OBSAN: An Out-Of-Bound Sanitizer to Harden DNN Executables

Yanzuo Chen, Yuanyuan Yuan*, Shuai Wang*
The Hong Kong University of Science and Technology
{ychenjo,yyuanaq,shuaiw}@cse.ust.hk

***Abstract*—The rapid adoption of deep neural network (DNN) models on a variety of hardware platforms has boosted the development of deep learning (DL) compilers. DL compilers take as input the high-level DNN model specifications and generate optimized DNN executables for diverse hardware architectures like CPUs and GPUs. Despite the emerging adoption of DL compilers in real-world scenarios, no solutions exist to protect DNN executables. To fill this critical gap, this paper introduces OBSAN, a fast sanitizer designed to check for out-of-bound (OOB) behavior in DNN executables. Holistically, DNN incorporates *bidirectional computation*: forward propagation which predicts an output based on an input, and backward propagation which characterizes how the forward prediction is made. Both the neuron activations in forward propagation and gradients in backward propagation should fall within valid ranges, and deviations from these ranges would be considered as OOB.**

OOB is primarily related to unsafe behavior of DNNs, which root from anomalous inputs and may cause mispredictions or even exploitation via adversarial examples (AEs). We thus design OBSAN, which includes two variants, FOBSAN and BOBSAN, to detect OOB in forward and backward propagations, respectively. Each OBSAN variant is designed as extra passes of DL compilers to integrate with large-scale DNN models, and we design various optimization schemes to reduce the overhead of OBSAN. Evaluations over various anomalous inputs show that OBSAN manifests promising OOB detectability with low overhead. We further present two downstream applications to show how OBSAN prevents online AE generation and facilitates feedback-driven fuzz testing toward DNN executables.

I. INTRODUCTION

With the emerging demand to use deep learning (DL) techniques in real-world scenarios, recent years have witnessed a tremendous trend in deploying DL models on a wide spectrum of computing platforms ranging from cloud servers to mobile phones and embedded devices. To handle complex deployment environments and explore the full potential of computing platforms for optimization, one promising opportunity is to employ *DL compilers* [19, 47, 68]. DL compilers take a high-level DNN model specification as input and generate corresponding low-level optimized binary code for a diverse set of hardware backends. To date, many suppliers of edge devices and low-power processors are incorporating DL compilers into their design and application to reap the benefits of compiled DNN models [31, 56, 57, 65, 93]. Cloud service providers such as Amazon and Google are also including DL compilers in their DL services to boost performance [9, 90].

Despite the prosperous use of DL compilers, techniques to harden DNN executables do not exist. While recent works have proposed techniques to validate the execution legitimacy of DNN models, detect exploitations and AEs, or launch security fuzz testing [21–23, 58, 80, 86, 102, 102], those approaches are essentially for DNN models running on DL frameworks like TensorFlow and PyTorch [7, 60]. Recent works test DL frameworks or DL compilers [30, 42, 62, 63, 71, 83, 89, 91, 104]. Nevertheless, to the best of our knowledge, no analysis or security hardening techniques have been deployed to protect DNN executables compiled by DL compilers.

Software *sanitizers* [73] are designed to harden executables and detect bugs. They instrument programs during compilation and insert checks that raise alarms when unsafe operations are performed at runtime. Various sanitizers [8, 50, 53, 70, 75] have helped uncover vulnerabilities in C/C++ software. Inspired by the success of sanitizers in securing executables compiled from C/C++ programs, this research proposes OBSAN, a sanitizer to detect DNN executable's abnormal behaviors, known as out-of-bound (OOB) behaviors, which frequently result in mispredictions and exploitation opportunities.

DNNs often process high-dimensional media data like images and make predictions. Holistically, they feature bidirectional computations, namely, forward and backward propagation. Forward propagation predicts an input using distinct neuron activations.[1] Backward propagation analyzes how predictions are made by propagating gradients from the output to the input through layers. Importantly, both neuron activations and gradients should fall within their *normal ranges* [28, 46] for inputs considered benign. Thus, OOB neuron activations or OOB gradients observed indicate DNN mis-behavior under adversarial examples (AEs) or other anomalous inputs.

We design two variants of OBSAN, FOBSAN and BOBSAN, to check for OOB forward neuron activations and backward gradients in DNN executables. While FOBSAN discovers abnormal inputs by comparing neuron activations with their known safe ranges, BOBSAN uses rich information encoded in gradients to achieve this goal. A common challenge for sanitizers including OBSAN, however, is the incurred high runtime overhead [73], which often inhibits their production adoption. We propose several optimization schemes to reduce

*Corresponding authors.

Network and Distributed System Security (NDSS) Symposium 2023
27 February - 3 March 2023, San Diego, CA, USA
ISBN 1-891562-83-5
https://dx.doi.org/10.14722/ndss.2023.24103
www.ndss-symposium.org

[1] Following the convention [61], neuron outputs are referred to as "neuron activations" in this research.

OBSAN overhead; in particular, quantization [20] helps replace floating point number computation with integer computation, effectively lowering computational and storage requirements while preserving inference and detection accuracy. Moreover, we extend popular DNN layer pruning techniques [25] to debloat inserted OBSAN checks. We identify "unimportant" layers in DNN, and remove their accompanied OBSAN checks. Furthermore, we explore optimizations that are specific for FOBSAN or BOBSAN. These optimizations can be combined for synergistic effects to effectively lower OBSAN overhead without undermining OOB detectability.

Both of OBSAN's variants, FOBSAN and BOBSAN, are designed as an extra compilation pass at the IR level of the industry-leading DL compiler, TVM [19]; implementing OBSAN using TVM's Relay IR allows us to bypass design discrepancies between different DL frameworks. Relay IR is generally analysis friendly, offering type system, analysis, and optimization utilities for use. We evaluate three large-scale DNN models with diverse structures, high volumes of parameters (up to 23.5 million), and large numbers of layers (over 100). We show that the vanilla FOBSAN and BOBSAN provide high OOB detection capabilities of AEs (less than 0.5% false negatives) and rarely treat normal inputs as OOB inputs (less than 1.2% false positives). With quantization/debloating, we successfully reduce the overhead of FOBSAN from 121.3% to 47.6% and that of BOBSAN to -34.3% (comparing with the baseline executables containing no OBSAN) while maintaining decent usability, therefore promoting the practical adoption of OBSAN in production. In terms of real-world usage, we show how FOBSAN facilitates detecting inputs of broken content given media data are presumed to semantically meaningful, and how BOBSAN enables the identification of undefined inputs under the background that DNN is trained for predicting a set of predefined labels — predictions for inputs outside its knowledge are worthless. We further present that FOBSAN and BOBSAN enable the prevention of online query-based AE generation and extend FOBSAN for feedback-driven fuzz testing of DNN executables. We also present conceptual and empirical comparison between OBSAN and existing approaches, and discuss migrating OBSAN to other DL compilers and architectures. Our main contributions are:

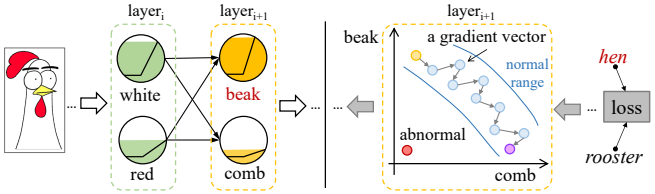
- This paper, for the first time, advocates for *hardening DNN executables*, an important yet under-researched area. We design OBSAN to check OOB, a common indicator of DNN mis-behaviors that can induce mis-predictions and exploitations by AE and other anomalous inputs.
- We instantiate two variants of OBSAN to capture forward- and backward-based OOB behaviors. We propose several optimization schemes to largely reduce OBSAN overhead without undermining its effectiveness.
- Our evaluation on the large-scale DNN models shows that OBSAN can deliver low-cost sanitization, and it enables critical and representative downstream applications with high effectiveness.

We publish the supplemental materials for this paper and source code of OBSAN at [2] to benefit future research.

II. PRELIMINARY

A. Deep Neural Networks

From a holistic view, a DNN, dubbed as F_θ, denotes a parameterized non-linear function enabling bidirectional computations: 1) a forward propagation that produces the prediction $F_\theta(x)$ for input x, and 2) a backward propagation that analyzes how the prediction is made. We now introduce each form of computation in the following.



(a) DNN forward propagation when processing a rooster image. Multiple neurons are activated and the neuron w.r.t. beak has the highest activation.

(b) DNN backward propagation after making a prediction. Gradients adjust the decision logic (●→●) to comb rather than beak to diff hen vs. rooster.

Fig. 1: Forward and backward computations of DNNs.

Forward Propagation. During the forward propagation phase, an input x is fed into the DNN such that neurons in each layer accept their input data (passed from the previous layer), process it according to the activation function, and pass it to the next layer. In general, neurons comprise the minimal unit of computation in forward propagation. Existing works have empirically demonstrated that a neuron typically signifies certain features in the input [14]. For instance, when processing an image for a rooster (as in Fig. 1(a)), certain neurons recognize its color, whereas some other neurons (often in deeper layers) signify its beak/comb features. When recognizing certain features in an image (e.g., the color "white" in Fig. 1(a)), the corresponding neurons would manifest a high activation level. Therefore, DNN behaviors can be characterized via neuron activations. In general, it is well acknowledged that each neuron has a *normal activation range*, such that unsafe DNN behaviors can be detected by observing neuron activations that fall beyond the normal range [46, 61].

Backward Propagation. Gradients characterize the decision logics of a DNN. For instance, the gradient of layer$_{i+1}$ (i.e., a 2-dimensional vector) marked as the yellow dot in Fig. 1(b) indicates that the DNN primarily relies on beak to recognize a rooster. During the training stage, the DNN adjusts its decision logics according to gradients. As shown in Fig. 1(b), the DNN prediction over the input, currently "hen" (incorrect) since it relies on beak, is compared with the ground truth label. The resulting "distance" (i.e., the loss) is back propagated into the DNN through gradients. Then, DNN decision logics are gradually tuned by modifying θ (e.g., $F_\theta \rightarrow F_{\theta'}$) until reaching saturation, i.e., the purple dot in Fig. 1(b) which focuses on comb, thereby correcting the prediction for "rooster." In short, gradients provide rich information on how a DNN makes decisions, and the magnitudes of gradients are associated with their "importance" in the decision process. Similar to neuron activations, the magnitudes of gradients also have normal ranges. Unsafe DNN behaviors, such as the red dot in Fig. 1(b) whose decision logic considers neither beak nor comb, are likely caused by AE inputs and can be identified by checking for gradients falling out of their normal ranges [28].

Two Variants of OBSAN. To capture OOB, we propose two sanitizers, namely, FOBSAN and BOBSAN, which monitor DNN executable's forward and backward computations, respectively. By observing OOB defined over neuron activations (for forward propagations) and gradient volumes (for background propagations), the two implementations can detect abnormal DNN behaviors in various scenarios. Note that BOBSAN is inserted in a pre-trained DNN executable and it only calculates the gradients without updating θ. We clarify that DNN executables compiled by TVM are immature for computing gradients, and we implement several new TVM operators to compute gradients; see details in Appx. B.

B. DL Compilers

DL compilers typically accept a high-level description of a *well-trained* DNN model, exported from DL frameworks like TensorFlow as their inputs. Such DNN descriptions are often encoded as computation graphs, specifying the connectivity of different operators in a DNN model without defining how exactly each DNN operator are implemented on the hardware.

Compiler Frontend: Graph Optimizations. Typically, DL compiler frontends convert the DNN computation graphs into graph intermediate representations (IRs) that are hardware independent. Graph IR facilitates platform-independent graph-level optimizations like operator fusion and layout transformation [19, 68]. For instance, "operator fusions" merge certain neighboring DNN operators to reduce overhead. Since DNN neurons are retained in the graph IRs, we implement OBSAN as several instrumentation passes toward the graph IR (namely Relay IR [5]) of TVM.

Compiler Backend: Low-Level Optimizations. Graph IRs specify how inputs of each operator are mapped to the outputs, without restricting how each operator should be implemented using machine code. Typically, a low-level IR is derived from optimized graph IRs for hardware-specific optimizations. Some DL compilers implement their own low-level IR to exploit hardware characteristics for optimizations [68]. Typical optimizations can include memory allocation, latency hiding, and loop-related optimizations [12, 19, 66, 99]. We also find that some DL compilers may convert their customized IR into standard tool-chains like LLVM IR [38] or CUDA IR [55] to harvest low-level optimization opportunities.

Code Generation. Low-level IRs are further compiled, using either just-in-time (JIT) or ahead-of-time (AOT) paradigms, to generate executable code for different hardware targets like CPUs and GPUs. To date, most DL compilers can generate standalone DNN executables as well as shared objects (the latter one can be used by linking with user application code) for shipping. This paper, for the first time, discuss security hardening techniques for the compiled DNN executables and shared objects. Our designed sanitizer checks, OBSAN, are inserted into TVM compiled executables/shared objects to provide low-cost security hardening. Nonetheless, OBSAN is not limited to TVM; see the extensibility in Sec. X-B.

