Yes, One-Bit-Flip Matters!
Universal DNN Model Inference Depletion with Runtime Code Fault Injection

Shaofeng Li
Peng Cheng Laboratory

Xinyu Wang
Shanghai Jiao Tong University

Minhui Xue
CSIRO's Data61

Haojin Zhu✉
Shanghai Jiao Tong University

Zhi Zhang✉
University of Western Australia

Yansong Gao
CSIRO's Data61

Wen Wu
Peng Cheng Laboratory

Xuemin (Sherman) Shen
University of Waterloo

Abstract

We propose, FrameFlip, a novel attack for depleting DNN model inference with runtime code fault injections. Notably, FrameFlip operates independently of the DNN models deployed and succeeds with only a single bit-flip injection. This fundamentally distinguishes it from the existing DNN inference depletion paradigm that requires injecting tens of deterministic faults concurrently. Since our attack performs at the universal code or library level, the mandatory code snippet can be perversely called by all mainstream machine learning frameworks, such as PyTorch and TensorFlow, dependent on the library code. Using DRAM Rowhammer to facilitate end-to-end fault injection, we implement FrameFlip across diverse model architectures (LeNet, VGG-16, ResNet-34 and ResNet-50) with different tasks (FMNIST, CIFAR-10, GTSRB, and ImageNet). With a single bit flipping, FrameFlip achieves high depletion efficacy that consistently renders the model inference utility as no better than guessing. We also experimentally verify that identified vulnerable bits are almost equally effective at depleting different deployed models. In contrast, transferability is unattainable for all existing state-of-the-art model inference depletion attacks. FrameFlip is shown to be evasive against all known defenses, generally due to the nature of current defenses operating at the model level (which is model-dependent) in lieu of the underlying code level.

1 Introduction

Deep neural networks (DNNs) have demonstrated impressive performance on various tasks [19, 26, 70, 71]. However, their security and safety usage is threatened by adversarial attacks that are generally introduced either during the model training phase or the model inference phase [31]. Training phase attacks conventionally include data poisoning attacks, which tamper with the model prior to deployment such that it deviates from its benign behavior on either all inputs [32] (e.g., model utility depletion) or specific inputs with triggers [21] (i.e., backdoor attacks). A notable attack at the model inference phase is the adversarial example attack [59].

Conventional attacks attempt to breach either model integrity (i.e., training phase attack) or data integrity (i.e., inference phase attack). However, they do not target the integrity of the hardware on which DNN models deploy and execute. Recent work has looked at a new type of fault injection attack [12, 29, 44, 53, 60, 68], where model integrity is violated by compromising the underlying hardware. It tampers the model in a manner similar to the conventional training phase attack *but occurs after the model deployment.* Consequently, all countermeasures applied prior to deployment are inapplicable. In contrast to the conventional inference phase attack (i.e., adversarial examples), which only manipulates each incoming input, the fault injection attack completely contaminates the underlying model to compromise all upcoming inputs.

These studies demonstrate the feasibility of launching fault injection attacks to compromise DNN model integrity after deployment. However, there are still some challenges that need to be overcome. Firstly, these works are all model dependent. A DNN model has some tolerance on its weight value change unless specific positional weights are delicately changed. The positions of those critical weights are unique to each victim model, and cannot be transferred to a different model. Secondly, they all require injecting multiple bits of fault deterministically and simultaneously, which is extremely challenging in practice. This is because flippable memory cells in the DRAM are sparse and it is hard to find a physical memory page that contains more than one flippable bit [29, 60]. In addition, injected faults may need to be increased as the model size increases. Thirdly, to be efficient, these attacks all assume to have full knowledge of the victim model – that is, white box access – which may not always be available. This is problematic when the deployed model, and its weight values, are updated through online learning, which renders the previously identified vulnerable bits futile. To this end, we are interested in the following research question:

Is it practical to universally breach the post-deployment

✉Haojin Zhu (zhu-hj@cs.sjtu.edu.cn) and Zhi Zhang (zhi.zhang@uwa.edu.au) are the corresponding authors.

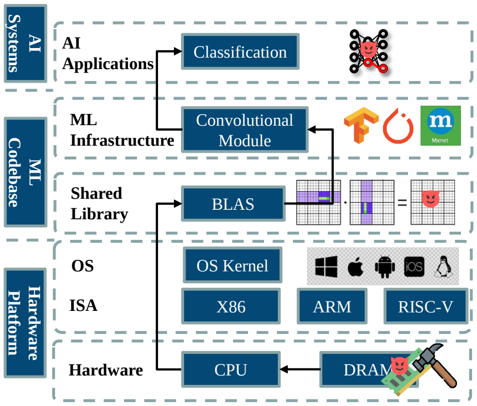


Figure 1: Attack flow of FrameFlip.

DNN model integrity with a single-bit fault injection given black-box victim models?

General Challenges. This work provides an affirmative answer to the above research question. However, given the extremely stringent constraints, there are three crucial general challenges (GCs):

- **GC1: Model-independence.** The exact bits of the model weights that ultimately affect the inference result vary from model to model, and even vary for the same model between updates, e.g., through fine-tuning. In this context, identifying a subset of bits that can be universally applied to every model is challenging.

- **GC2: Black-box knowledge.** As for a DNN model already deployed on a server, in a typical application of machine learning as a service, the model provider normally only offers an API interface to provision inference results. Therefore, accessing the underlying model is infeasible. In addition to **GC1**, this black-box condition (no knowledge of model weights, or even model architecture) makes faulty bit identification seemingly unattainable.

- **GC3: Practical single-bit injection.** While it is possible to inject multiple faults into the hardware, primarily in the main memory of the server, inducing a single-bit fault deterministically is realistic. However, constrained by **GC1** and **GC2**, reducing a set of faulty bits to a single bit that is able to universally affect any unknown models seems almost impossible.

Our Solutions. If we follow the existing paradigm of fault injection attacks on model inference, where the attack is conducted from the model level, it is almost impossible to resolve the research question when constrained by the above three general challenges. We overcome this hurdle by looking at the fundamentally different paradigm of the underlying *code level*. At a high level, we inject faults into the compiled running code library to consequently corrupt all model inferences that are supported by the library for computation (see an overview of the attack flow in Fig. 1). More specifically,

- To **address GC1**, the target runtime codebase supporting the model inference is independent of the specific upper-layer models.

- To **address GC2**, the libraries that are widely adopted across mainstream machine learning frameworks are identified. Faults induced into these libraries affect any model that must require these libraries' support.

- To **address GC3**, the branch condition of the code is targeted to alter the control flow with a single bit flip to amplify the fault adverse effect with a single-bit flip.

For **GC1** and **GC2**, we identify libraries that provide fundamental functionalities across prominant machine learning frameworks such as PyTorch [50], Tensorflow [3], Caffe [33], and Apache MXNet [1]). The linear algebra backend is a basic module of these Machine Learning (ML) frameworks, which has critical effects on the DNN's performance. To date, these ML frameworks rely on high-performance Basic Linear Algebra Subprograms (BLAS) to implement their linear algebra backend. BLAS is a de facto standard for low-level linear algebra routines, such as vector addition and dot product. Popular BLAS implementations include OpenBLAS [64], Eigen [25] and Intel MKL [61]. These implementations have been widely applied to mainstream ML frameworks. Therefore, the linear algebra backend is chosen as the targeted library.

To address **GC3**, the combined restrictions of single-bit fault determination and practicality present three technical challenges. The first technical challenge is to determine salient vulnerable bits within the identified library—ultimately, a single salient bit. We solve this by traversing the `cblas_dgemm` function of the OpenBLAS library and choosing all conditional branch instructions as vulnerable bit candidates. The reason is that a branch instruction has a significant effect on the final computation result due to control flow change. More specifically, for each vulnerable candidate, we manipulate its condition and switch it to the other traversed path. Then by evaluating the inference accuracy on the flipped instruction, we can determine the most vulnerable code point that exhibits the worst utility deterioration. We have designed an automatic and efficient vulnerable bit search scheme fulfilling the above goal by leveraging LLVM tools.

The second technical challenge towards practical single-bit injection is to retain stealthiness. The injected fault tampers with the control flow of the inference routine. In addition to the desired degradation of inference accuracy, the fault injection can lead to system warning notifications or result in program crash [29]. Our extensive experiments confirm that the compromised control flow of the inference routine does throw warning notifications, including *i)* error and warning messages from the DNN runtime process, *ii)* abnormal fluctuation in the memory usage statistics, and *iii)* a denial-of-service incident at the inference service. Obviously, the occurrence of either one of the listed warning signs notifies the victim of a possible DNN fault injection exploitation, reducing the feasibility of the attack. To circumvent this technical challenge, we introduce a more controlled fault injection primitive, *opflip*, to flip the opcode of an instruction to its adjacent instruction as, in most cases, the bit flips in opcodes will yield other valid opcodes in the x86 instruction set [24]. In this context, we identify that there is a type of instruction, conditional jump instruction, whose one-bit adjacent instruction is a valid instruction with the exact opposite semantics. By compromising this instruction with a single bit fault, the control flow of the DNN inference computation can be corrupted while avoiding exceptions. Thus, the fault injection is stealthy.

The third technical challenge of a practical single-bit injection is the viable means of injecting the fault. To be practical, the attacker has to be an unprivileged user who accesses the victim model on a deployed server machine, e.g., co-resident tenants in the MLaaS cloud. Therefore, fault injection requiring physical/local access to the machine (i.e., radiation-induced bit flip in main/DRAM memory [8]) is prohibited. In this work, we leverage the Rowhammer attack to manipulate a code page that resides in the address space of another process. We exploit memory deduplication [9, 55], a kernel feature that most operating systems support. Memory deduplication is a space-reduction scheme that allows identical virtual pages, held by multiple processes, to be mapped to one physical page, as long as none of them writes to this page. As Rowhammer is capable of flipping bits in DRAM, flipped code segmentations of the compiled ML codebase are located in the page cache. After that, the compromised page remains cached in the page cache. The OS does not detect this change as it is directly made in the hardware by a completely isolated process and it keeps providing the page cached modified copy to the victim on subsequent accesses. Therefore, the effect will synchronously appear on both sides since they essentially map to the same page.

