*, 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 *USENIX Security Symposium*, 2020, pp. 1309–1326.

[13] T. Orekondy, B. Schiele, and M. Fritz, "Knockoff nets: Stealing functionality of black-box models," in *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 *IEEE International Joint Conference on Neural Networks*, 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 *ACM Asia Conference on Computer and Communications Security*, 2017, pp. 506–519.

[19] S. Milli, L. Schmidt, A. D. Dragan, and M. Hardt, "Model reconstruction from model explanations," in *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 *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 *IEEE symposium on security and privacy*, 2015, pp. 605–622.

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

[24] M. H. I. Chowdhuryy, H. Liu, and F. Yao, "Branchspec: Information leakage attacks exploiting speculative branch instruction executions," in *IEEE 38th International Conference on Computer Design*, 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] H. Yu, H. Ma, K. Yang, Y. Zhao, and Y. Jin, "Deepem: Deep neural networks model recovery through em side-channel information leakage," in *IEEE International Symposium on Hardware Oriented Security and Trust*, 2020, pp. 209–218.

[27] L. Batina, S. Bhasin, D. Jap, and S. Picek, "CSI NN: Reverse engineering of neural network architectures through electromagnetic side channel," in *USENIX Security Symposium*, Santa Clara, CA, Aug. 2019, pp. 515–532. [Online]. Available: https://www.usenix.org/conference/usenixsecurity19/presentation/batina

[28] 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 *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020, pp. 385–399.

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

[30] T. Nakai, D. Suzuki, and T. Fujino, "Timing black-box attacks: Crafting adversarial examples through timing leaks against dnns on embedded devices," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pp. 149–175, 2021.

[31] 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.

[32] 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.

[33] 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 *IEEE Symposium on Security and Privacy*, 2019, pp. 888–904.

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

[35] Z. Zhang, Z. Zhan, D. Balasubramanian, B. Li, P. Volgyesi, and X. Koutsoukos, "Leveraging em side-channel information to detect rowhammer attacks," in *IEEE Symposium on Security and Privacy*, 2020, pp. 729–746.

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

[37] 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.

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

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

[40] N. Carlini, M. Jagielski, and I. Mironov, "Cryptanalytic extraction of neural network models," in *Annual International Cryptology Conference*. Springer, 2020, pp. 189–218.

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

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

[43] 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 *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2020, pp. 125–137.

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

[45] 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.

[46] Z. Zhan, Z. Zhang, S. Liang, F. Yao, and X. Koutsoukos, "Graphics peeping unit: Exploiting em side-channel information of gpus to eavesdrop on your neighbors," in *IEEE Symposium on Security and Privacy*, 2022.

[47] R. Callan, A. Zajić, and M. Prvulovic, "Fase: Finding amplitude-modulated side-channel emanations," in *IEEE Annual International Symposium on Computer Architecture*, 2015, pp. 592–603.

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

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

[50] 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.

[51] 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.

[52] 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*, 2018, pp. 245–261.

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

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

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

[56] 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.

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

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

[59] 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.

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

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

[62] 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.

[63] 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.

[64] 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, Aug. 2016, pp. 1–18. [Online]. Available: https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/razavi

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

[66] 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.

[67] 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

[68] 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.

[69] 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/

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

[71] 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 *IEEE International Conference on Image Processing*, 2020, pp. 648–652.

[72] 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.

[73] H. Hassan, Y. C. Tugrul, J. S. Kim, V. Van der Veen, K. Razavi, and O. Mutlu, "Uncovering in-dram rowhammer protection mechanisms: A new methodology, custom rowhammer patterns, and implications," in *IEEE International Symposium on Microarchitecture*, 2021, pp. 1198–1213.

[74] L. Orosa, A. G. Yaglikci, H. Luo, A. Olgun, J. Park, H. Hassan, M. Patel, J. S. Kim, and O. Mutlu, "A deeper look into rowhammer's sensitivities: Experimental analysis of real dram chipsand implications on future attacks and defenses," in *IEEE International Symposium on Microarchitecture*, 2021, pp. 1182–1197.

[75] "Half-Double Next-Row-Over Assisted Rowhammer." [Online]. Available: https://github.com/google/hammer-kit/blob/main/20210525_half_double.pdf

[76] P. Frigo, E. Vannacc, H. Hassan, V. Van Der Veen, O. Mutlu, C. Giuffrida, H. Bos, and K. Razavi, "Trrespass: Exploiting the many sides of target row refresh," in *IEEE Symposium on Security and Privacy*, 2020, pp. 747–762.

[77] W. Zhou, X. Hou, Y. Chen, M. Tang, X. Huang, X. Gan, and Y. Yang, "Transferable adversarial perturbations," in *European Conference on Computer Vision*, 2018, pp. 452–467.