C. Software Sanitizers

Sanitizers insert checks and specialized metadata structures to harden software; the inserted checks monitor program execution during in-house testing or online execution. When security properties are violated, sanitizer checks abort the execution and notify users. We introduce two sanitizers, address sanitizer (ASan) and undefined behavior sanitizer (UBSan), both of which have been critical in detecting many vulnerabilities [73].

ASan. ASan detects memory corruption errors and enforce memory safety [70]. ASan encodes the accessibility of each memory byte in its corresponding shadow memory. When compiling an input program, ASan instruments every memory access to check whether the memory address $addr$ is valid by mapping $addr$ to its corresponding shadow memory address:

$$addr_{shadow} = (addr \gg 3) + offset$$

where shadow byte B at $addr_{shadow}$ is loaded to check whether the access via $addr$ is safe. Values in B encode different memory accessibilities, where $B = 0$ means that all 8 bytes starting from addr are accessible. $B = k$ $(1 \leq k \leq 7)$ means that the first k bytes are addressable, and any negative value indicates that accessing $addr$ is invalid. Shadow memory itself at $addr_{shadow}$ is mapped to an inaccessible "bad" region.

UBSan. C/C++ programs may have undefined behaviors, which can induce serious vulnerabilities [87]. UBSan [50] identifies many undefined behaviors in C/C++ code, including out-of-bounds access, integer overflow, and division by zero. We briefly introduce how UBSan detects divide by zero. Consider the following code:

```
int quotient = dividend/divisor;
```

where an undefined behavior will be triggered when `divisor` is zero. UBSan detects this by duplicating `divisor` into two copies and checking whether the first copy equals zero before executing the division operation with the second copy.

III. MOTIVATION, ASSUMPTION, AND USAGES

Motivation. DL compilers provide systematic optimizations to boost the adoption of DNN models. To gain benefits from DNN models, suppliers of edge devices and low-power processors have been incorporating DL compilers into their systems [31, 56, 57, 65, 93]; cloud service providers such as Amazon and Google are also including DL compilers in their DL services to boost performance [9, 90]. In particular, Amazon has been seen to spend major efforts to compile DNN models on Intel x86 CPUs through use of DL compilers [32, 44]. Facebook is also seen to deploy compiled DNN models onto Intel CPUs [52]. Overall, it should be accurate to assume that DL compilers are increasingly vital to boost DL on heterogeneous hardware backends.

DL compilers have not yet provided security solutions to fortify DNN executables: prior works on AE detection are difficult to apply to DNN executables for various reasons, as will be noted in Section X-A. Recent community efforts have focused mostly on testing DL compilers [42, 71, 91], as opposed to securing DNN executables or analyzing their attack surface. This work designs OBSAN, the first security hardening solution integrated in the production DL compiler as extra compilation passes. OBSAN introduces sanitizer checks in DNN executables, enabling low-cost detection of OOB, a common suspicious behavior of DNNs. We illustrate the high OOB detectability and low cost of OBSAN in production DNN executables, and also use OBSAN in promoting downstream attack mitigation (Sec. IX-A) and feedback-driven fuzz testing

of DNN executables (Sec. IX-B). It should be noted that, however, although C/C++ sanitizers (e.g., ASan and UBSan) have few to none false positives, sanitizers for DNN executables can introduce an observable amount of them. OOB inputs for DNNs have no obvious "patterns" like out-of-bound C pointers and are inherently harder to detect. Given that said, OBSAN keeps false positives at a low level (on average 4.2%; see Sec. VIII-A) to ensure normal usages are seldom interrupted.

Main Audiences and Requirements. The main audiences of OBSAN are DNN model owners who compile their *well-trained DNN models* using DL compilers. OBSAN is integrated into the compilation pipeline of production DL compilers to generate hardened DNN executables. To clarify, OBSAN hardens a DNN executable in its prediction phase; it is *not* used to detect OOB issues during the "compile time" of the DNN executable or during the model training phase. OBSAN detects OOB issues in production usage of released DNN executables with reasonable costs (see Sec. VIII).

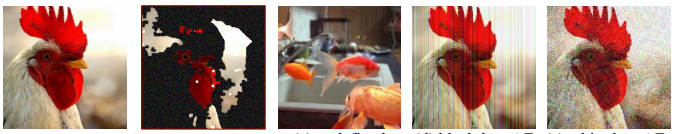
To facilitate initializing OBSAN checks, we require model owners to use a dataset representative of the expected benign inputs for the model. As a common approach, they can use standard model training datasets; typical training datasets (e.g. CIFAR-10 [37] adopted in our evaluation) shall depict the normal behavior comprehensively. This is consistently assumed by many existing AE detection works [48, 74, 85].

To clarify, the adopted training datasets do not need to be exposed to other (untrusted) parties. The main audiences of OBSAN, model owners, can use their local training datasets to train a model, and then use DL compilers to compile the trained model into an executable. During the compilation, OBSAN is inserted into the executable (as how ASan/UBSan is injected into C/C++ executables). Then, the OBSAN-injected DNN executable is released for use. For other cases where model owners sell their trained models to users and let them to compile, model owners can distribute partially optimized models (after the quantization process which requires training data; see Sec. V-B) together with the range data recorded for FOBSAN, and users can compile the models with DL compilers and enable OBSAN. In this case, only necessary (e.g., range) data derived from the training set is published.

We envision the use of OBSAN can notably increase the complexity of synthesizing AEs, even if that the data capturing normal behaviors was disclosed publicly (e.g., the "range data" noted above). This is consistently assumed by existing AE defense works: AE generation is typically a multi-objective optimization process, and OBSAN checks introduce many extra "constraints" (optimization objectives) that need to be considered in this process.

Usage Cases. OOB encapsulates a variety of defect-triggering DNN inputs (as in Fig. 2) that need users' attention. We discuss each of them as follows:

Detecting perception-broken inputs. Audiences may be aware that detecting images with invalid formats (e.g., RGB image pixels falling outside of $[0, 255]$) is straightforward. Nevertheless, in addition to format validity, real-world images contain perception-level contents (e.g., ear in a portrait photo) that are human-perceivable. In fact, DNN models generally presume that inputs are real-world meaningful images. More specifically, there are constraints over the pixels values of input images such that the content is human-perceivable [27, 106], akin to the input constraints (e.g., type or range) of conventional programs. DNNs lack input check routines.[2] Thus, *detecting images with broken contents* (as in Fig. 2(b)) is challenging and critical, since DNNs are deemed to behave abnormally when processing broken inputs. To our observation, input with perception-broken (or perception-meaningless) contents would trigger OOB and will be captured by OBSAN.



(a) normal (b) perception-broken (c) undefined (d) black-box AE (e) white-box AE

Fig. 2: Usage cases of OBSAN. OBSAN can detect various abnormal inputs (b–e) by checking OOB. It, however, will *not* flag normal inputs (a). When drawing this motivating example, we increase the attack budget in (d) and (e) to better visualize the adversarial perturbations over AEs. AEs with different visual quality are evaluated in Sec. VIII and Sec. IX-A.

Detecting undefined inputs. OBSAN also captures undefined inputs. Suppose a DNN is trained to classify between cat and dog images, then a fish image (as in Fig. 2(c)) can be treated as an undefined input. Instead of randomly "classifying" this image, we underline that DNN should alert the model owner explicitly. This is not easy, as DNNs are known to be over-confident in its inputs [13, 79]. That is, though it randomly classifies an undefined input as a cat or a dog, the associated confidence score is usually high and similar (hardly distinguishable) to that of classifying normal inputs. However, OOB is generally observable internally in DNN, suggesting the viability of detecting undefined inputs using OBSAN.

Detecting AE. Besides images with broken/undefined contents, we highlight that OBSAN can be used to detect AEs, which are images visually identical to normal images, but with subtle (pixel-level) changes that can confuse a DNN and incur mispredictions. AEs are often images that are visually similar to normal inputs and do not affect human judgments, yet they lead DNNs to behave abnormally. To date, AEs have been extensively leveraged to exploit DNNs, causing substantial confusion or even severe consequences in daily usage [18]. Though AEs are much more subtle than broken/undefined images, recent research [46] has shown that AEs can frequently trigger OOB behaviors as well. As a result, we envision that OBSAN will be able to detect AEs, preventing active attackers from manipulating the prediction outcomes of DNN executables. To clarify, de facto AE generation algorithms can be classified into white-box and black-box: white-box methods generate AEs under the guidance of DNN gradients whereas the black-box methods, with wider application scope, can generate AEs by only using the DNN prediction outputs as the feedback. We present two AEs generated using the black-box and white-box algorithms in Fig. 2(d) and Fig. 2(e), respectively. OBSAN is evaluated for detecting white-box AEs in Sec. VIII and mitigating black-box AE generation (as a downstream application) in Sec. IX-A.

Extended Applications. We also discuss two important downstream applications enabled by slightly extending OBSAN.

[2] Extracting perceptual-constraints is still an open problem.

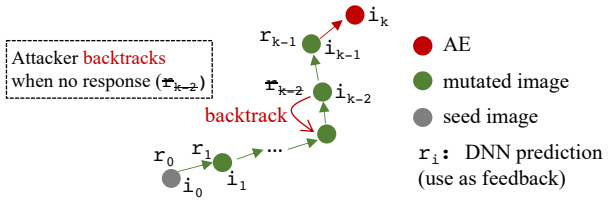


Fig. 3: A schematic view of online AE generation attacks. In the k-th iteration, the attack mutates input i_k to i_{k+1} according to feedback r_k (e.g., DNN predictions). By repeating the iteration progressively, an AE is generated from the seed input i_0. OBSAN can mitigate the attack by interrupting the progress and forcing the attacker to backtrack.

Mitigating online adversarial attacks. In practice, online adversarial attacks for DNN executables (i.e., generating AEs) are launched by iteratively accessing the output predictions and perturbing inputs, as seen in Fig. 3. By impeding this generation progress, OBSAN can be extended to mitigate online AE generation. We find that when the mutated input is "close" to AE (i.e., becomes an AE after only a few iterations), though the prediction has not yet flipped, OOB behaviors are triggered in most cases. Therefore, when OBSAN detects OOB, it can be extended to yield a dummy (empty) response to remote adversary queries. As a result, the attacker receives no feedback for mutating inputs and is forced to backtrack and perform random mutations, as illustrated in Fig. 3. See evaluations in Sec. IX-A.

Enabling feedback-driven fuzzing. DNN fuzz testing detects security flaws or mis-predictions of DNN models. Software greybox fuzzing is usually guided by the code coverage, assuming that "high code coverage promotes uncovering more bugs". Nevertheless, fuzz testing for *DNN executables* are not yet explored. Overall, fuzzing DNN executables denotes a costly and challenging setting, such that coverage information is obscure within DNN executables — a black box. In addition to detecting anomaly inputs, OBSAN checks can be slightly extended to constitute feedback to guide DNN executable fuzzing. In this setting, the OOB checks of OBSAN are modified to record neuron states following previous literatures [46, 58, 61]. Aiming at maximizing the covered neuron states, neuron coverage-guided fuzzing can be effectively conducted on DNN executables; see details in Sec. IX-B.

IV. Overview: OOB Detection

A. Detecting Forward OOB with FOBSAN

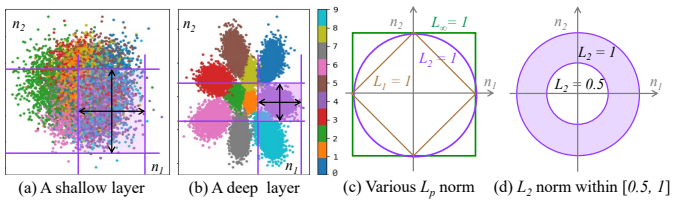


Fig. 4: Motivating examples. (a) and (b) show that during the forward propagation, neuron activations for inputs belonging to the same prediction class are bounded. (c) and (d) illustrate L_p-norm in the two-dimensional space.

Forward Neuron Activation. Consider Fig. 4(a)-(b), which illustrate the activations of two randomly selected neurons from a shallow layer and a deep layer from a well-trained convolution neural network performing classification. Each dot corresponds to one normal DNN input (i.e., training data) and each color denotes one predicated class. Though the distributions vary with layers, neuron activations for inputs belonging to the same prediction class tend to concentrate into a bounded region (i.e., the purple region in Fig. 4(a)-(b)). It is generally acknowledged that the output bounds specified by normal inputs describe the normal behavior of each neuron [46]. Consider a neuron n_1 whose normal output w.r.t. class c lies within $[l_1, u_1]$. A defect-triggering input, which induces unsafe/abnormal behaviors of the DNN, will likely lead to OOB outputs of n_1 that fall out of $[l_1, u_1]$. Overall, we define forward OOB as follows:

> Let the normal output bounds of a neuron n be $[l_n^c, u_n^c]$ w.r.t. class c. A forward OOB occurs when an input i^* causes the output of n to fall out of $[l_n^c, u_n^c]$.