By systematically resolving three general challenges and their associated technical challenges, we are now able to demonstrate an end-to-end universal DNN model inference depletion attack with a single bit flip. Notably, our attack debunks all existing defenses because they only consider model level fault injection attacks and not the lower code level fault injection attack. We analyze existing prominent countermeasures in terms of effectiveness and performance overhead. The results are summarized in Tab. 1.

In summary, we make the following key contributions:

Table 1: The effectiveness and efficiency of the SOTA countermeasures against miscellaneous fault-injection attacks. (●: Effective, ◐:Effective while Inefficient, ○: Ineffective)

Proposed Countermeasures	RFA [42]	BFA [53]	TBT [54]	TA-LBF [6]	CFT+BF [60]	Ours
Aegis [62]	●	●	●	●	●	○
DeepDyve [42]	●	●	●	●	○	○
Binarization [28]	◐	◐	◐	◐	◐	○
Weight Clustering [28]	●	●	●	○	○	○
Weight Encoding [43]	●	●	●	●	◐	○
RADAR [39]	●	●	●	●	◐	○
SentiNet [15]	○	○	●	●	◐	○
Weight Reconstruction [40]	●	●	●	●	○	○

- We reveal a new paradigm of universally depleting DNN model inference by generally injecting fault into the running compiled code, requiring only a single-bit flip to stealthily and completely alter the program control flow. The FrameFlip is the first end-to-end demonstrated attack under the practical constraints of black-box knowledge and model-independence.

- We devise a new automatic algorithm AutoVIS to identify vulnerable instructions in commonly-used machine learning code libraries. Our critical instruction search scheme can measure the influence of each instruction on the DNN model's utility when those instructions are flipped with a single bit fault.

- We evaluate the effectiveness of FrameFlip on 10 groups of DNN benchmarks. FrameFlip outperforms the state-of-the-art (SOTA) DNN prediction degradation attacks implemented by tampering with the DNN weight parameters. Significantly, FrameFlip, for the first time, exhibits high attack transferability across different DNN models.

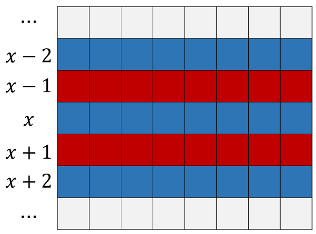
- We investigate several state-of-the-art mitigation techniques to prevent DNNs from fault injection attacks. Experiments show that our proposed attack can successfully circumvent state-of-the-art countermeasures.

Ethical Considerations. Our FrameFlip exploits a publicly known Rowhammer bug [35], and thus there is no need to report it.

2 Background

2.1 Machine Learning Codebases

Implementing a fully functional DNN from scratch is an extremely demanding task since it requires proficient coding skills and cross-domain expertise, e.g., algorithm optimization and hardware acceleration. Therefore, the modern DNN development pipeline is supported by industrial ML codebases (e.g., PyTorch [50], Caffe [33], TensorFlow [3] and Apache MXNet [1]). Those ML codebases are comprised of open-source repositories and off-the-shelf modules provided and maintained by commercial vendors and thousands of contributors. ML codebases' functions (e.g., training and inference) rely heavily on tiled GEMM (Generalized Matrix Multiply) which is implemented by high-performance BLAS (Basic Linear Algebra Subprograms) libraries. BLAS is a de facto standard for low-level linear algebra routines, which has



(a) Demonstration of Double-sided Rowhammer.

```
loop:
    mov (x-1), %r10
    mov (x+1), %r10
    clflush (x-1)
    clflush (x+1)
    jmp loop
```

(b) Code Snippet of Double-sided Rowhammer.

Figure 2: Double-sided Rowhammer

extensively optimized blocked matrix multiply. Examples of such libraries include OpenBLAS [64], Eigen [25], and Intel MKL [61].

2.2 The Rowhammer Bug

DRAM (Dynamic Random Access Memory) is organized in multiple memory channels. Each channel serves as a link between the DRAM module (DIMM) and its corresponding memory controller which is usually integrated into processors. A Dual Inline Memory Module (DIMM) is a physical memory module attached to the motherboard, with one or two ranks (on the front-side and back-side of the module). Each rank usually has 8 banks for DDR3-DRAM and 16 banks for DDR4-DRAM. A bank is the minimal unit the memory controller can control and is a grid of memory cells arranged in rows (wordlines) and columns (bit-lines). Each memory cell has a capacity which can be charged and discharged. A transistor controls access to the content of the capacity. When reading a row, the memory controller activates the wordline. The transistors in the activated row are opened and the content of all the capacitors on that row are discharged to the bitlines. Sense amplifier circuitry on each bitline captures and amplifies the signal and stores the result in the row buffer, and also refreshes the charge in the active row.

The Rowhammer bug refers to a hardware vulnerability validated on various DRAM chips [35]. As demonstrated in Fig. 2, if an attacker rapidly accesses two DRAM rows (aggressor rows) with row index $x-1$ and $x+1$ (i.e., *double-sided Rowhammer*), this will result in electromagnetic interference in row x, making its stored values flipped. This means that if a memory cell in row x originally stores 0, then it will be flipped to 1, or vice versa.

To mitigate the Rowhammer bug, DRAM vendors have implemented hardware solutions in recent DRAM modules (e.g., DDR4), such as Target Row Refresh (TRR) [48]. However, the TRR solution has been bypassed by TRRespass [20] via the so-called *many-sided Rowhammer*. To date, the Rowhammer bug still remains a threat to commodity DRAMs.

3 Threat Model

3.1 Victim's Capability

Trustworthy Model Training. The models are benign in the sense that the training process is not tampered with by any malicious party. The models are free from algorithm-based adversaries, e.g., model poisoning [32] and Trojan attacks [41, 45]. This is a practical assumption for well-trained models. The models can be either trained on a secure training device or downloaded from a trusted source. In both cases, the training process is not tampered with. This is in contrast to existing algorithm-based attacks that inject stealthy payloads to the DNN model and re-distribute it to victim users (e.g., model poisoning and Trojan attacks).

Resource-Sharing Platforms. Following prior bit-flip-based attacks [29, 54, 68], we assume that a trained deep learning model is deployed on a resource-sharing platform that provides an inference-phase service. This assumption is feasible, as current MLaaS jobs are deployed on clouds [17, 55, 66].

Online Learning. During inference, the trained DNN model is loaded into the system's (shared) memory and applied to the real world. However, in most inference practices, a DNN that achieves high evaluation results in training does not perform as effectively for real-world samples. This is mostly caused by the distribution discrepancy between the real-world data and the training data [57]. To tackle this issue, a simple but efficient approach is to continually fine-tune the trained DNN with more real-world samples after it goes online. Therefore, the model weights may or may not get updates, and the attacker is unaware of either circumstance. This deployment paradigm is becoming a typical scenario due the prevalence of MLaaS platforms [57, 69]. Note that, for those online learning cases, existing weight-based fault injection attacks [12, 29, 53, 54, 60, 68] are invalid because of their strong dependency on the DNN's weight parameters. Once the model weights are updated, the vulnerable bits that need to be flipped may also change.

Deployment of Defenses. We assume the victim can apply any SOTA defense against inference depletion attacks.

3.2 Attacker's Capability

Attacker's Goal. The attacker aims to deplete the DNN's inference utility by injecting a single-bit error into the runtime ML codebase while maintaining maximal stealthiness. The injected error tampers with the normal control flow of the inference routine, causing a degradation of inference accuracy. Meanwhile, the attacker aims to induce as few warnings as possible, to make the attack imperceptible as well as to extend the living time of the injected error. When a fault is injected, its adverse effect remains effective until a system reboots.

System Side. We assume that the attacker process can co-locate with the target DNN service, sharing computation resources with the victim's process. Specifically, the attacker is an unprivileged user who shares the same physical memory as the victim. This is a common assumption adopted by previous works [24, 29, 68]. In this paper, we leverage double-sided Rowhammer and many-sided Rowhammer for DDR3 and DDR4 chips, respectively, as the primitive of software-induced DRAM fault injection. Both hammering techniques need to know the DRAM memory address mapping function. We assume the adversary can obtain the DRAM addressing scheme by applying reverse engineering techniques [51, 63]. We assume that the OS of the resource-sharing platform is secure. Particularly, the system has installed an up-to-date administrative program, which implements necessary software-level confinement policies such as process isolation.

Model Side. The threat model assumptions of existing works [12, 29, 53, 54, 68] are conventional white-box attack approaches, that is, an adversary is assumed to have access to the network architecture, weight values and one batch of test data. Note that it is feasible to steal the model architecture [22] and model weights [52] through the side-channel information in the MLaaS setting. In contrast to them, we void such assumptions by adopting a black-box setting where no prior knowledge of a target network architecture (including weights at deployment and online weight updates) is required as the transferability of our attack applies to various network architectures (reported in our experiments).