[78] C. Xie, Z. Zhang, Y. Zhou, S. Bai, J. Wang, Z. Ren, and A. L. Yuille, "Improving transferability of adversarial examples with input diversity," in *IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2019, pp. 2730–2739.

[79] F. Liu, C. Zhang, and H. Zhang, "Towards transferable adversarial perturbations with minimum norm," in *ICML Workshop on Adversarial Machine Learning*, 2021.

[80] M. Salzmann *et al.*, "Learning transferable adversarial perturbations," *Advances in Neural Information Processing Systems*, vol. 34, 2021.

[81] 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.

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

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

[84] Z. He, A. S. Rakin, and D. Fan, "Parametric noise injection: Trainable randomness to improve deep neural network robustness against adversarial attack," in *IEEE Conference on Computer Vision and Pattern Recognition*, 2019.

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

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

[87] 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.
[88] S. Zagoruyko and N. Komodakis, "Wide residual networks," *arXiv preprint arXiv:1605.07146*, 2016.
[89] 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

## XI. Appendix

### A. 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 VI, 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.

### B. Analysis of Data Availability

Having access to a small portion of data (2-8% available data) to train a substitute model is a common practice and used in prior works [60], [61]. Following this practice, our threat model assumes the attacker has access to a portion of the dataset (train/test), even with a tiny fraction. In this ablation study, we demonstrate how variable availability of data affect the adversarial attack performance in **table VII**. It shows that our substitute model's accuracy and fidelity improve significantly with the increase in available training data. However, even with only 2% training data, we could recover the accuracy of the substitute model up to ∼ 80% (ResNet-18) and ∼ 76% (VGG-11). In general, increasing data availability from 2-8% increases the transferability of the adversarial samples for both models. Note that, 0% train data availability makes it a semi-supervised or unsupervised learning domain problem. We acknowledge that semi supervised techniques [11] may have their own pros and cons in constructing a proper substitute model. However, due to different experimental settings (i.e., access to unlabeled data) and threat model (i.e., model query access), exploration of 0% train availability case is beyond the scope of this work.

### C. Further Analysis of the Recovered Bit Error

In fig. 12, we have presented a bit error analysis of the VGG-11 model. Similar to ResNet-18, we observe a small fluctuation (e.g., 1-2%) in accuracy within the low bit error range (e.g., 0-4%). However, one notable difference for the VGG-11 model is the accuracy under attack increases linearly with increasing bit error rate. The result is expected, as being a dense model, VGG-11 intrinsically shows much higher resistance to adversarial examples. Such observation supports the conclusion of prior works [67], [84] about denser models being more resilient to adversarial examples.

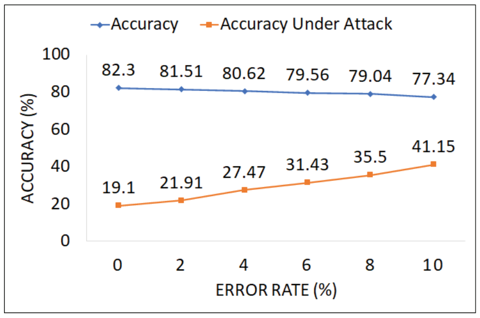


Fig. 12: ***Theoretical analysis of the impact of the recovered Bit Error Rate (e) % on DeepSteal attack performance.*** *In this ablation study, we vary the recovered bit error range between 0-10% and report the corresponding accuracy of the substitute model and accuracy under attack of the target model for VGG-11.*

### D. DeepSteal Attack for Full-Precision Models

Our proposed DeepSteal is a general method that could also be applied to floating-point/32-bit precision models. For example, the IEEE representation format of floating-point numbers contains the sign bit at the Most Significant Bit (MSB) location. In table VIII, we adapted our algorithm to train the substitute model for a full-precision target model. The result shows our attack could achieve 0.95% and 14.5% accuracy under attack for a full-precision target ResNet-18 and VGG-11 model, respectively. The result presented for both models is similar to our prior results for the 8-bit quantized models.

### E. Comparison between Cross-Entropy based Training and Algorithm-1

To quantify the effect of the proposed Mean Clustering loss term, in this study, we only train the substitute model with Cross-entropy loss by removing the second loss penalty in eq. (1). In table IX, we summarize the effect of our second loss term during substitute model training. Here We train two substitute models for both ResNet-18 and VGG-11: i) with cross-entropy training & ii) with proposed Mean Clustering penalty (i.e., algorithm-1). The result shows our