Deciding Activation Bounds $[l_n^c, u_n^c]$. Existing work [46] relies on profiling DNN models using training data to determine the normal neuron output bounds; training data is usually believed to encode the typical behavior of DNNs. Recall, as stated in Sec. III, that when FOBSAN is used to harden DNN executables, a training dataset is required. Overall, to harden a well-trained DNN model m, we first profile m by feeding it the training data and logging the normal value bounds $[l_n, u_n]$ for each neuron n; this normal value bound is taken as $[l_n^c, u_n^c]$ for class c, where c is the class the training input belongs to. Then, when compiling m into an executable e, we place one FOBSAN check to hook the output of each neuron n. Then, during normal usage of e, each FOBSAN check will constantly check if the output of its hooked neuron n falls out of the normal range $[l_n, u_n]$. If more than T neurons (see below) manifest OOB outputs, the FOBSAN check will report the abnormality to the user.

Deciding Threshold T. Given DNNs process diverse inputs, it is reasonable that some normal inputs trigger the OOB outputs of some neurons, but the DNN behaves normally.

Our preliminary study shows that an abnormal input triggers many more OOB neurons than that of the normal inputs. Therefore, we set a threshold T for #OOB to detect abnormal inputs such that FOBSAN will not flag an input as anomaly unless its triggered #OOB outputs is above T. Intuitively, as in Fig. 4(b), neuron activations (w.r.t. one class) distribute toward certain (not all) directions. That is, it is reasonable to assume that, when sufficient training data have been fed, the majority of neurons' activation ranges will have been fixed, and only a few neurons still need to expand their output bounds when more training data is fed. Therefore, after feeding 90% training inputs to update the bounds $[l_n^c, u_n^c]$, we use the reamining 10% of data to both update the bounds *and* decide T: we record #OOB (number of neurons with OOB outputs) whenever a training input causes the bounds to be updated, and then set T to $(1-\mu)a+\mu b$, where a and b are the recorded minimum and maximum #OOB, and $0 \leq \mu \leq 1$ is a user-decide parameter to indicate the user's preference between fewer false negatives (when μ is closer to 0) and fewer false positives (when μ is

closer to 1). In practice, we find $\mu = 0.3$ to be a good choice, and we use this value to evaluate FOBSAN.

B. Detecting Backward OOB with BOBSAN

Backward Gradients. DNN gradients encode rich information of how predictions are made (since it adjusts the decision logic during training; see Fig. 1(b)) and many existing works leverage DNN gradients to explain the mechanism behind each prediction [69, 77]. As demonstrated in Eq. 1, for an l-layer DNN $F_\theta(x) = f_l \circ f_{l-1} \circ \cdots f_2 \circ f_1(x)$ with parameters $\theta = (\theta_1, \theta_2, \cdots, \theta_l)$ and input x, the gradient is (inversely) back-propagated using $g(x)$ via layers.

$$\frac{\partial g(x)}{\partial \theta} = \frac{\partial g(x)}{\partial f_l} \frac{\partial f_l}{\partial f_{l-1}} \frac{\partial f_{l-1}}{\partial f_{l-2}} \cdots \frac{\partial f_2}{\partial f_1} \frac{\partial f_1}{\partial \theta} \tag{1}$$

Recent research has pointed out that gradients over DNN weights, when propagated using the Kullback-Leibler (KL) divergence between DNN output (i.e., a vector of confidence scores for all classes) and the uniform distribution U (assuming all classes have equal confidence scores), have bounded magnitudes [28]. The gradient computation is formulated as:

$$\frac{\partial g(x)}{\partial \theta} = \frac{\partial D_{KL}(U||F_\theta(x))}{\partial \theta}, \tag{2}$$

where x is the model input, $U = (1/C, 1/C, \cdots, 1/C) \in \mathbb{R}^C$, C is the number of classes, and $D_{KL}(p||q) = \sum_j p_j \log \frac{p_j}{q_j}$. Note that unlike gradient computation during DNN training which propagates the gradient using the distance (defined by developers) between DNN outputs and the ground truth label, this formulation does *not* require the ground truth label of an input; the uniform distribution U forms the reference to measure gradient deviations. Thus, it can be used as sanity checks for DNN executables when processing arbitrary inputs in the wild. Following, we define backward OOB as:

> For an l-layer DNN, let the range of a normal gradient at layer k (in a high-dimensional space) be $[l_k, u_k]$, then a backward OOB occurs when the gradient $\frac{\partial g(x^*)}{\partial f_l} \cdots \frac{\partial f_{k+1}}{\partial f_k}$, which is propagated using $g(x^*) = \partial D_{KL}(U||F_\theta(x^*))$, falls out of $[l_k, u_k]$.

Deciding Gradient Bound with L_p-Norm. Unlike neuron activations which are scalar values, the gradient $G_k(x^*) = \frac{\partial g(x^*)}{\partial f_l} \cdots \frac{\partial f_{k+1}}{\partial f_k}$ of the k-th layer is a vector of often high dimensions. In practice, its magnitudes are characterized using L_p-norm, which denotes the distance between a vector and the origin, as formulated in Eq. 3.

$$L_p(G_k) = \sqrt[p]{G_k[1]^p + G_k[2]^p + \cdots + G_k[d]^p} \tag{3}$$

Fig. 4(c) displays sets of points (connected as lines) whose $L_p = 1$ with varying p values in a 2-dimensional space; p value decides the geometry of the distance criterion. Fig. 4(d) highlights points (equivalent to 2-dimensional gradients) having L_2 norm within $[0.5, 0.1]$. The gradient value of a DNN layer, if falling out of the purple ring, is deemed a backward OOB. For BOBSAN, we feed 10% of the training data as normal inputs to the instrumented DNN and collect the values of $\mathcal{L} = \{L_p(G_k(r))\}$ for every input r. We then set l_k, u_k to $P_\alpha(\mathcal{L})$ and $P_\beta(\mathcal{L})$, respectively, where $P_x(\cdot)$ represents the x^{th} percentile. Similar to FOBSAN discussed earlier, we provide α and β as two configurable parameters to allow tweaking of BOBSAN. Based on empirical observation, we use $\alpha = 1$ and $\beta = 99.5$ for BOBSAN evaluation and encourage users to tune these parameters based on their needs. We refer readers to Sec. V-B for suggestions on deciding p.

V. OBSAN DESIGN

Given the prosperous development of TVM and its active community, we implement OBSAN in TVM and will explain how the two variants of OBSAN are integrated with TVM. We then discuss three optimizations to reduce OBSAN overhead. In We discuss implementation details of OBSAN in Sec. VI.

A. Hardening DNN Executables with OBSAN

Output of Hardened DNN Executable. An illustration of using OBSAN to protect DNN executables is given in Fig. 5. A DNN model, either in the high-level PyTorch/ONNX format (Fig. 5(a)) or in unprotected DNN executable format (omitted in Fig. 5), only outputs predictions (i.e., class labels and confidence scores). In contrast, as shown in Fig. 5(c), DNN executable hardened with OBSAN is extended to yield additional OOB "alert" (i.e., error messages). The reason for this design choice is two-fold: 1) Users (developers) do not usually expect DNN models, even in their executable form, to throw exceptions, as they normally consist of only arithmetic operations. 2) It will be significantly harder for an exception-throwing OBSAN to interact with other tools due to the lack of a standard on exception handling in DNN executables. As an implied result, OBSAN will *not* terminate the execution immediately after OOB is detected, unlike ASan or UBSan.

Instrumenting DNN Models. Fig. 5(b) introduces how OBSAN is incorporated into TVM. As introduced in Sec. II-B, TVM first converts the input DNN model into graph IR (namely Relay IR), and also performs TVM frontend optimizations. More importantly, we provide another optimization $\mathcal{O}_1$ (pruning; see details in Sec. V-B) at this step to reduce the overhead of the entire (instrumented) model.

Then, OBSAN is involved to instrument the model on its converted Relay IR. Depending on the variant of OBSAN, the instrumentation will function in different modes. For FOBSAN, the model needs to first be instrumented in a so-called "Record Mode," which produces an auxiliary recorder executable instead of the final protected executable. This intermediate executable facilitates the collection of the normal output bounds of each neuron, as introduced in Sec. IV-A. After this data is collected by feeding each training data sample into the recorder executable, the original model is instrumented again in "Detect Mode," where the recorded bounds of each neuron n will be embedded in n's FOBSAN check inserted into the hardened executable. For BOBSAN, however, the model is directly instrumented in Detect Mode where instructions to calculate backward OOB information, as outlined in Sec. IV-B, are inserted into the model.

From a high level, OBSAN has design principles similar to those of software sanitizers: given a program (DNN model), OBSAN identifies "interesting" spots and inserts checks to hook these these locations and perform sanitization tasks like bounds checking. To instrument a model, the instrumentation pass of FOBSAN traverses the model's computational graph in Relay IR and hook each neuron. As for BOBSAN, we implement new TVM operators, in accordance with DNN

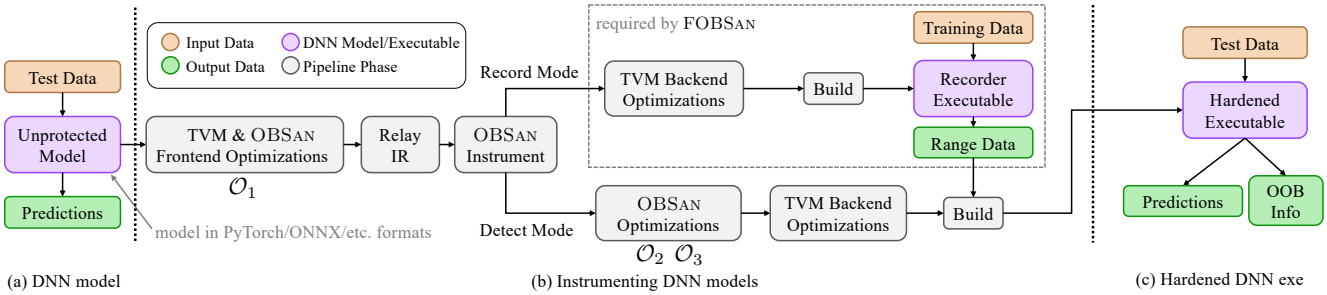


Fig. 5: Workflow of OBSAN. OBSAN incorporates three available optimizations $\mathcal{O}_{1-3}$ at different phases of the instrumentation.

operators, to compute gradients; see details in Appx. B. We present two DNNs instrumented using either FOBSAN or BOBSAN in Appx. A to ease understanding.

Optimizations to Reduce OBSAN Overhead. OBSAN is designed to insert a subgraph into the original computation graph of the input DNN in Relay IR. This allows TVM's graph-level and follow-up backend optimizations to be applied normally to OBSAN as well. More importantly, we design and apply three optimization schemes $\mathcal{O}_{1-3}$ to lower the overhead of inserted OBSAN checks. As shown in Fig. 5(b), one of them, $\mathcal{O}_1$, is applied globally to the entire computation graph. The other optimizations, $\mathcal{O}_{2-3}$, are applied locally to the inserted OBSAN subgraph and do not alter the original model's computational graph; optimization details are in Sec. V-B. Also, in Record Mode, these optimizations are omitted as there is no OOB detection logic in the auxiliary recorder model.

OBSAN then runs TVM's standard backend optimization passes on the resulting model to ensure any follow-up or lower-level optimizations are applied to the instrumented model as a whole, and finally invokes TVM's build module to produce the output DNN executable for use, as shown in Fig. 5(c).

B. *Optimizing Overhead of* OBSAN

Similar to conventional sanitizers like ASan/UBSan, OBSAN incurs high overhead as it extensively hooks DNN layers/neurons. Thus, we present three schemes $\mathcal{O}_{1-3}$ below to reduce OBSAN overhead. Among them, $\mathcal{O}_1$ does not need to access Relay IR and is applied in the early optimization stage (Fig. 5(b)), while $\mathcal{O}_{2-3}$ are applied on the subgraphs inserted by OBSAN. As a supplement, we also present and tentatively evaluate an intuitive extension of $\mathcal{O}_2$ on the project website [2]. We now give more details on the optimization schemes.

$\mathcal{O}_1$ **Quantization.** Quantization is a well-established technique to reduce computational and storage complexity of DL models. It uses integers as a low-precision substitution for the floating point numbers involved in forward and backward propagations while retaining the prediction accuracy of the models [34, 35, 84, 96, 105]. While a wide range of techniques exist for optimizing DNN models, quantization is perhaps the most frequently considered optimization when building DL systems on low-power devices (in contrast to GPUs), a scenario also primarily targeted by DL compilers like TVM. We use quantization to reduce the overhead incurred by OBSAN.