ML Codebase Side. Different to existing works that require the attacker to know model parameters, this work requires the attacker to know which ML frameworks are utilized by the model. Following Yan et al. [67], we believe the knowledge of open-source ML frameworks is widely used and publicly available (e.g., Tensorflow, Pytorch, Caffe, and MXNet), while the victim's DNN weight parameters are valuable intellectual property and are private. Hence, it is easier for an attacker to acquire knowledge about the ML frameworks than the weight parameters of the DNN. For the model inference frameworks that are not within the set of the mainstream ML frameworks mentioned above, their BLAS backends may still be one of the reputable, efficient, and standard linear algebra libraries, such as OpenBLAS, Eigen, and MKL. As implementing these libraries requires advanced expert knowledge of both algorithms and hardware to achieve optimal performance, it is unnecessary and difficult to implement these BLAS backends from scratch. Once ML frameworks adopt the linear algebra libraries mentioned above, they become vulnerable to our attacks. Notably, all mainstream ML frameworks (e.g., TensorFlow, PyTorch, Caffe, and MXNet) are supported by the OpenBLAS library. Thus, our security analysis can be practically applied to all of these frameworks.

Options for Linking. Static linking links related library functions directly into a victim's binary, and thus the linked functions cannot be called by other binaries. Consequently, FrameFlip cannot find the exact physical locations of vulnerable bits of library functions in the victim process and instead might cause the victim to crash. However, static linking generates large-size binaries and dynamic linking produces small-size binaries. Thus, developers are likely to use dynamic linking in practice to distribute their models. Therefore, in this work, we assume the victim adopts dynamic linking to link object files into an executable output file.

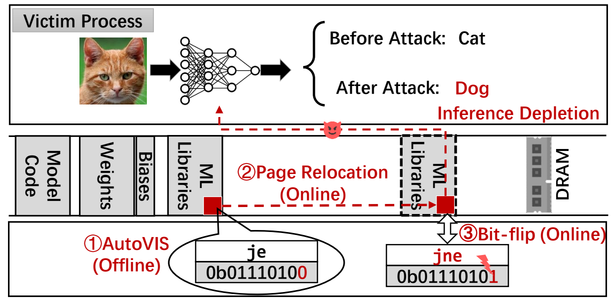


Figure 3: An overview of FrameFlip.

4 FrameFlip Design

In this section, we present FrameFlip in detail. FrameFlip is comprised of three steps: the vulnerable bit search in ML codebases, memory massage, and single-bit fault injection. To clarify the notations, we call bit flips found in the victim's physical memory "flippable bits", and bit flips found in ML codebase search "vulnerable bits". Fig. 3 demonstrates the overview of the proposed FrameFlip.

The attack is comprised of offline and online phases. In the offline phase, the attacker scans the ML codebase via an automatic vulnerable bit identification algorithm (detailed in Section 4.2) to select the instruction to be flipped in the online phase. In the online phase, the attacker scans the target memory to identify a suitable bit flip, and exploits the memory waylaying to relocate the vulnerable instruction into the matched memory page (i.e., the flippable page has the same page offset and flipping direction). At last, the attacker employs a hammering technique to flip the bit in the specific memory cell.

4.1 Vulnerable Bit Search in ML Codebase

In this step, the adversary identifies the bit to be flipped in the ML codebase. Note that in our threat model, attackers do not need to manipulate the codebase repository and redistribute it. The attacker is only required to perform vulnerability analysis on the codebase.

4.1.1 Linear Algebra Backend

ML codebases invoke miscellaneous shared libraries (`ELF` files) to support its foundational functions. In practice, a function called GEMM, a part of the BLAS (Basic Linear Algebra

Subprograms) library is invoked to make deep neural networks perform faster and more power efficient. Note that these shared libraries are optimized for algorithms and hardware, and thus possess high performance on time and space complexity. We choose the shared library of the linear algebra backend as the attack module because of its indispensability and wide adoption. Implementations of the linear algebra backend used by ML codebases are compiled into linkable object files with file extensions such as `.so` (Shared libraries) for Unix-based systems. A shared library is comprised of several sections specified in a section header table. The most important section is the `.text` section which holds the executable instructions of the library. For a shared library, its `.text` section is a list of independent functions that can be invoked by external `ELF` files.

The shared library that contains the linear algebra backend implements a set of APIs defined in the Basic Linear Algebra Subprograms (BLAS) standard. For instance, scalar and vector operations are defined in level 1 BLAS, including the dot product (DOT: $\boldsymbol{x}^T \boldsymbol{y}$) and vector scaling (SCAL: $\alpha \boldsymbol{x}$). Vector-matrix operations are defined in level 2 BLAS. The operations include general matrix-vector multiplication (GEMV: $\alpha A\boldsymbol{x} + \beta \boldsymbol{y}$) and general rank 1 operation (GER: $\alpha \boldsymbol{x}\boldsymbol{y}^T + A$). Matrix-matrix operations are defined in level 3 BLAS and include general matrix multiply (GEMM: $\alpha AB + \beta C$).

In the next subsection, we present a method for identifying a vulnerable bit in ML codebases that can be exploited by an attacker. Our approach involves a meticulous code analysis of the linear algebra backend, examining the control flow graph from junction blocks to individual instructions, and ultimately identifying and locating the vulnerable bit.

4.1.2 Vulnerable Code Statements: Branch Statements

In code analysis, a *control-flow graph (CFG)* is a graph representation of all paths that might be traversed through a program during its execution. In a CFG, each node represents a *basic block*, which is a straight-line piece of code sequence with no branches in except at the entry and no branches out except at the exit. In particular, the processor must execute the entire basic block from start to finish and there are no instructions that can leave this basic block except the entry and exit points. This restricted form makes a basic block highly amenable to program analysis. Directed edges in the CFG represent the relationships among blocks, i.e., jumps in the control flow. By definition, there is an edge from block b_1 to block b_2 if and only if the code in b_2 can be executed immediately after the code in b_1. Fig. 4 demonstrates the CFG of a demo binary search program. In the assembly code we can find that conditional and unconditional branch instructions that comprise the global control logic of the program, as represented by the CFG.

Compared with the basic blocks, most of which are not invoked by the control logic of the program, the *control flow statements* (edges in CFG) are critical to controlling the execution logic of the whole program. More specifically, the control flow statements can alter the contents of the CPU's Program Counter (PC). The PC maintains the memory address of the next machine instruction to be fetched and executed. Therefore a control instruction, if executed, causes the CPU to execute code from a new memory address, changing the program logic. Control flow statements mainly contain three types of control instructions, listed as follows:

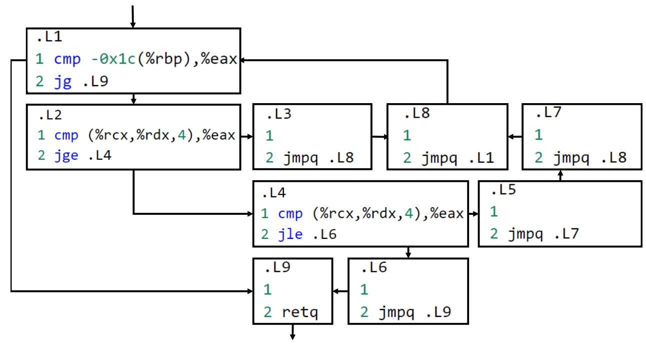


Figure 4: Demonstration of CFG.

Jump instructions directly modify the value of the PC with another basic block's entry point other than being incremented past the current instruction to its next instruction. Typical jump instructions in assembly language include `je`, `jne`, `jg`, and `jle`. Jumps typically have unconditional and conditional forms where the latter may be taken or not taken (the PC is modified or not) depending on some conditions.

Call instructions are used to implement subroutines. In assembly language, it is presented as the `call` instruction. The `call` instruction pushes the current value of the PC to a stack data structure in memory, and leaves this value as the return address. Upon completion of the subroutine, this return address is restored to the PC, and program execution resumes with the instruction following the call instruction.

Return instructions pop a return address off the stack and load it into the PC register, thus returning control to the calling routine, i.e., a `ret` instruction.

Among those control flow statements, the jump instruction possesses semantic similarity, in which its adjacent instruction is still a valid instruction but with the opposite semantic. This property can minimize the risk of program crashes introduced by control flow tampering. Fig. 5 illustrates the semantic similarity property of jump instructions. Each node in Fig. 5 presents an instruction (denoted by an opcode of x86-64 ISA), and the adjacent node presents its adjacent instructions in which the Hamming distance is 1. In particular, there is only 1 bit of difference in their opcodes. Meanwhile, those two instructions have the same operands. Note that even if two instructions only have 1 bit of difference in their opcode, they may not be thought of as adjacent. Since, if we exchange their opcodes, it will result in invalid semantics due to a mismatch

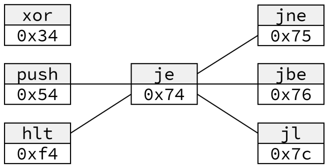


Figure 5: Demonstration of the adjacent instruction.

in the operands format, such as `je` (jump if equal) with 1 operand and `xor` (exclusive or) with 3 operands.

As demonstrated in Fig. 5, the adjacent instructions of the jump instruction `je` (jump if equal) contain its semantic opposite instruction `jne` (jump if not equal). The semantic opposite instruction of `jg` (jump if greater than) also appears in its adjacent instructions (i.e., `jle`), as well as `jc` (jump if carry) and `jnc` (jump if no carry). This observation indicates that a single bit flip in the opcode of branch instructions can switch the semantic of the control flow into the opposite branch. Based on this observation, we selected the conditional jump instructions as the target instruction to compromise the global control logic of the program.

4.1.3 Vulnerable Bit in Opcode

In this subsection, we introduce the process of how the attacker selects the specific bit to be flipped in chosen branch instructions. On the processing architecture, a given machine instruction may specify: i) the opcode (the instruction to be performed) e.g., add, copy, test; or ii) any explicit operands (registers, literal/constant values, addressing modes used to access memory). For readability for programmers, assembly language was designed to use a mnemonic to represent each low-level machine instruction. For each machine instruction, assembly language usually has one corresponding statement. For instance, the instruction `cmp $0x65, %edi` (AT&T format) corresponds to the machine code `83 ff 65`, where `0x83` is the opcode with the corresponding format `CMP r/m32, imm8` (Intel format) under the x86-64 ISA. The machine code `0xff` tells the processor to fetch the first operand from the `EDI` register. `0x65` is an immediate operand.