TABLE VI: *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 VII: *In this ablation study, we vary the percentage of data available to the attacker between 1-8% and report the results. As a baseline, we report the original architecture only information case with 8% available data. For DeepSteal, here we use 4000 rounds of HammerLeak attack information.*

| *Training Data (%)* | *Accuracy (%)* | *Fidelity (%)* | *Accuracy Under Attack (%)* |
|---|---|---|---|
| ***ResNet-18*** | | | |
| Baseline (8%) | 73.18 | 74.29 | 61.33 |
| 1 | 71.22 | 71.98 | 44.47 ↓ |
| 2 | 79.21 | 80.46 | 24.74 ↓ |
| 4 | 84.86 | 86.27 | 8.56 ↓ |
| 8 | 89.05 | 90.74 | 1.94 ↓ |
| ***VGG-11*** | | | |
| Baseline (8%) | 70.76 | 72.06 | 61.19 |
| 1 | 68.66 | 69.65 | 51.91 ↓ |
| 2 | 75.8 | 76.91 | 38.82 ↓ |
| 4 | 80.81 | 82.08 | 25.96 ↓ |
| 8 | 84.59 | 86.24 | 16.87 ↓ |

TABLE VIII: *This table presents the results after attacking a full-precision model with 4000 rounds of HammerLeak. The baseline is again the architecture information only case.*

| *Training Data (%)* | *Accuracy (%)* | *Fidelity (%)* | *Accuracy Under Attack (%)* |
|---|---|---|---|
| ***ResNet-18 (93.17%)*** | | | |
| *Baseline* | 72.2 | 73.2 | 53.94 |
| *DeepSteal* | 89.78 | 91.6 | 0.95 |
| ***VGG-11 (91.2%)*** | | | |
| *Baseline* | 71.66 | 72.57 | 61.0 |
| *DeepSteal* | 83.01 | 84.65 | 14.5 |

proposed DeepSteal algorithm achieves a *7-8%* improvement in accuracy and fidelity compared to the traditional cross-entropy based training scheme for ResNet-18. The proposed algorithm-1 also helps improve the adversarial attack efficacy by *26%*. Similarly, for the VGG-11 model, our proposed algorithm-1 improves the test accuracy of the substitute model by 3% and under attack accuracy by 7%. This ablation study validates the effectiveness of our proposed *Mean Clustering* algorithm in comparison to cross-entropy training, which prior side-channel attacks [28] have adopted to recover the weight information.

In addition, the mean clustering penalty is only applied to the weights with more significant bits recovered, leading to a known projected range, where the penalty term guides such 'partially recovered weights' to be learned close to its projected range mean. For those un-recovered less significant bits, we believe there would be a bias due to mean clustering penalty, but may not be 'undesired' in this problem, for three main reasons: First, the mean clustering penalty is a 'soft' penalty with a small value of $\lambda$ (e.g., $1^{-4}, 5^{-5}$) in the loss function. This penalty term will favor the weights to converge 'near' the known mean of the projected range based on the recovered partial bit info, not precisely equal to the mean. Second, during training, the cross-entropy loss (another loss term) and the gradient of the weights will play a more important role in dictating the final convergence value of those weights. Hence, the first loss term should help reduce the bias effect from the mean penalty loss to some extent. Third, our experiment results, such as accuracy under adversarial example attack, validate that such bias does not play a negative effect, as our DeepSteal gets almost similar attack accuracy as the white-box attack using a recovered substitute model.

TABLE IX: *In this ablation study, we show the effectiveness of our proposed loss function in algorithm-1 by comparing it with Cross-Entropy loss training using the exact same HammerLeak attack information (i.e., after 3000 rounds).*

| *Loss Function* | *Accuracy (%)* | *Fidelity (%)* | *Accuracy Under Attack (%)* |
|---|---|---|---|
| ***ResNet-18*** | | | |
| *Cross-Entropy* | 78.81 | 80.25 | 31.26 |
| *Algorithm-1* | 86.32 (↑ 8) | 87.86 (↑ 7) | 5.24 (↓ 26) |
| ***VGG-11*** | | | |
| *Cross-Entropy* | 78.65 | 79.84 | 43.03 |
| *Algorithm-1* | 81.03 (↑ 3) | 82.88 (↑ 3) | 36.45 (↓ 7) |

### *F. Dataset and Model Architecture Description*

We take three visual datasets: CIFAR-10 [85], CIFAR-100 [86] and German Traffic Sign Recognition Benchmark (GTSRB) [87] for object classification task. CIFAR-10 contains 60K RGB images in size of $32 \times 32$. 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 (result in appendix), we use 40k labeled 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 classes. For CIFAR-10 experiments, we used residual networks (e.g., ResNet-18/34,Wide-ResNet-28) [1], [88] and VGG-11 [38] architectures. For GTSRB and CIFAR-100, we adopted ResNet-18 and ResNext-50-32x4d [89] as the evaluation architecture respectively. For all experiments, the weights of each victim model are quantized into 8 bit.

Following the standard practice in [60], [61], 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 settings for the 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 worst (e.g., accuracy) of the three rounds.