To clarify, at this step we leverage a post-training quantization scheme; it calibrates the quantized model by computing quantization parameters (zero point and scale) based on the activations of the DNN model when training data is fed, creating a floating point $\rightarrow$ integer mapping of higher quality. Also, we use mixed-precision quantization, where the quantized model still uses full-precision representations for certain intermediate results (e.g., outputs of conv2d operations) and performs dequantization and requantization when needed; this circumvents the severe precision loss and overflow problems that can arise in single-precision quantized models. Therefore, our quantization does not notably degrade prediction accuracy (only 0.4% on average for all three models; see Table V). In sum, we observe that quantization:1) lowers the model execution time without causing large accuracy losses, and 2) applies globally to the whole model. Thus, we deploy it at the early optimization stage (as in Fig. 5(b)). We use the quantization module from the ONNX Runtime toolbox [1]. The default setup, as adopted in OBSAN, converts floating point numbers to 8-bit integers.

$\mathcal{O}_2$ **Layer-Wise Checks Pruning.** A common optimization to reduce the performance overhead of software sanitizers is to debloat the inserted checks [81, 103]. Inspired by these efforts, we propose $\mathcal{O}_2$ to reduce overhead of both FOBSAN and BOBSAN. In particular, the proposed scheme, namely layer-wise checks pruning, determines whether a layer is important using a parameterized predicate (see below). If a layer is deemed unimportant, it will not be hooked by OBSAN.

For FOBSAN, we construct the pruning predicate by first assigning an importance score to each applicable layer in the model and then pruning the checks for layers with importance scores below a configurable level. To compute the importance scores, we use a popular heuristic [11, 51] based on weight magnitudes: Suppose a layer has n scalar weight parameters collected (flattened) into a vector w, then the importance score of the layer is $S = \frac{1}{n}\sum w_i^2$, where w_i denotes elements in w. The layers are then sorted by their importance scores, and the checks for a configurable fraction γ of the layers are pruned, starting from those with the lowest importance scores.

BOBSAN, on the other hand, requires the computational graph to be free of discontinuity (gaps) to perform backward propagation. Thus, users can instead configure the last n layers of the model as important. Moreover, we clarify that only a few layers will usually be sufficient for OOB detection regardless of the DNN model. n will typically be a small integer rather than a fraction like γ. In Appx. A, Fig. 7 will present a case, where after using $\mathcal{O}_2$, we only retain the last layer's connectivity to BOBSAN. To "prune a layer," we only remove its edges to the subgraph inserted by FOBSAN or BOBSAN. By leaving untouched the computational graph of the protected DNN, $\mathcal{O}_2$ does not affect the prediction accuracy of the DNN.

In contrast, $\mathcal{O}_1$ (quantization) globally optimizes the DNN model and the inserted OBSAN.

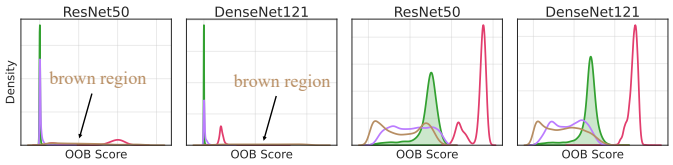
$\mathcal{O}_3$ **Deciding p for L_p-Norm.** This optimization scheme is exclusively designed for BOBSAN. Recall Sec. IV-B depicts the geometric interpretations of different L_p-norms. Though p is often used as a fixed hyperparameter, we clarify that the selection of p has an observable effect on OOB detectability. Moreover, varying p influences the execution overhead, e.g., $p = 2$ introduces more overhead than the other settings in computing L_p-norm if the DNN has extensively large layers or too many layers. Thus, it is necessary to select an optimal p value to maximize BOBSAN's effectiveness and efficiency. At this step, we explore the optimal value of p based on benchmark results; see our findings in Sec. VIII-B.

VI. OBSAN IMPLEMENTATION

Implementation as Compiler Passes. The current codebase of OBSAN [2] is implemented as several TVM passes. TVM provides a Python interface and APIs using which users can implement their own instrumentation passes. OBSAN uses this interface for the implementation. OBSAN is not limited to TVM, and we discuss migration in Sec. X-B. Users can select either FOBSAN or BOBSAN (or both). We give our recommendations on selecting FOBSAN and BOBSAN in Sec. VII. As a common setting, when compiling DNN into executable, we enable the default optimization levels of TVM.

Implementation Consideration. This paper champions to implement OBSAN at Relay IR level; this shares conceptually similar design consideration with C/C++-compiler-inserted sanitizers like ASan and UBSan. Though those sanitizers are technically feasible to be inserted into C/C++ source code, inserting them during compilation is more beneficial due to more analysis-friendly compiler IR and compiler analysis utilities. Doing so is also necessary for us to protect the most DNN models, as it allows us to bypass design discrepancies between different DL frameworks. For example, PyTorch has trouble exporting its Adaptive Max/Avg Pool operators to ONNX [3, 6]. And ONNX [4], though being an open exchange standard, lack many backward operator definitions like AvgUnpool, which we consider a consequence primarily of its forward-propagation-centric design goal. Nevertheless, since OBSAN instruments compiler IR directly lowered from DL framework languages, these problems are resolved. Moreover, implementing our checks at the IR level facilitates the use of TVM-provided analysis utilities and passes. For instance, local structures in graphs may change after quantization (e.g. modified/inserted nodes), and we perform flexible pattern matching with TVM's graph tools and the type system to recover their original semantics.

That said, instrumenting compiler IR poses several technical hurdles. First, as DL compilers focus mainly on forward propagation of DNNs, they usually come with incomplete support for backward operations like computing gradients. To solve this, we extended TVM by implementing support for gradient computation for more operators; we also implemented our backward propagation toolkit to insert checks for BOBSAN. Also, selecting the proper IR to work on requires deliberate consideration and exploration. For OBSAN, IRs we may choose from include Relay IR, TIR, and LLVM IR, ordered by their respective levels in the TVM compilation pipeline.



(a) OOB score distributions produced by FOBSAN. (b) OOB score distributions produced by BOBSAN.

Fig. 6: OOB score distributions for different types of inputs to DNN models instrumented with the two variants of OBSAN. Green and red curves represent normal and AE inputs, respectively, and undefined and perception-broken inputs are encoded in purple and brown.

Although TIR and LLVM IR may allow finer optimizations for protected executables, much information of the original computational graphs might be lost. Based on this, we decided that Relay IR will be a more suitable IR to work on.

VII. OBSAN USAGE SCENARIOS

TABLE I: The usage scenarios. Δ denotes partial support.

	perception-broken	undefined	AE
FOBSAN	✓	×	✓
BOBSAN	Δ	Δ	✓

Sec. III has discussed three representative usage scenarios of OBSAN. We outline FOBSAN and BOBSAN's suitability for these use cases in Table I and present empirical justification in Fig. 6. First, since "broken regions" are replaced with meaningless noise, a broken input can be viewed as $i + \delta$ where i is the normal input and δ is the noise. Perception-broken images thus primarily induce abnormal forward neuron activations. Therefore, we deem FOBSAN as proper to capture broken inputs in Table I. As illustrated in Fig. 6(a), the brown region (denoting broken inputs) is clearly separated from the normal inputs in the green peak. As for BOBSAN, the brown region and the green region have distinct peak, though certain amount of overlaps are present in Fig. 6(b).

An undefined input, though having meaningful contents, denotes a new class which leads to *new decision logics*. Hence, abnormal behaviors are more obvious when observing the back propagation, as back propagation characterizes the decision logic of a DNN (Sec. II-A). In this regard, BOBSAN is appropriate to detect undefined inputs, and as shown in Fig. 6(b), the purple region (denoting undefined inputs) contains peaks that are distinct from those of the normal inputs (green region). In contrast, the purple and green regions mostly overlap in Fig. 6(a). We clarify that, though BOBSAN shows a reasonable capacity to detect undefined images, this task is inherently challenging. Often, "undefined inputs" share aligned perceptual contents with training data. For instance, a dog image in the training data and a fish image not in the training data may present comparable visual features, such as yellowish colors and round shapes, to a DNN, creating challenges for BOBSAN to flag undefined images.

AEs alter a DNN's decision logic. Moreover, from the implementation viewpoint, AEs are often generated by adding carefully designed perturbations, which can also be formulated as $i+\delta$ (similar to broken inputs). Conceptually, therefore, AE induces *both* abnormal forward activations and backward gra-

dients. Fig. 6 empirically shows that AE can be well-separated from normal inputs using both FOBSAN and BOBSAN.

Similar to how ASan and UBSan are selected for use, users with concern on AE and perception-broken images can choose to enable FOBSAN, whereas users concerned more with AE and undefined images can opt for BOBSAN. Also, given both FOBSAN and BOBSAN can detect AEs, and they do *not* conflict with each other (i.e, the forward and backward computation), they can be further integrated as a hybrid OBSAN to boost the performance for detecting various kinds of anomalous inputs; see experiments in Sec. IX-A.

VIII. EVALUATION

This section studies the following research questions. **RQ1**: Can OBSAN achieve high OOB detectability when hardening DNN executables? **RQ2**: Can OBSAN be optimized to reduce the performance overhead using the three optimization schemes $\mathcal{O}_{1-3}$? **RQ3**: What is the effectiveness of OBSAN in supporting downstream AE defense and feedback-based fuzzing applications? Before answering each research question, we first report the evaluation setup. We then present evaluation to answer **RQ1** and **RQ2** in this section. Sec. IX answers **RQ3**.

TABLE II: Statistics of instrumented DNN models.

Model	#Parameters	#Neurons	#Layers
ResNet50 [26]	23.5M	26,570	54
GoogLeNet [72]	5.5M	7,162	56
DenseNet121 [64]	7.0M	10,250	121

DNN Models. Table II reports the statistics of three DNN models used for the evaluation. All three leveraged models are large-size DNNs with complex architectures and a diverse set of DNN operators. They are all widely-used for CV tasks. Each of them has 7k to 27k neurons with up to 23.5M parameters.

TABLE III: Processing time and exe size. "Compile/Instr." denotes compilation + instrumentation when OBSAN is enabled.

Model	Compile time (sec)	Compile/Instr. time (sec)		DNN exe size (MB)	DNN exe (with OBSan) size (MB)	
		FOBSAN	BOBSAN		FOBSAN	BOBSAN
ResNet50	30.2	38.5	30.7	89.9	92.5	89.9
GoogLeNet	36.6	49.1	37.2	21.4	22.7	21.4
DenseNet121	79.8	94.0	80.7	27.3	28.9	27.4

Processing Time & Size Increase. Our experiments are launched on an AMD Ryzen 3970X with 256GB of RAM. Table III reports the compilation costs with and without OBSAN. We consider compiling with OBSAN sufficiently fast as a one-time effort that only increases the compilation time by around 20%. In addition, FOBSAN checks also needs to be initialized by feeding samples from training datasets; we use the full training dataset (CIFAR-10 [37]) for this step and report that it takes about 2 minutes. We also report the average size increase of DNN executables fully instrumented with OBSAN is less than 10%, indicating trivial disk cost.

A. RQ1: OOB Detectability

Setup and Metrics. This section reports the OOB detectability and performance overhead of OBSAN. We compute the false positives (FPs) and false negatives (FNs) to assess the accuracy of detecting OOB inputs. To prepare test inputs, we use all 10,000 images from the test split of the CIFAR-10 dataset to form a collection of *normal inputs*. Then, we generate 10,000 AEs using one state-of-the-art AE generation algorithm, Projected Gradient Descent (PGD) [36, 49], with the default set of configuration which has a perturbation budget of $\epsilon = 0.3$. Note that, as this algorithm launches a white-box attack (requires DNN gradients to guide AE generation), it has proved very difficult to defend against, either theoretically or empirically [67, 98]. We use PGD to synthesize AEs against DNN models running on PyTorch; for each model, we generate 10,000 AEs, and we confirm all of them can trigger mispredictions of the corresponding DNN executables when not enabling OBSAN. For *undefined inputs*, we form a test set with 10,000 images from the ChestX-Ray8 dataset [88]; these images have radically different features and information from CIFAR-10 on which the DNN models are trained. Finally, we also generate 10,000 *perception-broken inputs* from the CIFAR-10 training split by randomly identifying image regions and replacing them with Gaussian noise. These regions are generated by applying erosion and dilation transformations from OpenCV [16], a common image processing toolkit.

We feed these four collections of inputs in alignment with the usage scenario distinctions discussed in Sec. VII to benchmark OOB detectability. That is, we feed the same normal and AE inputs to both FOBSAN and BOBSAN as both of them can detect these two categories of OOB. For perception-broken (PB) and undefined (UD) inputs, however, we feed the former only to FOBSAN and the latter only to BOBSAN. In accordance with Table IV, FP_{norm} is the ratio of normal inputs triggering OBSAN alarms while FN_{ae}, FN_{pb}, and FN_{ud} denote the ratio of AE, PB, and UD inputs that are overlooked by OBSAN checks, respectively. Ideally, all FP and FN cases should be rare. We also measure the inference time of each (OBSAN-enabled) DNN executable using TVM's built-in `benchmark` function and control the standard deviation at a low level (1% of mean) with repeated executions.