Flipping any bit in the instructions can dramatically affect the operation results of the instructions. Notably, when the bit flip appears in the opcode, the semantics of the instructions are manipulated. Since the potentially flippable memory cells in the DRAM are sparse, it is hard to find a physical memory page that can flip more than one bit [29, 60]. To minimize the cost of attacking time and the requirement of the hardware constraints, we select the bit in the opcodes as the vulnerable bit to be flipped.

Takeaway 1: In summary, given an ML codebase, we have selected the shared library of the linear algebra backend as the attack module because of its indispensability and wide adoption. We have selected the conditional jump instructions from a shared library (i.e., OpenBLAS), to manipulate the control flow within the program and minimize the risk of program crashes. Finally, we have identified the vulnerable bit to be flipped within the opcode.

The vulnerability of a branch statement, or candidate bit, is measured by two properties: the degree of degradation induced in the DNN model and whether the change generates warnings or crashes. There are 65913 over 843724 (8%) branch instructions within the shared library (i.e., `openblas-0.3.20.so`) that need to be verified and compared according to this measure. To avoid manually analyzing each branch statement, we introduce an automatic and efficient algorithm to identify the most vulnerable code point.

4.2 Automatic Vulnerable Instruction Search

In this subsection, we introduce an LLVM-based **Auto**matic **V**ulnerable **I**nstruction **S**earch algorithm (AutoVIS) that automatically identifies the most vulnerable bit in the linear algebra library of the ML codebase.

LLVM is a set of compiler and toolchain technologies used to turn source code (e.g., C programs) into a language-independent intermediate representation (IR) that serves as a high-level assembly language that can be optimized by LLVM passes. Then the LLVM backend turns the IR into the final machine code. In this process, LLVM passes perform the transformations and optimizations that make up the compiler. In this work, we implement a LLVM pass to analyze the linear algebra library (i.e., `cblas_dgemm` function of OpenBLAS). In particular, the LLVM compiler decomposes the program into different levels of granularity according to hierarchical relationships (function units, basic blocks, and instructions) within the code. In AutoVIS, we traverse each instruction and assign a branch instruction an index (Branch Index). For each branch instruction (i.e., `br`, `switch` and `select` in LLVM IR), we employ the opcode flipping primitive to compromise the control logic of the program.

Specifically, the AutoVIS algorithm was defined by overriding the `llvm::ModulePass::runOnModule` method and registering it as a standard LLVM pass. The AutoVIS algorithm was written in the required format of an LLVM pass and then compiled using `CMake`, resulting in a shared object file (`.so` in Linux). Next, we utilized the optimization tool provided by LLVM, namely `opt`, to dynamically load the shared objective file generated above and modify the IR of the `cblas_dgemm` function according to the AutoVIS algorithm. Finally, the LLVM backend converted the modified IR into the binary code for different ISAs. In this way, for each branch instruction, we generate a re-compiled corrupted version of the `cblas_dgemm` function by invoking our AutoVIS Pass. The details of the AutoVIS are shown in Algorithm 1.

In Algorithm 1, *parseOpcode* translates an instruction into

Algorithm 1: LLVM-based automatic vulnerable bit search algorithm

Input: IR files of compiled ML codebase *Code*, DNN infer infrastructure *network*
Data: Validation dataset *dataset*
Output: Vulnearble code points in code section that can be used to inject fault

1 Vulnerable set $V \leftarrow \{\}$;
2 // Random guess accuracy
3 $acc_{random} \leftarrow 1/numOfClasses(dataset)$;
4 **foreach** *functional unit F in Code* **do**
5   **foreach** *basic block BB in F* **do**
6   **foreach** *instruction Inst in BB* **do**
7   **if** $Inst \in \{\text{br}, \text{switch}, \text{select}\}$ **then**
8                 $opcode \leftarrow parseOpcode(Inst)$;
9                 $instrument(Inst, opcode)$;
10                 $lib \leftarrow generateLibrary()$;
11                 $model \leftarrow buildDeepLearningApp(lib, network)$;
12                 $acc \leftarrow inference(model, dataset)$;
13   **if** $acc \leq acc_{random}$ **then**
14                     $V \leftarrow V \cup \{V\}$;
15   **end**
16                 $recover(Inst, opcode)$;
17   **end**
18   **end**
19   **end**
20 **end**
21 **return** V;

its opcode. The *instrument* function in line 9 queries all adjacent instructions for a given instruction, then returns the instruction with the opposite semantic. After the traversal, the algorithm outputs the vulnerable bit location in the shared library's code section that can be used to inject fault. We then evaluate the prediction accuracy of the DNN inference infrastructure that invokes the modified linear algebra library. By evaluating all of these branch statements, we can find the most vulnerable branch instruction that has the best attack performance (most utility degradation, neither warnings nor crashes). All these actions are performed automatically by LLVM and only take tens of minutes. In contrast, as reported by Hong et al. [29], a weight-based fault injection would take approximately 942 days to identify a single bit corruption in a 138M VGG model on a 488 node high-performance computing cluster.

In AutoVIS, the LLVM compiler plays a critical role, as it enables us to inspect and manipulate instructions. LLVM translates human-readable code (e.g., C) to executable machine instructions. This compiling process can be briefly decomposed into three phases, namely, programming code, intermediate representation (IR), and machine-level instructions. Our LLVM plugin is embedded in the process of translating from IR to instructions. The plugin inspects every incoming IR, filters the conditional branching instructions (a subset of IR opcodes), and modifies the branch by changing the branching condition. The plugin helps us produce modified BLAS libraries in which exactly one branching condition is maliciously modified. The modified library acts as the candidate and is linked to the DNN model. If this modified library achieves the best attack performance (most utility degradation, neither warnings nor crashes), the corresponding branch instruction is chosen as the target conditional branch.

Takeaway 2:
(i) **Executed Offline.** Note that our algorithm is executed offline because the targeted ML codebases are open-sourced. This avoids the time overhead concern arising from the brute-force strategy.
(ii) **Scalability.** AutoVIS is designed to improve the efficiency of the complete analysis of ML Codebases as they scale up. It is also easily extended to other libraries.

5 End-to-End Attack via Rowhammer

By employing the automatic vulnerable instruction search algorithm, the attacker can identify the most effective instruction in the victim ML codebase, as measured by utility degradation and the requirement that the change does not induce warnings or crashes. The attacker now needs to locate the corresponding vulnerable bit at the flippable physical location in the DRAM, and precisely induce the desired bit flip in real hardware systems. Specifically, the attacker first identifies the physical pages that contain flippable memory cells that match the virtual page containing the found vulnerable instruction (i.e., has the same bit offset and flip direction). Then, the attacker employs the memory waylaying technique [24] to relocate the page containing the vulnerable instruction to one of the matched physical locations mentioned above. Finally, the attacker prepares the pattern for the aggressor rows of Rowhammer and frequently accesses these rows to successfully obtain the desired bit flip.

5.1 Offline Memory Profiling

Memory profiling is a process utilized to locate the addresses of flippable bits in the DRAM. This procedure can be executed offline before the victim starts to operate. In particular, we perform double-sided Rowhammer or many-sided Rowhammer on randomly allocated memory and document all the identified bit flips. To enable the hammering technique, the attacker configures its virtual pages in physically consecutive rows in the same bank (sandwich layout) of DRAM chips. Thus, the attacker should crack the virtual-to-physical address translation and decode the DRAM addressing mechanism.

Cracking Virtual-to-Physical Address Translation. The attacker exploits the deterministic behavior of the *buddy allocator* by coercing the kernel to provide physically consecutive memory [14, 37]. Specifically, the attacker keeps requesting small free memory blocks using the `mmap` system call with the

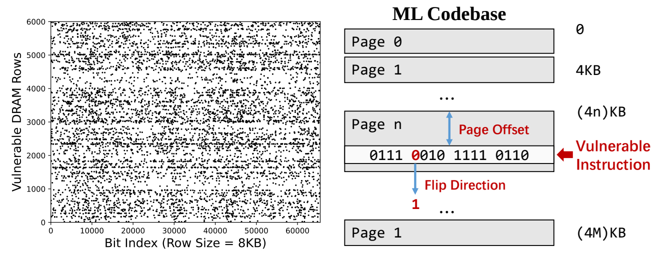
`MAP_POPULATE` flag, until there are less than 2 MiB of free space left in blocks with an order smaller than 10.

If the free space in blocks of order below 10 is less than 2MiB, the attacker sends two 2MiB requests. The kernel satisfies the first request by splitting one of the 10th order blocks (4 MiB in size). Therefore, the second request is fulfilled by a consecutive physical space. The second request's allocated memory has an identical lowest 21 bits in the virtual and physical addresses. Consequently, this block can have the equivalent offset in both virtual and physical address spaces.

Decoding Physical-to-DRAM Address Mapping. Note that the physical address space profiles memory as a continuous large array, which hides the components of the actual physical memory architecture, such as channel, DIMM, rank, bank, row, and column. We are required to identify three physical addresses that are located in three consecutive rows within the same bank of DRAM chips. We exploit a DRAM row buffer timing side channel [51] to identify the pages belonging to the same bank.

When two physical addresses (presented as A and B) are located in the same bank, if we alternatively access them, the time for accessing B is the sum of the row buffer update time (switch from A to B) and the time for reading the row buffer (read B from row buffer). Conversely, when A and B are in different banks, and we alternatively access them, the accessing time for B is only the row buffer reading time, because it is recently accessed. Employing this row buffer timing channel, we can know whether two physical addresses are located in the same bank of DRAM.

Based on the primitives mentioned above, we have adopted the Rowhammer-test tool [2] to profile our DDR3 DRAM chips and TRRespass [20] for DDR4 DRAM chips. The result of memory profiling is a list of bit positions that are flippable in the physical memory (i.e., candidate flippable pages). Each entry records the partial physical address of the byte that contains the flippable bits (because we can only partially translate virtual to physical addresses in a consecutive 2MB block), the bit index within the memory cell and the flipping direction. Fig. 6a demonstrates the bit flip locations in one bank of our DDR3 chip. From the physical address, we can infer the offset of the flippable bits within a page. In a 4KB page, the offset refers to the index of the byte that contains the flippable bits. The offset is used to rule out bit flips that are inconsistent with our attack. As demonstrated in Fig. 6b, the vulnerable page must have the same page offset and flip direction as the flippable page. The offset of the vulnerable page from the starting address of the `cblas_dgemm` function can be gained in the compiling process, where we know to which instruction the target is translated. The attacker can translate the virtual address to a physical address using methods such as side-channel attacks [51]. The Rowhammer attack handles the rest of the attack, mainly including a page relocation and triggering an accurate bit flip.



(a) Offline Memory Profiling. **(b) Page Offset Alignment.**

Figure 6: Demonstration of offline memory profiling and page offset alignment.

5.2 Rowhammer Exploitation

Memory Massage. We exploit the *memory waylaying* [24] technique to relocate the vulnerable virtual page to the matched physical page found in the previous step. Memory waylaying performs on pages in the page cache. Since these cache pages can be evicted at any time, they are not shown in the system's memory utilization and are treated as available memory. After being removed from DRAM, page cache pages are randomly relocated when accessed. Continuous eviction eventually places the vulnerable page on the physical location desired by the attacker. Memory waylaying leverages the prefetch side channel to identify when a virtual page is loaded to predetermined physical locations. Once the data is in the desired location, the intended bit flip can be induced.

Rowhammer Exploitation. Once the vulnerable bit is populated to the flippable bits in DRAM, the attacker starts initializing two aggressor rows to hammer the victim row in the middle. As per the prior research on Rowhammer attacks [68], we use a column-page-stripe pattern to initialize the aggressor rows. Specifically, the attacker duplicates the victim row's bits to two nearby aggressor rows and then sets the stripe pattern for the column that is expected to experience a bit flip. Meanwhile, the remaining bits remain untouched. This allows the attacker to have precise control over the bit flips at targeted locations and prevent simultaneous bit flips at unwanted positions. Once the targeted bit flip is triggered by the attacker, the change instantly takes effect on the victim side. On subsequent accesses, the compromised library in the page cache is continuously provided to the victim.

6 Evaluation

In this section, we provide a comprehensive evaluation of our proposed FrameFlip, to demonstrate its effectiveness and efficiency. Specifically, we report the experimental setting in Section 6.1, evaluate the effectiveness and performance in Section 6.2, and compare FrameFlip with other fault injection attacks in Section 6.3. Lastly, we evaluate the ability of FrameFlip to circumvent the software-based defense method (DeepDyve [42]) in Section 6.4.

6.1 Experimental Setting

Hardware Setup. Our DNN models are trained and analyzed on the Nvidia Titan RTX GPU platform. The GPU operates at a clock speed of 1350MHz with 24GB of dedicated memory. The trained model is deployed on a testbed machine where our proposed attack is evaluated. The inference service runs on a Comet Lake-based Intel i7-10700 CPU. As for memory configuration, the testbed machine possesses an 8 GB Apacer DDR4 SDRAM memory module. For this module, TRRespass has been executed and reports that double-sided Rowhammer is effective in inducing bit flips, i.e., many-sided Rowhammer becomes double-sided Rowhammer.

Datasets. We evaluate FrameFlip on four widely used datasets, including FMNIST, CIFAR-10, GTSRB, and ImageNet. FMNIST [65] consists of 28×28 grayscale images, associated with a label from 10 fashion classes. The training set has $60,000$ examples and the test set has $10,000$ examples. The CIFAR-10 dataset [36] consists of $60,000$ 32×32 colour images in 10 classes, with $6,000$ images for each class; so there are $50,000$ training images and $10,000$ test images. The German Traffic Sign Recognition Benchmark (GTSRB) [30] contains 43 classes, split into $39,209$ training images and $12,630$ test images. For the ImageNet dataset, we use its ILSVRC-2012 subset [56] containing $1,281,167$ colored training images and $50,000$ evaluation images of size 224×224 coming from 1000 classes. These datasets are widely used by previous related works [29, 53, 68].

Models. For different datasets, we choose four popular network architectures that are widely used for image classification tasks, including VGG-16 [58], ResNet-34, ResNet-50 [27] and LeNet [38]. For CIFAR-10, GTSRB, ImageNet datasets, we adopt three network architectures (VGG-16, ResNet-34, ResNet-50) to perform the classification tasks. In addition, we use LeNet-5 to learn models on the FMNIST dataset. Thus, we have 10 groups of configurations that cover tasks of varying difficulty.

Metrics. The objective of FrameFlip is to degrade the prediction accuracy of DNN models. A lower prediction accuracy after attacking (noted as $ACC_{corrupted}$), indicates better attack performance of FrameFlip. The baseline of attack performance is random guess prediction accuracy defined as $ACC_{random} = 1/CLASS(D)$, where $CLASS(D)$ is the number of classes of the dataset D.

The issue with using $ACC_{corrupted}$ to define the attack performance is that $ACC_{corrupted}$ is influenced by the DNN model's original prediction ability. An untrained DNN model may also achieve an extremely low $ACC_{corrupted}$ after an attack, making it difficult to properly measure the attack performance. So we define the relative prediction loss as the attack performance metric: $RPL = (ACC_{pristine} - ACC_{corrupted})/ACC_{pristine}$, where $ACC_{pristine}$ presents DNN model's prediction accuracy before the attack. Under this measurement, RPL equal to 0.00% means the attack has no effect on the performance of DNN models. While RPL equal to 100.00% means the DNN model loses its prediction ability after the attack.

Table 2: The attack performance of FrameFlip on multiple datasets and network architectures.

Dataset	Network	Prediction Accuracy (%)			RPL (%)
		Before Attack	After Attack	Random Guess	
ImageNet	VGG-16	71.59	0.00	0.10	100.00
	ResNet-34	73.30	0.00		100.00
	ResNet-50	76.15	0.00		100.00
GTSRB	VGG-16	92.36	0.71	2.33	99.23
	ResNet-34	95.14	0.71		99.25
	ResNet-50	94.67	1.19		98.75
CIFAR-10	VGG-16	92.48	10.00	10.00	89.19
	ResNet-34	93.44	9.47		89.87
	ResNet-50	93.58	10.00		89.31
FMNIST	LeNet	88.91	6.53	10.00	92.66

6.2 Attack Performance

Tab. 2 demonstrates the attack performance of FrameFlip on 4 datasets and 4 network architectures. The RPL denotes the relative prediction loss of the model after the attack. A higher RPL corresponds to better attack performance. As shown in Tab. 2, for ImageNet dataset, the average prediction is 73.68% across three network architectures (VGG-16, ResNet-34 and ResNet-50) before the attack. After the attack, the prediction accuracy of all three models degrades to 0.00%, and the observed RPL is 100.00%. The experimental results on ImageNet show that FrameFlip completely degrades the prediction utility of three dominant DNN network architectures. The 0% accuracy is a natural outcome of the fact that bit flip actually incurs a deterministic change of behavior to the branching instruction. With the property of "deterministic", by triggering a bit flip, a conditional branch, which is originally evaluated as `TRUE`, will always be flagged as `False` with the same set of inputs. As a result, the model behavior after the attack is not random, but deterministic instead. Therefore, it is very likely that for a highly accurate model, the attacked model always produces a false classification, leading to a 0% accuracy. For the GTSRB dataset, the average prediction is 94.06% across three network architectures (VGG-16, ResNet-34 and ResNet-50) prior to the attack. After the attack, the average prediction accuracy of all three models is reduced to 0.87%, and the observed RPL is 99.08%. The attack performance is close to the upper bound (100.00% RPL). For CIFAR-10 dataset, the average prediction is 93.17% across three network architectures (VGG-16, ResNet-34 and ResNet-50) before the attack. After the attack, the average prediction accuracy of all three models degrades to 9.82%, and the average RPL reaches 89.46%. For FMNIST dataset, the prediction

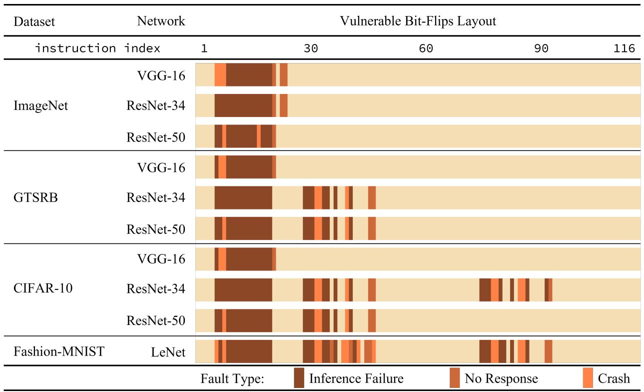
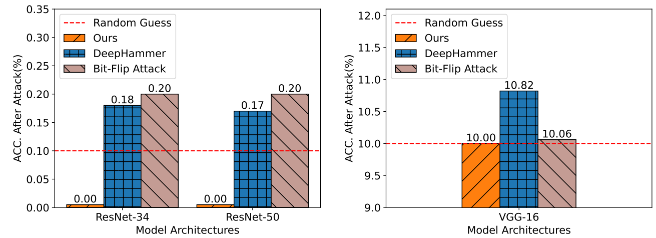


Figure 7: The severity of 116 branch instructions on 10 DNN-based classification tasks.