Results. Table IV reports the evaluation results. Overall, we interpret that OBSAN manifests promising OOB detectability. The FPs and FNs are reasonably low for different normal/abnormal inputs across all three models. The reader may find the FN_{ud} ratio for BOBSAN to be relatively high. As clarified in Sec. VII, flagging undefined inputs (which often share similar perceptual contents with the training inputs) is inherently different, and we deem these results as reasonable. Nevertheless, the FN_{ae} and FP_{norm} are generally low, indicating that OBSAN can accurately flag defect-triggering inputs of DNN executables and OBSAN rarely interferes the normal execution. During production usage, users may launch postmortem analysis on these small collection of inputs that trigger alert of OBSAN and confirm that they are FPs.

Despite the encouraging OOB detectability, performance overhead is notable: FOBSAN extensively instruments each DNN neuron, and it incurs an average of 102% (up to 143% for the DenseNet121 case) performance slowdown.

We clarify that some BOBSAN-specific optimizations are properly enabled when producing the results in Table IV as our preliminary study finds that BOBSAN with no optimization at all (i.e. when every layer is hooked) would have been impractically slow for informative comparison. We thus optimize BOBSAN to hook only the last layer of each DNN

TABLE IV: OOB detectability and overhead. Here, optimizations $\mathcal{O}_{2-3}$ have been applied to BOBSAN (details in Sec. VIII-B).

Model	Infer time (ms)	Infer time with OBSAN (ms)		FP_{norm} ratio		FN_{ae} ratio		$FN_{pb/ud}$ ratio	
		FOBSAN	BOBSAN	FOBSAN	BOBSAN	FOBSAN	BOBSAN	FOBSAN (PB)	BOBSAN (UD)
ResNet50	1.22	3.28	1.30	1.11%	6.11%	2.35%	0.01%	10.57%	65.36%
GoogLeNet	3.79	5.75	4.36	2.52%	9.17%	0.00%	0.00%	0.06%	77.78%
DenseNet121	2.65	6.45	2.64	1.27%	6.69%	0.01%	0.02%	0.32%	55.02%

TABLE V: Quantization evaluation for unprotected models (left half) and OBSAN-protected models (right half).

Model	Infer. time (ms)	Quant. Infer. time (ms)	Pred. acc.	Quant. pred. acc.	OBSAN variant	OBSAN				OBSAN + quantization			
						Infer. time (ms)	FP_{norm} ratio	FN_{ae} ratio	$FN_{pb/ud}$ ratio	Infer. time (ms)	FP_{norm} ratio	FN_{ae} ratio	$FN_{pb/ud}$ ratio
ResNet50	1.22	0.76	93.78%	93.55%	FOBSAN	3.28	1.11%	2.35%	10.57%	2.57	1.44%	0.00%	61.13%
					BOBSAN	1.30	6.11%	0.01%	65.36%	0.82	6.31%	0.00%	65.79%
GoogLeNet	3.79	1.63	92.85%	92.59%	FOBSAN	5.75	2.52%	0.00%	0.06%	3.60	2.89%	0.00%	25.07%
					BOBSAN	4.36	9.17%	0.00%	77.78%	1.67	9.38%	0.00%	75.96%
DenseNet121	2.65	2.21	94.03%	93.34%	FOBSAN	6.45	1.27%	0.01%	0.32%	5.74	0.90%	6.78%	27.05%
					BOBSAN	2.64	6.69%	0.02%	55.02%	2.28	4.72%	9.64%	73.62%

TABLE VI: Layer-wise checks pruning ($\mathcal{O}_2$) for FOBSAN.

Model	Pruning config.	Pruned #neuron/total	Pruned %neuron	Infer. time (ms)	FP_{norm} ratio	FN_{ae} ratio	FN_{pb} ratio
ResNet50 (baseline)	N/A	N/A	N/A	3.28	1.11%	2.35%	10.57%
ResNet50	0.2	7680/26570	28.9%	2.88	1.25%	0.82%	1.90%
ResNet50	0.4	17152/26570	64.55%	2.39	0.27%	28.21%	0.03%
ResNet50	0.6	22016/26570	82.86%	2.05	0.3%	99.42%	0.00%
ResNet50	0.8	25024/26570	94.18%	1.62	0.07%	100.00%	0.00%
ResNet50	1.0	26560/26570	99.96%	1.23	0.42%	22.84%	99.11%
ResNet50 (w/ quant.)	0.2	7680/26570	28.9%	2.19	1.40%	0.20%	49.79%
ResNet50 (w/ quant.)	0.4	17152/26570	64.55%	1.81	0.57%	24.13%	32.98%
ResNet50 (w/ quant.)	1.0	26560/26570	99.96%	0.75	0.25%	99.83%	98.00%

(optimization $\mathcal{O}_2$) and set $p = 2$ (optimization $\mathcal{O}_3$) for BOBSAN; we will discuss further details on this in Sec. VIII-B. In the following section, we explore reducing the incurred high performance overhead without primarily hindering the OOB detectability.

Answer to RQ1: When securing common DNN models, OBSAN can achieve a reasonably high accuracy in detecting OOB inputs with low FPs and FNs (particularly for AEs). Detecting undefined inputs is fundamentally difficult. Nevertheless, the performance overhead is notable.

B. RQ2: Optimization Evaluation

Sec. V-B introduces three optimizations $\mathcal{O}_{1-3}$. This section first benchmarks the effectiveness of individual schemes, and then assess combining them together. To ease comparison, we also report the overhead and detectability of OBSAN after combining all recommended optimizations in Table IX.

$\mathcal{O}_1$ **Quantization.** We report the quantization evaluation in Table V. We interpret that models, after quantization, manifest a reasonable boost in terms of inference speed (the second column in Table V). Nevertheless, the quantized executable, after inserting OBSAN, is still much slower comparing to the baseline, e.g., it takes the vanilla ResNet50 executable 1.22ms (as in Table IV) to make one inference, whereas FOBSAN can still double the cost (2.57ms). Also, as a tradeoff, $FN_{pb/ud}$ increased, particularly the latter. In contrast, we report that the detectability of AE inputs are still high; some evaluations even report FN_{ae} ratios dropping to zero. DenseNet121 appears to be an "outlier" with observably higher FN_{ae} ratios. To clarify, this is because we use the same threshold selection parameters (μ in Sec. IV-A and α and β in Sec. IV-B) for all three models to keep our setup easy to follow; in practice, users are suggested to fine-tune these parameters for individual models to explore the best performance. Additionally, quantization has no negative impact on FP_{norm}, meaning that normal use of protected DNN executables is largely uninterrupted. The changes observed in OOB detectability is mainly because quantization prioritizes to preserve the prediction accuracy for normal inputs during the floating point $\rightarrow$ integer remap process. This can compress the space for OOB data and cause the distributions for different types of data, as shown in Fig. 6, to converge and become harder to distinguish.

$\mathcal{O}_2$ **Layer-Wise Checks Pruning.** Recall $\mathcal{O}_2$ can be applied to optimize both FOBSAN and BOBSAN. In Table VI (first 6 rows), we report the pruning results for FOBSAN. Here, **pruning config.** refers to the value of γ mentioned in Sec. V-B, and as it increases towards 1, more checks are pruned. For BOBSAN, the pruning results are reported in in Table VII, where the **Last n layers** column reports the number of layers *retained* to hook with BOBSAN, starting from the last layer of a DNN. Overall, we observe that, with more layers pruned, the inference time is largely reduced, especially for BOBSAN where the performance gain can be as high as 36x; when nearly all checks are pruned, the inference cost is almost negligible (recall the baseline for ResNet50 is 1.22ms). Consistent findings are also reported for the other models, data of which are available on the project website [2].

FOBSAN and BOBSAN also behave differently to varying pruning configurations: For FOBSAN, FN_{ae} increases steadily from 0.82% to 100% as γ moves from 0.2 to 0.8, but it then drops to a lower level when γ is 1.0. Interestingly, FN_{pb} spikes at this configuration, rising from about 0% to 99%, rendering almost a complete loss of capabilities to detect perception-broken inputs. In fact, this can be seen as a radical remap of the OOB score space, similar to what may happen if an extreme quantization profile is applied: Compared with unpruned ResNet50 whose OOB score ranges from $[0, 26570]$, this pruning configuration limits the range to $[0, 10]$, resulting in more difficulty in separating different types of inputs.

BOBSAN, as shown in Table VII, shows steady improvements over all metrics as more check layers are pruned, and its performance peaks when only the last layer of checks are preserved; this is particularly desirable because it gives the best protection with the lowest runtime overhead. Recent research also underlines that the last layer usually has a high distinguishability of (anomalous) DNN behaviors [28]. However, we highlight that "checking only the last layer" may not be proper for all DNNs; depending on the DNN architectures (e.g., considering a very shallow model) and the type of input data, other layers may also be needed. Also, from Table VII, we see why it is unsuitable to consider unpruned BOBSAN as a baseline: DNNs can easily comprise more than 100 layers, at which level unpruned BOBSAN becomes impractical for real-world use. Therefore, for optimal results, we recommend always using the pruning optimization for BOBSAN and only retaining the checks for the last layer.

TABLE VII: Layer-wise pruning ($\mathcal{O}_2$) for BOBSAN.

Model config.	Last n layers	Infer. time (ms)	FP_{norm} ratio	FN_{ae} ratio	FN_{ud} ratio
ResNet50, $p = 2$	5	47.13	6.21%	18.82%	63.63%
ResNet50, $p = 2$	4	35.07	6.16%	24.53%	63.64%
ResNet50, $p = 2$	3	18.19	5.93%	13.18%	67.16%
ResNet50, $p = 2$	2	7.16	6.06%	1.00%	65.40%
ResNet50, $p = 2$	1	1.30	6.11%	0.01%	65.36%

TABLE VIII: Selection of p ($\mathcal{O}_3$) for BOBSAN under different n configurations for layer-wise checks pruning ($\mathcal{O}_2$).

Model	p	Infer. time (ms)	FP_{norm} ratio	FN_{ae} ratio	FN_{ud} ratio
ResNet50, $n = 1$	1	1.29	4.11%	20.76%	82.61%
ResNet50, $n = 1$	2	1.30	6.11%	0.01%	65.36%
ResNet50, $n = 1$	∞	1.29	5.92%	16.79%	67.47%
ResNet50, $n = 2$	1	6.72	2.70%	100.00%	89.62%
ResNet50, $n = 2$	2	7.16	6.06%	1.00%	65.40%
ResNet50, $n = 2$	∞	6.22	5.73%	71.74%	69.30%
ResNet50, $n = 3$	1	16.69	3.88%	100.00%	85.20%
ResNet50, $n = 3$	2	18.19	5.93%	13.18%	67.16%
ResNet50, $n = 3$	∞	15.06	5.92%	72.62%	68.80%

$\mathcal{O}_3$ Optimizing p for L_p-Norm. We explore an optimal p for BOBSAN's L_p-norm by benchmarking three common p values and report results in Table VIII.

To allow a clearer comparison between different settings, we repeat the evaluation for different values of n (1, 2, 3) for layer-wise pruning ($\mathcal{O}_2$), where $n = 1$ is the recommended configuration for production, as mentioned earlier. Theoretically, varying p values will result in different performance penalties and detection accuracies as they decide how the L_p-norm is calculated. In our experiments, this is especially visible in the $n = 3$ case where $p = 2$ brings an extra overhead of 20.78% compared to $p = \infty$, but it also also offers the best OOB detectability in terms of all FP_{norm}, FN_{ae}, and FN_{ud}. However, this observation may not be universally true for users protecting their own models. Thus, we recommend $p = 2$ as the default value, and users can perform similar benchmarks following our setup to optimize this value for their scenarios based on their tradeoff preferences. In addition to Table VIII, supplemental data on the project website [2] report consistent findings for other models.

Answer to RQ2: Layer-wise selection ($\mathcal{O}_2$) reduces overhead with a reasonable tradeoff of AE detection. It also alleviates FP_{norm}, by enhancing DNN robustness. Quantization ($\mathcal{O}_1$) reasonably reduces the cost while incurring relatively trivial loss in accuracy and OOB detection. Users can consider to set $p = 2$ as the default setting for $\mathcal{O}_3$, particularly when $\mathcal{O}_2$ is enabled to hook only the last layer.

TABLE IX: Overhead and detection capabilities of OBSAN with all optimizations enabled as recommended.