(a) ImageNet **(b) CIFAR-10**

Figure 8: Comparison with existing works on attack performance.

is 88.91% before the attack. After the attack, the prediction accuracy degrades to 6.53%, and the RPL is 92.66%. Compared with ImageNet, GTSRB and FMNIST datasets, the attacker performance on CIFAR-10 is lower but still achieves lower prediction accuracy than the random guess.

In addition, FrameFlip achieves better attack performance on tasks that have more classes, i.e., the average RPLs for ImageNet (1000 classes), GTSRB(43 classes), CIFAR-10(10 classes) datasets are 100.00%, 99.08% and 89.10%, respectively. In summary, the results shown in Tab. 2 manifest the effectiveness of FrameFlip on a variety of datasets and network architectures that cover tasks of varied difficulty.

6.2.1 Transferability

Recall that AutoVIS finds the vulnerable `branch` instruction in a compiled `cblas_dgemm` of the OpenBLAS library. These vulnerable instructions are defined as attack code points. The attack transferability of these code points means that code points found in one attack instance (ImageNet dataset and ResNet-34 network) have comparable attack performance in other attack instances.

Fig. 7 reports the severity of 116 branch instructions on 10 DNN-based classification tasks. A strip in each row represents a branch instruction, the color of this strip represents its severity for the DNN model's prediction accuracy when this instruction is flipped. A deeper color indicates a more significant degradation in model accuracy. A shallow color indicates that the instruction does not have an effect on the corresponding DNN model's prediction accuracy. As we can see from Fig. 7, those attack code points that have incurred dramatic utility degradation (see deeper colored strips in Fig. 7) are general and transferable, which indicates we can always choose those attack points to be flipped and achieve a satisfying attack performance on different datasets and networks.

Tab. 3 further shows details about the severity of those attack points on 10 classification tasks. The transferability of those attack points reveals that there exist some universal vulnerable instructions that are datasets- and networks-agnostic and potentially exploited by fault injections.

6.3 Comparison with Existing Work

In this subsection, we compare the attack performance and attack cost with existing work. Bit-Flip attack [53] and DeepHammer [68] are two comparable fault injections attacks against the DNN model's weight parameters via Rowhammer. In these works, the attacker searches for vulnerable bits in the DNN model's weight parameters. When those vulnerable bits are flipped by Rowhammer, the prediction accuracy the prediction accuracy of the model degrades. Their attack objectives are the same as ours, i.e., degrading the prediction accuracy of DNN models. So we compare our work with these two DNN fault injection attacks.

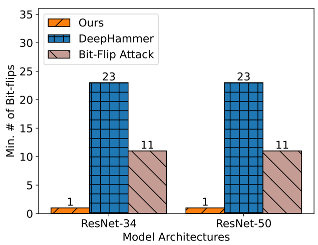
For comparing the attack performance, we evaluate the prediction accuracy of the three different attacks on two datasets and three network architectures. The results are presented in Fig. 8. As shown in Fig. 8, the red dash lines are the baseline (i.e., random guess prediction accuracy). For ImageNet and CIFAR-10 datasets, they are 0.1% and 10% respectively. The results shown in Fig. 8a show that the prediction accuracy after Bit-Flip attacks and DeepHammer is still higher than the random guess on the ImageNet dataset. On the contrary, the prediction accuracy after our FrameFlip attack is significantly lower than the random guess. For the CIFAR-10 dataset, all three attacks induce a similar prediction accuracy. Note that FrameFlip maintains the best attack performance. The results demonstrated in Fig. 8 reveal that FrameFlip exhibits better attack performance than DeepHammer and Bit-Flip attack.

Consider that flipping a vulnerable bit by Rowhammer is time-consuming. In addition, there are some hardware constraints caused by the memory layout of hardware specifications when flipping multiple bits simultaneously. Thus, we further compare the attack cost of the three attacks. The attack cost is defined as the number of bits that need to be flipped in a given attack.

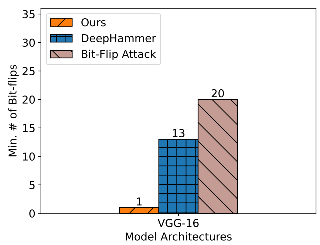
Fig. 9 demonstrates the minimum number of bits required to be flipped by the three attacks. FrameFlip always needs 1 bit to be flipped when conducting its attack. In comparison, for the ImageNet dataset, DeepHammer needs 23 bits flipped simultaneously to complete an attack. The number for the Bit-Flip attack is 11. For the CIFAR-10 dataset, Bit-Flip attack

Table 3: The vulnerable instructions that reach top-3 attack performance on different datasets and networks.

Datasets	Networks	Top-1 PRL(%)	Instructions	Top-2 PRL(%)	Instructions	Top-3 PRL(%)	Instructions
ImageNet	VGG-16	100.00	8~19	-	-	-	-
	ResNet-34	100.00	5~8, 10 ~19	99.65	9	-	-
	ResNet-50	100.00	5, 8~15, 17 ~19	99.80	6	-	-
GTSRB	VGG-16	99.23	5, 10~19	94.60	9	94.08	8
	ResNet-34	99.25	5, 10~19	96.14	6	94.79	7
	ResNet-50	98.75	5, 10~19	98.60	6	94.73	8, 9
CIFAR-10	VGG-16	89.19	5, 8~19	-	-	-	-
	ResNet-34	89.32	30	88.73	5, 7, 8, 10~19, 74, 75, 79, 82	79.92	6
	ResNet-50	88.80	5, 8~19	87.29	30	72.24	34
FMNIST	LeNet	92.66	6	88.75	8~19, 28, 29, 33, 34, 36, 74, 75, 79, 82	88.70	30



(a) ImageNet



(b) CIFAR-10

Figure 9: The minimum number of bit-flips required by three attacks.

and DeepHammer need 20 and 13 bits to be flipped respectively. Thus, FrameFlip incurs the lowest attack cost. Note that the attacks being compared have not yet been demonstrated through end-to-end implementations. This is potentially due to the difficulty in concurrently flipping their multiple required bits through Rowhammer.

6.4 Existing Fault Injection Defense

In this subsection, we evaluate the effectiveness of FrameFlip in circumventing existing defense techniques that aim to protect DNN models against fault injection attacks. A popular software-based fault injection defense technique is DeepDyve [42]. This method involves utilizing a simplified and smaller DNN (known as the checker DNN) to approximate the output of the original complicated DNN model (known as the task DNN). Subsequently, the approach verifies the consistency of the outputs between the two models in an end-to-end manner. If the outputs of the checker model and the task model do not match, re-computation on the task DNN is performed for potential fault recovery.

DeepDyve has three metrics to measure the detection performance for fault injections. These include i) the false positive rate (FPR), ii) the false negative rate (FNR), and iii) the fault coverage (FC). FC denotes the rate of detected faults against all faults. We evaluate the effectiveness of FrameFlip against DeepDvye. The results are reported in Tab. 4. The dataset and network are GTSRB and ResNet-34 respectively.

DeepDyve first creates a simple checker model based on the original complicated DNN task model. Then, it compares the inference consistency of both models for attack detection. As shown in Tab. 4, the FPR of DeepDyve against our FrameFlip is up to 99.42%, which reveals that DeepDyve outputs almost all predictions as the faults and thus is not capable of defending our fault injection attack. The reason is that FrameFlip targets the code level, and simultaneously corrupts both the task model and the checker model. As such, the outputs of the two models are unlikely to be consistent, resulting in a high FPR of DeepDyve. This shows that DeepDyve has lost its functionality under FrameFlip, as the checker model can no longer verify the output of the task model, making DeepDyve is ineffective against FrameFlip.

Table 4: The defense performance of DeepDyve.

Fault Injections	FPR (%)	FNR (%)	FC (%)
Ours	99.42	4.48	95.52
Bit-Flip Attack	0.00	0.79	99.21
Random Fault Attack	0.00	5.07	94.93

7 Discussion

7.1 Countermeasures

In this section, we investigate the mitigation strategies from an attack chain perspective.

Mitigating Rowhammer. A number of countermeasures have been proposed to mitigate Rowhammer attacks including hardware-based, software-based, and software-hardware co-design approaches [73]. Hardware-based defenses [46, 47, 49] necessitate modifications to the underlying hardware, including the memory controller and/or DRAM, making them unable to be backported. In contrast, software-based defenses [4, 10, 72] mainly exploit the specific characteristics of Rowhammer to detect associated attacks. Therefore, they are compatible with legacy DRAM modules. In recent hardware-software co-design defense [34], DRAM and memory controller are modified to detect bit flips, and the proposed instruction-set extension is used to correct flipped bits by OS.
Thwarting Memory Waylaying. Memory waylaying allows the attacker to precisely manipulate the memory allocation. This technique is crucial for an attacker to mount our attack.

To thwart this primitive, the OS can monitor the abnormal activities of page-cache pages and restrict the allocation of excessive page cache pages in a single process [24].

Protecting critical instructions. Selective protection of the most vulnerable bits through system software can effectively mitigate our attack. Specifically, by utilizing the proposed AutoVIS, the vulnerable instructions that significantly degrade DNN accuracy can be identified and subsequently protected by a secure enclave (e.g., Intel SGX [18, 24]). Additionally, compilers could generate code for those critical instructions that ensure an attacker needs a minimum of N bit flips to manipulate the control flow successfully [7, 13, 24]).

7.2 Generality of AutoVIS

In the offline phase, AutoVIS employs the opcode flipping primitive to modify each branch logic of a function in the linear algebra library and then records the inference accuracy of a DNN that uses a modified library. To investigate its generality, we choose FBGEMM (Facebook GEneral Matrix Multiplication), a library that specializes in low-precision and high-performance matrix-matrix multiplications for server-side inference, to serve as a backend of PyTorch on x86 machines. In particular, the function `fbgemmPacked` in FBGEMM is a good candidate, as it provides the same functionality as the aforementioned `cblas_dgemm` function from the OpenBLAS library, i.e., both provide low-level matrix-matrix multiplication that has been optimized to fit for modern CPU cache hierarchy [23]. The most vulnerable bit within the `fbgemmPacked` function can be identified by AutoVIS, which we believe can achieve significant accuracy degradation.

7.3 Limitations

FrameFlip is currently applicable to models executed on CPUs and UNIX/Linux systems. For other BLAS operations on a GPU (e.g., by utilizing cuBLAS), the feasibility of our approach has not been explored, as the Rowhammer exploitation relies on Rowhammer bugs (specific to DRAM) and memory waylaying. Particularly, memory waylaying exploits OS's memory management feature, i.e., page-cache, making itself specific to general CPUs. Thus, FrameFlip affects CPU-based inference that is supported by cloud providers such as Amazon, Google and Alibaba Clouds.

Static linking links related library functions directly into a victim's binary, and thus the linked functions cannot be called by other binaries. Consequently, FrameFlip is unlikely to find the exact physical locations of vulnerable bits in library functions and instead might cause the victim to crash. However, static linking generates large-size binaries and dynamic linking produces small-size binaries. Thus, developers are likely to use dynamic linking in practice to distribute their models.

Besides the accuracy depletion of classification models, FrameFlip can be used to flip other critical bits (e.g., those that control the number of iterations), resulting in significantly prolonged execution time for DNN inference. In addition to the image classification, we will also explore other tasks in the future work, such as object detection and natural language processing (NLP).

8 Related Work

In the context of DNNs, fault injections are used to directly tamper with the DNN inference process. Primarily, compromised DNN weight parameters result in two types of attack: utility degradation and Trojan attacks. For utility degradation, Liu et al. [44] present a simulated fault attack that is aimed at disrupting DNN prediction by flipping model bias parameters. DeepLaser [11] demonstrates a laser-based fault injection technique that hijacks DNN activation function. Hong et al. [29] conduct bit flip attacks against various model parameters in full-precision DNN models. As for Trojan attacks, Rakin et al. [54] insert a targeted Trojan into a DNN through the bit-flip. However, this work does not consider realistic restrictions in hardware. Tol et al. [60] and Chen et al. [12] have proposed incorporating hardware specifications as constraints during trigger pattern generation and backdoor injection.

To defend against weight-based fault injections in the context of DNNs, several works have been proposed. Aegis [62] is a multi-exit mechanism that allows input samples to exit early from different layers of DNN models in order to disrupt the attackers' plans. He et al. [28] adopt binarization-aware training to defend against bit-flipping attacks. Li et al. [39] propose a checksum-based detection technique during model inference. During inference, the checksum of the weights is validated against the original signatures.

Nevertheless, all the above attacks and defenses focus on the weight parameters of DNNs instead of other software and hardware platforms of ML services. Bagdasaryan et al. [5] compromise ML training code before the training starts. Clifford et al. [16] insert imperceptible backdoors by a malicious compiler during compilation. These studies reveal that ensuring the security of ML models necessitates examining all components within the pipeline including the data, model architecture, compiler, and hardware specification.

9 Conclusion

We have proposed FrameFlip, a novel hardware-based fault injection attack on machine learning (ML) codebases that can universally and significantly reduce DNN models' prediction accuracy to a random guess level. The impact of this work is the search for vulnerable instructions in ML codebases, highlighting the importance of increased attention to security analysis for ML codebases. Extensive experiments have demonstrated the pronounced susceptibility of ML codebases to malicious bit-flips even with a single bit fault.

Acknowledgments

The work has been supported in part by the National Natural Science Foundation of China (Grant No. 62132013, 62325207, 62002167, 62201311), the Peng Cheng Laboratory Major Key Project (Grant No. PCL2023AS1-5, PCL2021A09-B2), the China National Postdoctoral Program for Innovative Talents, and the China Postdoctoral Science Foundation (Grant No. 2022M721736).

Availability. The relevant code is publicly available at https://github.com/FrameFlip/SGXBLAS.

References

[1] Apache mxnet: A flexible and efficient library for deep learning.

[2] Program for testing for the dram "rowhammer" problem using eviction.

[3] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, et al. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467*, 2016.

[4] Zelalem Birhanu Aweke, Salessawi Ferede Yitbarek, Rui Qiao, Reetuparna Das, Matthew Hicks, Yossi Oren, and Todd M. Austin. ANVIL: software-based protection against next-generation rowhammer attacks. In *Proc. of ASPLOS*, pages 743–755, 2016.

[5] Eugene Bagdasaryan and Vitaly Shmatikov. Blind backdoors in deep learning models. In *Proc. of USENIX Security*, 2021.

[6] Jiawang Bai, Baoyuan Wu, Yong Zhang, Yiming Li, Zhifeng Li, and Shu-Tao Xia. Targeted attack against deep neural networks via flipping limited weight bits. In *Proc. of ICLR*, 2021.

[7] Thierno Barry, Damien Couroussé, and Bruno Robisson. Compilation of a countermeasure against instruction-skip fault attacks. In *Proc. of CS2@HiPEAC*, pages 1–6, 2016.

[8] Robert C Baumann. Radiation-induced soft errors in advanced semiconductor technologies. *IEEE Transactions on Device and Materials Reliability*, 5(3):305–316, 2005.

[9] Erik Bosman, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Dedup est machina: Memory deduplication as an advanced exploitation vector. In *Proc. IEEE S&P*, 2016.

[10] Ferdinand Brasser, Lucas Davi, David Gens, Christopher Liebchen, and Ahmad-Reza Sadeghi. Can't touch this: Software-only mitigation against rowhammer attacks targeting kernel memory. In Engin Kirda and Thomas Ristenpart, editors, *USENIX Security Symposium*, pages 117–130, 2017.

[11] Jakub Breier, Xiaolu Hou, Dirmanto Jap, Lei Ma, Shivam Bhasin, and Yang Liu. Practical fault attack on deep neural networks. In *Proc. of CCS*, 2018.

[12] Huili Chen, Cheng Fu, Jishen Zhao, and Farinaz Koushanfar. Proflip: Targeted trojan attack with progressive bit flips. In *Proc. of IEEE ICCV*, 2021.

[13] Zhi Chen, Junjie Shen, Alex Nicolau, Alexander V. Veidenbaum, Nahid Farhady Ghalaty, and Rosario Cammarota. CAMFAS: A compiler approach to mitigate fault attacks via enhanced simdization. In *Proc. of FDTC*, pages 57–64, 2017.

[14] Yueqiang Cheng, Zhi Zhang, Surya Nepal, and Zhi Wang. Cattmew: Defeating software-only physical kernel isolation. *IEEE Trans. Dependable Secur. Comput.*, 18(2):605–622, 2021.

[15] Edward Chou, Florian Tramèr, and Giancarlo Pellegrino. Sentinet: Detecting localized universal attacks against deep learning systems. In *Proc. IEEE S&P Workshops*, 2020.

[16] Tim Clifford, Ilia Shumailov, Yiren Zhao, Ross J. Anderson, and Robert Mullins. Impnet: Imperceptible and blackbox-undetectable backdoors in compiled neural networks. *arXiv preprint:2210.00108*, 2022.

[17] Lucian Cojocar, Jeremie Kim, Minesh Patel, Lillian Tsai, Stefan Saroiu, Alec Wolman, and Onur Mutlu. Are we susceptible to rowhammer? an end-to-end methodology for cloud providers. In *Proc. IEEE S&P*, 2020.

[18] Victor Costan and Srinivas Devadas. Intel SGX explained. *IACR Cryptol. ePrint Arch.*, page 86, 2016.

[19] Tian Dong, Shaofeng Li, Guoxing Chen, Minhui Xue, Haojin Zhu, and Zhen Liu. RAI2: responsible identity audit governing the artificial intelligence. In *Proc. of NDSS*, 2023.

[20] Pietro Frigo, Emanuele Vannacc, Hasan Hassan, Victor Van Der Veen, Onur Mutlu, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. TRRespass: Exploiting the many sides of target row refresh. In *Proc. IEEE S&P*, 2020.

[21] Yansong Gao, Bao Gia Doan, Zhi Zhang, Siqi Ma, Jiliang Zhang, Anmin Fu, Surya Nepal, and Hyoungshick Kim. Backdoor attacks and countermeasures on deep

learning: A comprehensive review. *arXiv preprint arXiv:2007.10760*, 2020.

[22] Yansong Gao, Huming Qiu, Zhi Zhang, Binghui Wang, Hua Ma, Alsharif Abuadbba, Minhui Xue, Anmin Fu, and Surya Nepal. Deeptheft: Stealing dnn model architectures through power side channel. In *Proc. IEEE S&P*, 2024.

[23] Kazushige Goto and Robert A. van de Geijn. Anatomy of high-performance matrix multiplication. *ACM Trans. Math. Softw.*, 34(3):12:1–12:25, 2008.

[24] Daniel Gruss, Moritz Lipp, Michael Schwarz, Daniel Genkin, Jonas Juffinger, Sioli O'Connell, Wolfgang Schoechl, and Yuval Yarom. Another flip in the wall of rowhammer defenses. In *Proc. IEEE S&P*, 2018.

[25] Gaël Guennebaud, Benoît Jacob, et al. Eigen v3, 2010.

[26] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proc. of IEEE ICCV*, 2015.

[27] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proc. of IEEE CVPR*, 2016.

[28] Zhezhi He, Adnan Siraj Rakin, Jingtao Li, Chaitali Chakrabarti, and Deliang Fan. Defending and harnessing the bit-flip based adversarial weight attack. In *Proc. of IEEE CVPR*, 2020.