OBSAN variant	Model	Infer. time (ms)		OBSAN Overhead	FP_{norm} ratio	FN_{ae} ratio	$FN_{pb/ud}$ ratio
		Vanilla	OBSAN				
FOBSAN w/ opt.	ResNet50	1.22	2.19	79.51%	1.40%	0.20%	49.79%
	GoogLeNet	3.79	3.12	-17.68%	2.41%	0.00%	26.02%
	DenseNet121	2.65	4.80	81.13%	1.21%	5.11%	21.43%
	Average	**2.55**	**3.37**	**47.65%**	**1.67%**	**1.77%**	**32.41%**
FOBSAN w/ opt.	ResNet50	1.22	0.82	-32.79%	6.31%	0.00%	65.79%
	GoogLeNet	3.79	1.67	-55.94%	9.38%	0.00%	75.96%
	DenseNet121	2.65	2.28	-13.96%	4.72%	9.64%	73.62%
	Average	**2.55**	**1.59**	**-34.23%**	**6.80%**	**3.21%**	**71.79%**

Combining Optimizations. We also explore combining optimizations $\mathcal{O}_{1-3}$ as it is common to enable all (non-conflict) optimizations for C/C++ compilers. Our optimizations are designed to be compatible with each other, and we assess the overhead and detection accuracy of OBSAN after enabling all optimizations using the best settings discussed above in Table IX. In short, the overhead is on average 48% for FOBSAN and -34% for BOBSAN, with reasonably high OOB detectability. We interpret the results as promising, illustrating that OBSAN achieves a proper balance between speed and detectability, and has high applicability in production usage.

Recommandation. In sum, we suggest enabling all optimizations simultaneously with recommended configurations whenever possible in production. In particular,

FOBSAN. We recommend enabling quantization ($\mathcal{O}_1$) and layer-wise pruning ($\mathcal{O}_2$) at $\gamma = 0.2$; this shall effectively reduce the performance overhead without notably impeding AE detection, and maintain a reasonable level of FN_{pb}. Moreover, Table IV shows that, when enabling full FOBSAN, the ResNet50 executable is 169% slower than the executable without protection. And with the recommended optimizations enabled, the overhead is reduced to 79.51% (Table IX). Similar conclusions are drawn for the other two models.

BOBSAN. We recommend enabling quantization ($\mathcal{O}_1$), checks pruning until the last layer ($\mathcal{O}_2$), and setting $p = 2$ ($\mathcal{O}_3$). This combo should offer the lowest overhead with reasonably strong OOB protection, and users can further tune optimization parameters in accordance with their use cases and tradeoff preferences. As in Table IX, when enabling this optimization combo, the inference time of ResNet50, GoogLeNet, DenseNet121 executables are only -32.79%, -55.94%, and -34.23% of the executables without protection, respectively. Simultaneously, BOBSAN facilitates highly accurate detectability of AEs, i.e., both ResNet50 and DenseNet121 cases have zero FN_{ae}.

Furthermore, we mentioned the use of a *hybrid* OBSAN in Sec. VII, namely integrating both FOBSAN and BOBSAN into the same executable for extra protection; our optimizations are also compatible with this setup. Soon in Sec. IX-A, we compare hybrid and non-hybrid OBSAN's effectiveness in mitigating query-based AE generation attack, where all optimizations are enabled as recommended above. We report

that with hybrid OBSAN, most attacks (on average 77%) can be mitigated, whereas non-hybrid OBSAN (using either FOBSAN or BOBSAN) shows less desirable effectiveness.

In our fuzz testing application (Sec. IX-B), we anticipate to fully instrument each neuron with FOBSAN, as FOBSAN checks can provide neuron coverage information as feedback to guide fuzzing. This is comparable to logging branch or basic block coverage in fuzzing C/C++ executables. In that application, we only enable quantization which should considerably accelerate the fuzzing process. Moreover, accuracy is not a concern in fuzz testing, so only the "neuron coverage" feedback is obtained from FOBSAN.

IX. RQ3: Downstream Security Applications

To answer **RQ3**, we assess OBSAN's ability over two security applications: inhibiting online AE generation (Sec. IX-A) and guiding feedback-driven fuzzing (Sec. IX-B).

TABLE X: Preventing online AE generation.

Scenario	Configuration	#AE	Infer. time (ms)	#queries per seed	Attack success rate
Default	w/o OBSAN	97.60	1.22	2.17	99.59%
	w/ FOBSAN	8.49	2.19	19.68	8.66%
	w/ BOBSAN	96.52	0.82	4.64	98.49%
	w/ HOBSAN	5.25	3.03	20.92	5.38%
Sophisticated	w/o OBSAN	62.41	1.22	127.26	63.68%
	w/ FOBSAN	56.17	2.19	130.52	57.32%
	w/ BOBSAN	30.98	0.82	146.05	31.61%
	w/ HOBSAN	28.03	3.03	139.13	28.60%

A. Mitigating Query-Based Online AE Generation

Setup. As a common assumption in query-based online AE generation, we assume that the DNN executable is a *blackbox* to an adversary. The attackers cannot access the internals of this DNN executable. To generate AEs to exploit the DNN executable, attackers would launch query-based blackbox AE generation, interacting with the target DNN model and mimicking how a remote DNN API is exploited via queries. Here, attackers can feed their crafted inputs to the DNN executable and observe the model prediction outputs [29, 59, 78, 95, 97]. For this attack, we employ a SOTA AE generation method to simulate a realistic attacker [10].

As in Table X, we consider two representative adversary behaviors: In the **default** scenario, the attacker is configured with a higher perturbation budget ($\epsilon = 0.3$) and a lower number of queries per seed (50). In the **sophisticated** scenario, the attacker decides to use a low perturbation budget, which means to mutate images in a finer-grained, incremental manner. AEs are expected to manifest better visual quality with such a low budget setting. Particularly, they use a budget $\epsilon = 0.035$, but allow more queries per seed (500). We assess three OBSAN settings: FOBSAN, BOBSAN, and the hybrid OBSAN (termed HOBSAN), where HOBSAN inserts both variants in a DNN executable. We configure HOBSAN to issue an alert if either FOBSAN or BOBSAN alerts. For each setting, we also enable all applicable optimizations for OBSAN in accordance with our recommendations in Sec. VIII-B.

Extending Blackbox AE Generation. For this application, we configure DNN executable such that it refrains from responding to attacker's query whenever OBSAN alerts. Accordingly, to *mimic a practical attacker who may be aware of the usage of* OBSAN *in the executable*, we extend the employed SOTA blackbox AE generation algorithm; a schematic view has been given in Fig. 3 in Sec. III, such that whenever the attacker receives no response, they "backtrack" by reverting the most recently-performed mutation and performing random mutation.

Results. We randomly select 100 images from the CIFAR-10 test set as seeds. For each seed, we allow the AE generation algorithm to issue queries until the allowed numbers of queries (either 50 or 500) are exhausted. We launch each experiments for five times and average the results. Table X reports the evaluation results. To clarify, two seeds are excluded as they already trigger mis-predictions, given an initial accuracy of 98%, which we deem reasonable under this setting. In the default scenario, the SOTA AE generation algorithm is highly effective; it can generate AEs from nearly all seeds within 50 queries ($\frac{97.60}{98} = 99.59\%$ success rate, 2.17 queries per seed on average). Nevertheless, after enabling FOBSAN, we find that a much lower number of 8.49 AEs (8.66% success rate, a 91% decrease) are successfully generated, and the average number of queries required to generate an AE over a seed is prolonged (19.68 queries, a 9x increase). For BOBSAN, however, it only provides negligible protection against this attack scenario, although it still slightly postpones the generation of AEs (2x queries per seed). HOBSAN gives the best protection in this scenario, successfully defending against 95% of the attacks. The fact that HOBSAN provides better protection than FOBSAN alone indicates that FOBSAN and BOBSAN are respectively sensitive to slightly different sets of AEs. In the "sophisticated" scenario, the AE generation algorithm only successfully generates AEs for 63.7% of the seeds, but with generally improved visual quality. Under this setting, FOBSAN, in contrast to its prior performance, only mitigates about 10% of the attacks, while BOBSAN blocked 52%. HOBSAN further lowers the attack success rate, giving a protection effectiveness of 57% against sophisticated attacks.

Readers may be curious about the difference between FOBSAN and BOBSAN in the default scenario. In this scenario, the attacker makes relatively heavy mutations to the seed inputs, causing the mutation results to resemble perception-tweaked inputs in earlier experiments, which FOBSAN is better at detecting; in the sophisticated scenario, on the contrary, mutations are more subtle and can cause the AEs to be more similar to undefined yet natural inputs. Careful readers may further notice that BOBSAN also produces satisfactory results on detecting white-box AEs generated with the same perturbation budget (Table IV). White-box AEs are generated with knowledge of gradients, which BOBSAN also uses for OOB detection; this makes BOBSAN sensitive to these AEs. The blackbox AEs here, however, are generated without gradients, bypassing BOBSAN's sensitivity to gradient-based attacks.

B. DNN Executable Feedback-Driven Fuzzing

Motivation. We setup another downstream application, mimicking a typical in-house fuzzing scenario on DNN executables. While there exists a large number of previous literatures on DNN fuzzing, they primarily target on DNN models running on DL frameworks like TensorFlow. Note that it is *insufficient* to only fuzz DNN models in DL frameworks, as DNN executables may exhibit inconsistent behaviors comparing with the same models running on DL frameworks, due

TABLE XI: Boosting DNN executable fuzzing with FOBSAN. The "greybox" setting denotes the DNN executable with FOBSAN injected, whereas the "blackbox" setting denotes the DNN executable w/o FOBSAN.

	Model	#mis-predictions	%neurons covered
Blackbox	ResNet50	4268	18.48%
	GoogLeNet	4513	16.10%
	DenseNet121	2550	39.00%
Greybox	ResNet50	44132	21.29%
	GoogLeNet	49550	17.91%
	DenseNet121	32027	44.09%

to floating number precision losses or potential bugs in DL compilers [42, 71]. As a result, we assume it is necessary for users to fuzz the compiled DNN executables and detect its attack surface under AEs. Blackbox fuzzing randomly mutates inputs. Greybox fuzzing mutates inputs under coverage feedback. Fuzzing software executables [100] is often done by intercepting compilation and inserting counters to record branch coverage. Accordingly, FOBSAN can provide neuron-wise coverage, enabling greybox fuzzing in DNN executable.

Setup. We assume that the DNN model is compiled by TVM into executables with FOBSAN inserted. We use a slightly extended version of FOBSAN to adapt to the needs of fuzzing: While the original FOBSAN is designed to be "trained" with training datasets to compare neuron activations and detect abnormal inputs, the fuzzing task's goal is to discover *mis-predictions* that can reveal the internal deficiencies of DNN models. Therefore, we drop the comparison between neuron activations and their reference data (and thus the need for training before using), and instead compare the neuron outputs (scaled to $[0, 1]$) with a manually set threshold; if neuron outputs are above the threshold, we mark the neuron as *covered*. In this experiment, we set this threshold to 0.9, a relatively high value to require a neuron being activated to a great extent in order to count as a "coverage". As noted in **Usage** of Sec. VIII-B, since we do not need an accurate OOB score in this application, we enable quantization to harvest reduction of FOBSAN's overhead. We however disable pruning because we need the complete information on which neurons have been covered. We view this as a practical setting that can be followed in daily usage of FOBSAN to fuzz DNN executables. When mutating DNN inputs, we use mutation methods proposed in [61, 80, 92]. We use executables of all three models in this evaluation.

Results. We let each of the 6 experiment run for six hours and collect the results, as reported in Table XI. After each task is finished, we measure the number of triggered mis-predictions (**#mis-predictions**) and overall percentage of covered neurons (**%covered neurons**). We also report that this evaluation does not result in any crashes, indicating that memory exploitation opportunities for DNN executables may be limited. A great number of mis-predictions found indicates that fuzz testing is of high effectiveness, as per the stated goal of the task earlier. In contrast, the percentages of covered neurons are not particularly high and the gaps between blackbox and greybox settings are less than 10%. While this can be accounted by the adoption of a high threshold to count "neuron coverage," it should not be a significant indicator of fuzzing performance. We find that under the feedback of FOBSAN, the fuzzing campaign becomes far more effective than that of the blackbox setting. In greybox, the numbers of triggered mis-predictions for all three models exceed 10 times higher than those of the blackbox settings. We interpret the results as highly promising and are consistent across all three popular models.

Answer to RQ3: OBSAN facilitates representative downstream applications to secure real-world usage of DNN executables and enable feedback-driven fuzz testing of DNN executables.

X. DISCUSSION

A. Comparison with Existing Approaches

Many methods have been proposed to protect DNN models against OOB inputs. However, various obstacles prevent them from being applied to protect DNN executables. Holistically, they are implemented at the DNN model level, which narrows down the range of DNN executables that they may be able to protect (e.g., due to incompatibilities between DL frameworks; see below). Moreover, we see it as a major trend that existing works emphasize effectiveness more than efficiency: None of our reviewed representative works (see below) report their performance overhead. We note that high overhead deems an obstacle for production adoption by DNN executables. Below, we review recent and representative works for a conceptual-level comparison with OBSAN. Then, we conduct an empirical comparison from both the accuracy and overhead perspectives.

Methodological Comparison. At this step, we review recent works on AE and OOB inputs detection and compare the technical differences between OBSAN and them. To allow a more insightful review, we classify visible works in this field into five categories according to the methodology they employ for anomalous input detection: coverage-, classification-, reclassification-, gradient-, and statistics-based.

① Coverage-based works [46, 61, 85] borrow the concept of coverage from fuzzing (as explained in Sec. III) and build on the assumption that OOB inputs cause increases in coverage metrics. They then use these metrics to detect undesired inputs. Conceptually, they are similar to FOBSAN and can usually be implemented on DNN executables; we include this class of works in our later experimental comparison. Nevertheless, methods implementable for DNN executables may not entail real-world suitability. For example, they may require the user to provide a malicious training dataset [85] or may require unsupported operations for common DL compilers (e.g., set operations); we exclude methods with such requirements (e.g., [85]) in our empirical comparison considering the adoption difficulty.

② Classification-based works [48, 74, 85] train new auxiliary models in advance for each protected DNN model; to use their methods, we need to somehow integrate the auxiliary models into the to-be-shipped DNN executable for OOB input detection. As noted in Sec. III, DL compilers expect well-trained DNN models as input and generally do not support training. This makes the said methods hard to implement as a standalone compiler pass, besides the high time cost that may be incurred for training; these factors discourage adoption. To clarify, though FOBSAN, as mentioned in Sec. V, also has an initialization phase, we do *not* train new models (only feeding each training sample to the DNN executable for a forward pass to decide OOB thresholds) and thus is compatible with

DL compilers. To include this class of works in our empirical comparison, we workaround the compatibility issue by training a set of auxiliary models externally and compiling them into DNN executables.

③ Reclassification-based works [40, 41, 107] assume that adversarial features are usually subtle and can be "muted" by modifying the inputs; re-feeding the modified input to the protected model and comparing the classification results before and after allows anomalous inputs to be detected. OBSAN is different from these works in two aspects. First, OBSAN does not depend on the nature of adversarial features; instead, it focuses OOB, a generalized concept that encompasses both adversarial and "non-adversarial" features, e.g., undefined inputs. Second, OBSAN provides floating point OOB scores while these methods only output stepwise or binary scores; they will be harder to tune and less informative to analyze than OBSAN, as laid out in Sec. IV. Below, we compare OBSAN with these works empirically. That said, we still note that the premise on adversarial features may be rather strong and can limit the detection to a specific class of adversarial attacks [107].

④ Gradient-based works [28, 41, 45] make use of the gradient information during inference, for example to extract a compressed representation of the network's inference process [28, 45] or to transform the input for later use such as reclassification [41]. As explained in Sec. IV-B, BOBSAN also falls into this category. In terms of computational complexity, gradient calculation and backward propagation are typically not fast, e.g., they may require additional large matrix multiplications. Moreover, different ways to derive gradients (e.g., with respect to different variables) and to transform them into OOB metrics can lead to distinct overhead and detection capability. BOBSAN is currently designed with both speed and detection capability into consideration; we launch empirical comparison between our implementation with these methods below.

⑤ Statistics-based works [17, 39, 43, 82] calculate statistical metrics from the activations or classification outputs of the protected DNN model and use them to detect OOB inputs. Typically, this class of methods requires to compute empirical mean, variance, etc. on the training set during initialization [39, 82], which can be more costly than OBSAN's two variants (see Sec. V; BOBSAN does not need initialization). Below, we empirically assess this category of works about accuracy and overhead. Some heavyweight works also require modifying the training and inference processes of the protected models [17]; we deem them expensive and technically challenging, if at all feasible, for DNN executables.

Empirical Comparison. We explore the technical feasibility of (re-)implementing existing works, and narrow down to six works, which we believe are the most representative and well-performing cases. We implement and run experiments on all three DNN models for these six works: DeepGauge [46], DLA [74], ANR [40], ODIN [41], GradNorm [28], and ViM [82]. A summary of them is in Table XII; we consider them representative for a wide range of methodologies. These works are divided into two groups, "vision-based" and "semantics-based", to compare with either FOBSAN or BOBSAN. We select the OBSAN variant against which a work should be compared based on whether or not it is able to detect undefined (UD) inputs: as noted in Sec. III, UD inputs are generally inputs from outside the trained DNN model's knowledge and are not maliciously crafted. In other words, there are no "visual artifacts" in them to be picked up. Holistically, vision-based OOB detectors, as listed in Table XII, rely on the the artifacts' vision patterns (e.g., noise-like pixel patterns) to detect OOB inputs. It is easy to see that UD inputs do not contain such noise-like patterns, and cannot be detected by vision-based OOB detectors (we confirmed with preliminary experiments). Hence, for vision-based methods, we compare them with FOBSAN instead of BOBSAN.

We also compare with three representative works marked as "semantics-based" in Table XII. They essentially analyze whether new (unexpected) semantics appear in an input to decide if it was anomalous, thereby fitting to detect AE and UD inputs. Nevertheless, our tentative experiments show that perceptual broken (PB) inputs can hardly be detected by them. Therefore, we compare these methods with BOBSAN.

For each work, we run benchmarks to collect its inference time, FP_{norm}, FN_{ae}, together with either FN_{pb} (for the first three works compared with FOBSAN) or FN_{ud} (for the rest, compared with BOBSAN). To offer a fair comparison, we do not enable optimizations for OBSAN during these experiments. We report the experimental results in Table XIII (for comparison against FOBSAN) and Table XIV (for BOBSAN).

When focusing only on benign and AE inputs, we see that OBSAN is not the best performing method in terms of detection accuracy; ODIN has the lowest FP_{norm} and FN_{ae} ratios. However, ODIN is significantly slower, with overhead on average 9.64x and 17.96x those of FOBSAN and BOBSAN, respectively; this is because it backpropagates through the entire protected model to obtain gradients w.r.t. the input, and uses this information to derive an augmented input which will be fed again to the protected model, increasing the computational complexity to a level unsuitable for use by DNN executables in production.

In each of the two comparison groups, we also identify DLA and GradNorm as the respective fastest methods in their groups. Nevertheless, DLA is difficult to adopt in practice, as it requires training a series of auxiliary classifier models, which is not supported by DL compilers and also very time-consuming (55 minutes on average, compared to FOBSAN's initialization time of about 2 minutes, as reported in Sec. VIII); GradNorm, on the other hand, has no extra requirements but is not as accurate as OBSAN, particularly on large models. GradNorm has comparable results with BOBSAN on GoogLeNet, as reflected by the FP_{norm} and FN_{ae} columns of the GoogLeNet comparison setting in Table XIV. We note that GoogLeNet is relatively small and has the fewest parameters among all three models (see Table II). The FN_{ae} of GradNorm on other two large models are much higher than BOBSAN and the FN_{ud} is almost trivial. When PB and UD inputs are further considered, we observe that DeepGauge and ViM have the best FN_{pb} and FN_{ud} ratios in their respective groups and also comparable execution overhead to OBSAN, but they do not perform well in terms of FP_{norm} and FN_{ae}. In sum, we conclude from the experiments that both FOBSAN and BOBSAN strike a balance between runtime overhead, ensuring normal inference for benign inputs (reasonably low FP_{norm}), and achieving encouraging detection capabilities for OOB inputs (FN_{ae}, FN_{pb}, FN_{ud}).

TABLE XII: A list of existing works empirically compared. We spent considerable engineering efforts over most of them to identify workarounds and make them compatible with DL compilers. See our released code for their extensions [2].

Work	Methodology Category	Logical category	Compared with
DeepGauge	Coverage	Vision-based	FOBSAN (AE + PB)
DLA	Classification	Vision-based	FOBSAN (AE + PB)
ANR	Reclassification	Vision-based	FOBSAN (AE + PB)
ODIN	Reclassification + Gradient	Semantics-based	BOBSAN (AE + UD)
GradNorm	Gradient	Semantics-based	BOBSAN (AE + UD)
ViM	Statistics	Semantics-based	BOBSAN (AE + UD)

TABLE XIII: Comparison between FOBSAN and other works.

Model	Method	Infer. time (ms)	FP_{norm} ratio	FN_{ae} ratio	FN_{pb} ratio
ResNet50	(None)	1.22	-	-	-
	DeepGauge	3.35	52.62%	0.00%	0.00%
	DLA	1.40	20.46%	0.00%	77.55%
	ANR	4.79	62.92%	86.23%	42.49%
	FOBSAN	3.28	1.11%	2.35%	10.57%
GoogLeNet	(None)	3.79	-	-	-
	DeepGauge	5.84	31.55%	0.00%	0.00%
	DLA	3.91	16.3%	0.00%	67.63%
	ANR	8.12	80.75%	63.15%	58.81%
	FOBSAN	5.75	2.52%	0.00%	0.06%
DenseNet121	(None)	2.65	-	-	-
	DeepGauge	6.54	38.65%	0.00%	0.00%
	DLA	2.75	17.87%	0.00%	16.91%
	ANR	6.06	65.11%	74.48%	63.53%
	FOBSAN	6.45	1.27%	0.01%	0.32%

TABLE XIV: Comparison between BOBSAN and other works.

Model	Method	Infer. time (ms)	FP_{norm} ratio	FN_{ae} ratio	FN_{ud} ratio
ResNet50	(None)	1.22	-	-	-
	ODIN	79.66	1.55%	0.00%	96.58%
	GradNorm	1.25	4.11%	20.76%	82.61%
	ViM	1.80	13.46%	8.60%	24.99%
	BOBSAN	1.30	6.11%	0.01%	65.36%
GoogLeNet	(None)	3.79	-	-	-
	ODIN	42.90	3.67%	0.00%	97.96%
	GradNorm	3.75	6.46%	0.00%	78.61%
	ViM	3.78	10.55%	57.29%	75.79%
	BOBSAN	4.36	9.17%	0.00%	77.78%
DenseNet121	(None)	2.65	-	-	-
	ODIN	26.69	2.29%	0.22%	91.79%
	GradNorm	2.65	3.89%	11.04%	88.48%
	ViM	2.70	14.71%	0.06%	28.45%
	BOBSAN	2.64	6.69%	0.02%	55.02%

B. Extensibility

Migration to Other DL Compilers and Kernel Libraries. As shown in Sec. IV, the design of OBSAN is orthogonal to particular DL compiler frameworks. In the current research, we implement OBSAN on TVM given its popularity and adoption in real-world production scenarios. We anticipate that the current implementation of OBSAN can be smoothly migrated to other DL compilers like Glow [68], which also generate DNN executable files or shared objects from high-level DNN descriptions. The migration demands extra, non-trivial engineering efforts, but should not incur new research challenges. Users can extend our codebase [2] for the migration.

Migration to Other Artechitures. The current implementation of OBSAN instruments DNN executables running on CPUs of 64-bit x86 platforms. We clarify that recent DL compiler testing works also mainly configure DL compilers in the same setting — generating executables running on CPUs of 64-bit x86 platforms [42, 71, 91]. This seems to be the most mature setup for today's DL compilers. However, in contrast to C/C++ sanitizers which are frequently architecture specific [33, 70], the core techniques of OBSAN are *platform independent*. Therefore, it is reasonable to assume that porting the current CPU-based sanitizer implementation to other architectures (e.g., GPUs) poses no new research challenges.

XI. Related Work

We have discussed existing literatures on detecting OOB inputs in Sec. X. Below, we review related work on optimizing sanitization techniques. Existing works have proposed program analysis techniques to flag and remove redundant sanitizer checks, for example, removing an unnecessary array bound check by confirming that the value ranges of an array index are always valid [15, 24, 54, 76, 94]. Most existing works focuses on launching specific analysis to remove redundant checks inserted in C/C++ executables, which cannot be directly extended to optimize OBSAN. ASAP [81] observes that sanitizer checks on the hot paths often contribute less important to vulnerability detectability, given that (unknown) vulnerabilities mostly reside in cold paths that are not fully tested. Shaving those less important checks on hot paths, however, reduce overhead. Our pruning ($\mathcal{O}_2$ and $\mathcal{O}_4$) also identify neurons/layers that are less important, and prune their accompanied OBSAN checks. [101, 103] identify redundant checks using static analysis or heuristics. For instance, in case two checks are found to repeatedly check the same array index in a sequential program, the second check may be removable without undermining security. Nevertheless, it is obscure to identify "redundant" OBSAN checks, given that DNN layers/neurons are hardly semantics-identical.

Conclusion

With the promising trend of compiling DNN models into executables, security hardening over DNN executables is yet unavailable. We propose OBSAN, a sanitization technique capturing OOB behavior of DNN executables. We demonstrate the high OOB detectability of OBSAN, and with various optimization applied, we reduce its runtime overhead to a low level, promoting its usage in production. We present two downstream applications and discuss its extensibility.

References

[1] Onnx runtime. https://onnxruntime.ai/.
[2] OBSAN. https://sites.google.com/view/oob-sanitizer/.
[3] Github issue: Onnx support for adaptivemax/avgpool. https://github.com/pytorch/pytorch/issues/5310, 2018.
[4] Onnx operator definitions. https://github.com/onnx/onnx/blob/main/docs/Operators.md, 2018.
[5] Relay IR. https://tvm.apache.org/docs/arch/relay_intro.html, 2021.
[6] Github issue: [torch.onnx] onnx export failed on adaptive_avg_pool2d because input size not accessible not supported. https://github.com/pytorch/pytorch/issues/74034, 2022.