[29] Sanghyun Hong, Pietro Frigo, Yigitcan Kaya, Cristiano Giuffrida, and Tudor Dumitras. Terminal Brain Damage: Exposing the graceless degradation in deep neural networks under hardware fault attacks. In *Proc. of USENIX Security*, 2019.

[30] Sebastian Houben, Johannes Stallkamp, Jan Salmen, Marc Schlipsing, and Christian Igel. Detection of traffic signs in real-world images: The german traffic sign detection benchmark. In *Proc. of IJCNN*, 2013.

[31] Yupeng Hu, Wenxin Kuang, Zheng Qin, Kenli Li, Jiliang Zhang, Yansong Gao, Wenjia Li, and Keqin Li. Artificial intelligence security: Threats and countermeasures. *ACM Computing Surveys (CSUR)*, 55(1):1–36, 2021.

[32] Matthew Jagielski, Alina Oprea, Battista Biggio, Chang Liu, Cristina Nita-Rotaru, and Bo Li. Manipulating machine learning: Poisoning attacks and countermeasures for regression learning. In *Proc. IEEE S&P*, 2018.

[33] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. *arXiv preprint arXiv:1408.5093*, 2014.

[34] Jonas Juffinger, Lukas Lamster, Andreas Kogler, Maria Eichlseder, Moritz Lipp, and Daniel Gruss. Csi:rowhammer - cryptographic security and integrity against rowhammer. In *Proc. IEEE S&P*, 2023.

[35] Yoongu Kim, Ross Daly, Jeremie S. Kim, Chris Fallin, Ji-Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors. In *Proc. of ISCA*, 2014.

[36] Alex Krizhevsky, Geoffrey Hinton, et al. Learning multiple layers of features from tiny images. 2009.

[37] Andrew Kwong, Daniel Genkin, Daniel Gruss, and Yuval Yarom. RAMBleed: Reading bits in memory without accessing them. In *Proc. IEEE S&P*, 2020.

[38] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proc. IEEE*, 86(11):2278–2324, 1998.

[39] Jingtao Li, Adnan Siraj Rakin, Zhezhi He, Deliang Fan, and Chaitali Chakrabarti. RADAR: run-time adversarial weight attack detection and accuracy recovery. In *Proc. of DATE*, 2021.

[40] Jingtao Li, Adnan Siraj Rakin, Yan Xiong, Liangliang Chang, Zhezhi He, Deliang Fan, and Chaitali Chakrabarti. Defending bit-flip attack through DNN weight reconstruction. In *Proc. of DAC*, 2020.

[41] Shaofeng Li, Minhui Xue, Benjamin Zi Hao Zhao, Haojin Zhu, and Xinpeng Zhang. Invisible backdoor attacks on deep neural networks via steganography and regularization. *IEEE Trans. Dependable Secur. Comput.*, 18(5):2088–2105, 2021.

[42] Yu Li, Min Li, Bo Luo, Ye Tian, and Qiang Xu. Deepdyve: Dynamic verification for deep neural networks. In *Proc. of CCS*, 2020.

[43] Qi Liu, Wujie Wen, and Yanzhi Wang. Concurrent weight encoding-based detection for bit-flip attack on neural network accelerators. In *Proc. of ICCAD*, 2020.

[44] Yannan Liu, Lingxiao Wei, Bo Luo, and Qiang Xu. Fault injection attack on deep neural network. In *Proc. of ICCAD*, 2017.

[45] Yingqi Liu, Shiqing Ma, Yousra Aafer, Wen-Chuan Lee, Juan Zhai, Weihang Wang, and Xiangyu Zhang. Trojaning attack on neural networks. In *Proc. of NDSS*, 2018.

[46] Michele Marazzi, Patrick Jattke, Flavien Solt, and Kaveh Razavi. Protrr: Principled yet optimal in-dram target row refresh. In *Proc. IEEE S&P*, pages 735–753, 2022.

[47] Michele Marazzi, Flavien Solt, Patrick Jattke, Kubo Takashi, and Kaveh Razavi. REGA: scalable rowhammer mitigation with refresh-generating activations. In *Proc. IEEE S&P*, pages 1684–1701, 2023.

[48] Janani Mukundan, Hillery C. Hunter, Kyu-hyoun Kim, Jeffrey Stuecheli, and José F. Martínez. Understanding and mitigating refresh overheads in high-density DDR4 DRAM systems. In *Proc. of ISCA*, 2013.

[49] Yeonhong Park, Woosuk Kwon, Eojin Lee, Tae Jun Ham, Jung Ho Ahn, and Jae W. Lee. Graphene: Strong yet lightweight row hammer protection. In *Proc. IEEE/ACM MICRO*, 2020.

[50] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. In *Proc. of NeurIPS*, 2019.

[51] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. DRAMA: exploiting DRAM addressing for cross-cpu attacks. In *Proc. of USENIX Security*, 2016.

[52] Adnan Siraj Rakin, Md Hafizul Islam Chowdhuryy, Fan Yao, and Deliang Fan. Deepsteal: Advanced model extractions leveraging efficient weight stealing in memories. In *Proc. IEEE S&P*, 2022.

[53] Adnan Siraj Rakin, Zhezhi He, and Deliang Fan. Bit-flip attack: Crushing neural network with progressive bit search. In *Proc. of IEEE ICCV*, 2019.

[54] Adnan Siraj Rakin, Zhezhi He, and Deliang Fan. TBT: targeted neural network attack with bit trojan. In *Proc. of IEEE CVPR*, 2020.

[55] Kaveh Razavi, Ben Gras, Erik Bosman, Bart Preneel, Cristiano Giuffrida, and Herbert Bos. Flip feng shui: Hammering a needle in the software stack. In *Proc. of USENIX Security*, 2016.

[56] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael S. Bernstein, Alexander C. Berg, and Fei-Fei Li. Imagenet large scale visual recognition challenge. *International Journal of Computer Vision*, 115(3):211–252, 2015.

[57] Ahmed Salem, Apratim Bhattacharya, Michael Backes, Mario Fritz, and Yang Zhang. Updates-leak: Data set inference and reconstruction attacks in online learning. In *Proc. of USENIX Security*, 2020.

[58] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. In *Proc. of ICLR*, 2015.

[59] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, and Rob Fergus. Intriguing properties of neural networks. *arXiv preprint arXiv:1312.6199*, 2013.

[60] M Caner Tol, Saad Islam, Berk Sunar, and Ziming Zhang. Don't knock! Rowhammer at the backdoor of DNN models. In *Proc. of DSN*, 2023.

[61] Endong Wang, Qing Zhang, Bo Shen, Guangyong Zhang, Xiaowei Lu, Qing Wu, and Yajuan Wang. Intel math kernel library. In *High-Performance Computing on the Intel® Xeon Phi™*, pages 167–188. Springer, 2014.

[62] Jialai Wang, Ziyuan Zhang, Meiqi Wang, Han Qiu, Tianwei Zhang, Qi Li, Zongpeng Li, Tao Wei, and Chao Zhang. Aegis: Mitigating targeted bit-flip attacks against deep neural networks. In *Proc. of USENIX Security*, 2023.

[63] Minghua Wang, Zhi Zhang, Yueqiang Cheng, and Surya Nepal. Dramdig: a knowledge-assisted tool to uncover dram address mapping. In *Proc. of DAC*, 2020.

[64] Zhang Xianyi, Wang Qian, and Zhang Yunquan. Model-driven level 3 blas performance optimization on loongson 3a processor. In *Proc. of ICPADS*, 2012.

[65] Han Xiao, Kashif Rasul, and Roland Vollgraf. Fashion-MNIST: a novel image dataset for benchmarking machine learning algorithms, 2017.

[66] Yuan Xiao, Xiaokuan Zhang, Yinqian Zhang, and Radu Teodorescu. One bit flips, one cloud flops: Cross-vm row hammer attacks and privilege escalation. In *Proc. of USENIX Security*, 2016.

[67] Mengjia Yan, Christopher W. Fletcher, and Josep Torrellas. Cache telepathy: Leveraging shared resource attacks to learn DNN architectures. In *In Proc. of USENIX Security*, pages 2003–2020, 2020.

[68] Fan Yao, Adnan Siraj Rakin, and Deliang Fan. Deephammer: Depleting the intelligence of deep neural networks through targeted chain of bit flips. In *Proc. of USENIX Security*, 2020.

[69] Yuanshun Yao, Huiying Li, Haitao Zheng, and Ben Y. Zhao. Latent backdoor attacks on deep neural networks. In *Proc. of CCS*, 2019.

[70] Zhenlong Yuan, Yongqiang Lu, Zhaoguo Wang, and Yibo Xue. Droid-sec: deep learning in android malware detection. In *Proc. of SIGCOMM*, 2014.

[71] Weiting Zhang, Dong Yang, Wen Wu, Haixia Peng, Ning Zhang, Hongke Zhang, and Xuemin Shen. Optimizing

federated learning in distributed industrial IoT: A multi-agent approach. *IEEE Journal on Selected Areas in Communications*, 39(12):3688–3703, Dec. 2021.

[72] Zhi Zhang, Yueqiang Cheng, Minghua Wang, Wei He, Wenhao Wang, Surya Nepal, Yansong Gao, Kang Li, Zhe Wang, and Chenggang Wu. Softtrr: Protect page tables against rowhammer attacks using software-only target row refresh. In *Proc. of USENIX ATC*, pages 399–414, 2022.

[73] Zhi Zhang, Jiahao Qi, Yueqiang Cheng, Shijie Jiang, Yiyang Lin, Yansong Gao, Surya Nepal, Yi Zou, Jiliang Zhang, and Yang Xiang. A retrospective and future-spective of rowhammer attacks and defenses on DRAM. *arXiv preprint arXiv:2201.02986*, 2022.