[7] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. {TensorFlow}: A system for {Large-Scale} machine learning. OSDI, 2016.

[8] Periklis Akritidis, Cristian Cadar, Costin Raiciu, Manuel Costa, and Miguel Castro. Preventing memory error exploits with WIT. IEEE S&P, 2008.

[9] Amazon. Amazon SageMaker Neo uses Apache TVM for performance improvement on hardware target. https://aws.amazon.com/sagemaker/neo/, 2021.

[10] Maksym Andriushchenko, Francesco Croce, Nicolas Flammarion, and Matthias Hein. Square attack: A query-efficient black-box adversarial attack via random search. *Lecture Notes in Computer Science*, page 484–501, 2020.

[11] Sajid Anwar, Kyuyeon Hwang, and Wonyong Sung. Structured pruning of deep convolutional neural networks. *JETC*, 13(3):1–18, 2017.

[12] Riyadh Baghdadi, Jessica Ray, Malek Ben Romdhane, Emanuele Del Sozzo, Abdurrahman Akkas, Yunming Zhang, Patricia Suriana, Shoaib Kamil, and Saman Amarasinghe. Tiramisu: A polyhedral compiler for expressing fast and portable code. CGO, 2019.

[13] Yu Bai, Song Mei, Huan Wang, and Caiming Xiong. Don't just blame over-parametrization for over-confidence: Theoretical analysis of calibration in binary classification. ICML, 2021.

[14] David Bau, Jun-Yan Zhu, Hendrik Strobelt, Agata Lapedriza, Bolei Zhou, and Antonio Torralba. Understanding the role of individual units in a deep neural network. *Proceedings of the National Academy of Sciences*, 117(48):30071–30078, 2020.

[15] Rastislav Bodík, Rajiv Gupta, and Vivek Sarkar. ABCD: eliminating array bounds checks on demand. PLDI, 2000.

[16] Gary Bradski and Adrian Kaehler. Opencv. *Dr. Dobb's journal of software tools*, 3:120, 2000.

[17] Senqi Cao and Zhongfei Zhang. Deep Hybrid Models for Out-of-Distribution Detection. pages 4733–4743.

[18] Anirban Chakraborty, Manaar Alam, Vishal Dey, Anupam Chattopadhyay, and Debdeep Mukhopadhyay. Adversarial attacks and defences: A survey. *arXiv preprint arXiv:1810.00069*, 2018.

[19] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. {TVM}: An automated end-to-end optimizing compiler for deep learning. OSDI, 2018.

[20] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. Binaryconnect: Training deep neural networks with binary weights during propagations. 2015.

[21] Samet Demir, Hasan Ferit Eniser, and Alper Sen. DeepSmartFuzzer: Reward guided test generation for deep learning. *arXiv preprint arXiv:1911.10621*, 2019.

[22] Xiaoning Du, Xiaofei Xie, Yi Li, Lei Ma, Yang Liu, and Jianjun Zhao. A quantitative analysis framework for recurrent neural network. ASE, 2019.

[23] Sainyam Galhotra, Yuriy Brun, and Alexandra Meliou. Fairness testing: testing software for discrimination. In *ACM ESEC/FSE*, pages 498–510. ACM, 2017.

[24] Rigel Gjomemo, Phu H Phung, Edmund Ballou, Kedar S Namjoshi, VN Venkatakrishnan, and Lenore Zuck. Leveraging static analysis tools for improving usability of memory error sanitization compilers. In *2016 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, pages 323–334. IEEE, 2016.

[25] Song Han, Jeff Pool, John Tran, and William Dally. Learning both weights and connections for efficient neural network. 2015.

[26] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016.

[27] Xiaofei He, Wei-Ying Ma, and Hong-Jiang Zhang. Learning an image manifold for retrieval. In *Proceedings of the 12th annual ACM international conference on Multimedia*, pages 17–23, 2004.

[28] Rui Huang, Andrew Geng, and Yixuan Li. On the Importance of Gradients for Detecting Distributional Shifts in the Wild. In *Advances in Neural Information Processing Systems*, volume 34, pages 677–689. Curran Associates, Inc., 2021.

[29] Zhichao Huang and Tong Zhang. Black-box adversarial attack with transferable model-based embedding. ICLR, 2019.

[30] Nargiz Humbatova, Gunel Jahangirova, Gabriele Bavota, Vincenzo Riccio, Andrea Stocco, and Paolo Tonella. Taxonomy of real faults in deep learning systems. ICSE, 2020.

[31] Texas Instruments. The AM335x microprocessors support TVM. https://software-dl.ti.com/processor-sdk-linux/esd/docs/latest/linux/Foundational_Components/Machine_Learning/tvm.html, 2021.

[32] Animesh Jain, Shoubhik Bhattacharya, Masahiro Masuda, Vin Sharma, and Yida Wang. Efficient execution of quantized deep learning models: A compiler approach, 2020.

[33] Yuseok Jeon, WookHyun Han, Nathan Burow, and Mathias Payer. {FuZZan}: Efficient sanitizer metadata design for fuzzing. USENIX ATC, 2020.

[34] Qing Jin, Linjie Yang, and Zhenyu Liao. Towards efficient training for neural network quantization. *arXiv preprint arXiv:1912.10207*, 2019.

[35] Qing Jin, Linjie Yang, and Zhenyu Liao. Adabits: Neural network quantization with adaptive bit-widths. CVPR, 2020.

[36] Hoki Kim. Torchattacks: A pytorch repository for adversarial attacks, 2020.

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

[38] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. CGO, 2004.

[39] Kimin Lee, Kibok Lee, Honglak Lee, and Jinwoo Shin. A Simple Unified Framework for Detecting Out-of-Distribution Samples and Adversarial Attacks. NeurIPS, 2018.

[40] Bin Liang, Hongcheng Li, Miaoqiang Su, Xirong Li, Wenchang Shi, and Xiaofeng Wang. Detecting Adversarial Image Examples in Deep Neural Networks with Adaptive Noise Reduction. 18(1):72–85.

[41] Shiyu Liang, R Srikant, and Yixuan Li. ENHANCING THE RELIABILITY OF OUT-OF-DISTRIBUTION IMAGE DETECTION IN NEURAL NETWORKS. page 27.

[42] Jiawei Liu, Yuxiang Wei, Sen Yang, Yinlin Deng, and Lingming Zhang. Coverage-guided tensor compiler fuzzing with joint ir-pass mutation. OOPSLA, 2022.

[43] Weitang Liu, Xiaoyun Wang, John Owens, and Yixuan Li. Energy-based Out-of-distribution Detection. NeurIPS, 2020.

[44] Yizhi Liu, Yao Wang, Ruofei Yu, Mu Li, Vin Sharma, and Yida Wang. Optimizing {CNN} model inference on cpus. USENIX ATC, 2019.

[45] Julia Lust and Alexandru P Condurache. GraN: An efficient gradient-norm based detector for adversarial and misclassified examples. page 6.

[46] Lei Ma, Felix Juefei-Xu, Jiyuan Sun, Chunyang Chen, Ting Su, Fuyuan Zhang, Minhui Xue, Bo Li, Li Li, Yang Liu, et al. DeepGauge: Comprehensive and multi-granularity testing criteria for gauging the robustness of deep learning systems. ASE, 2018.

[47] Lingxiao Ma, Zhiqiang Xie, Zhi Yang, Jilong Xue, Youshan Miao, Wei Cui, Wenxiang Hu, Fan Yang, Lintao Zhang, and Lidong Zhou. Rammer: Enabling holistic deep learning compiler optimizations with rtasks. OSDI, 2020.

[48] Shiqing Ma and Yingqi Liu. NIC: Detecting adversarial samples with neural network invariant checking. NDSS, 2019.

[49] Aleksander Madry, Aleksandar Makelov, Ludwig Schmidt, Dimitris Tsipras, and Adrian Vladu. Towards deep learning models resistant to adversarial attacks. *arXiv preprint arXiv:1706.06083*, 2017.

[50] Clang User's Manual. Undefined behavior sanitizer. https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html, 2019.

[51] Pavlo Molchanov, Stephen Tyree, Tero Karras, Timo Aila, and Jan Kautz. Pruning convolutional neural networks for resource efficient inference. *arXiv preprint arXiv:1611.06440*, 2016.

[52] Timothy Prickett Morgan. INSIDE FACEBOOK'S FUTURE RACK AND MICROSERVER IRON. https://www.nextplatform.com/2020/05/14/inside-facebooks-future-rack-and-microserver-iron/, 2020.

[53] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. SoftBound: Highly compatible and complete spatial memory safety for C. PLDI, 2009.

[54] George C Necula and Peter Lee. The design and implementation of a certifying compiler. *ACM SIGPLAN Notices*, 33(5):333–344, 1998.

[55] Nvidia. NVVM IR. https://docs.nvidia.com/cuda/nvvm-ir-spec/index.html, 2021.

[56] NXP. NXP uses Glow to optimize models for low-power NXP MCUs. https://www.nxp.com/company/blog/glow-compiler-optimizes-neural-networks-for-low-power-nxp-mcus:BL-OPTIMIZES-NEURAL-NETWORKS, 2020.

[57] OctoML. OctoML leverages TVM to optimize and deploy models. https://octoml.ai/features/maximize-performance/, 2021.

[58] Augustus Odena and Ian Goodfellow. Tensorfuzz: Debugging neural networks with coverage-guided fuzzing. *arXiv preprint arXiv:1807.10875*, 2018.

[59] Nicolas Papernot, Patrick McDaniel, Ian Goodfellow, Somesh Jha, Z Berkay Celik, and Ananthram Swami. Practical black-box attacks against machine learning. AsiaCCS, 2017.

[60] 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. 2019.

[61] Kexin Pei, Yinzhi Cao, Junfeng Yang, and Suman Jana. DeepXplore: Automated whitebox testing of deep learning systems. SOSP, 2017.

[62] Hung Viet Pham, Thibaud Lutellier, Weizhen Qi, and Lin Tan. CRADLE: Cross-backend validation to detect and localize bugs in deep learning libraries. ICSE, 2019.

[63] Hung Viet Pham, Shangshu Qian, Jiannan Wang, Thibaud Lutellier, Jonathan Rosenthal, Lin Tan, Yaoliang Yu, and Nachiappan Nagappan. Problems and opportunities in training deep learning software systems: An analysis of variance. ASE, 2020.

[64] Pytorch. Dense Convolutional Network (DenseNet). https://pytorch.org/hub/pytorch_vision_densenet/, 2021.

[65] Qualcomm. Qualcomm contributes Hexagon DSP improvements to the Apache TVM community, 2020.

[66] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *Acm Sigplan Notices*, 48(6):519–530, 2013.

[67] Kui Ren, Tianhang Zheng, Zhan Qin, and Xue Liu. Adversarial attacks and defenses in deep learning. *Engineering*, 6(3):346–360, 2020.

[68] Nadav Rotem, Jordan Fix, Saleem Abdulrasool, Garret Catron, Summer Deng, Roman Dzhabarov, Nick Gibson, James Hegeman, Meghan Lele, Roman Levenstein, et al. Glow: Graph lowering compiler techniques for neural networks. *arXiv preprint arXiv:1805.00907*, 2018.

[69] Ramprasaath R Selvaraju, Michael Cogswell, Abhishek Das, Ramakrishna Vedantam, Devi Parikh, and Dhruv Batra. Grad-cam: Visual explanations from deep networks via gradient-based localization. ICCV, 2017.

[70] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. AddressSanitizer: A Fast Address Sanity Checker. USENIX, 2012.

[71] Qingchao Shen, Haoyang Ma, Junjie Chen, Yongqiang Tian, Shing-Chi Cheung, and Xiang Chen. A comprehensive study of deep learning compiler bugs. FSE, 2021.

[72] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[73] Dokyung Song, Julian Lettner, Prabhu Rajasekaran, Yeoul Na, Stijn Volckaert, Per Larsen, and Michael Franz. SoK: Sanitizing for security. 2019.

[74] Philip Sperl, Ching-Yu Kao, Peng Chen, Xiao Lei, and Konstantin Böttinger. DLA: Dense-Layer-Analysis for Adversarial Example Detection. pages 198–215. IEEE Computer Society.

[75] Evgeniy Stepanov and Konstantin Serebryany. MemorySanitizer: Fast detector of uninitialized memory use in C++. CGO, 2015.

[76] Yulei Sui, Ding Ye, Yu Su, and Jingling Xue. Eliminating redundant bounds checks in dynamic buffer overflow detection using weakest preconditions. *IEEE Transactions on Reliability*, 2016.

[77] Mukund Sundararajan, Ankur Taly, and Qiqi Yan. Axiomatic attribution for deep networks. ICML, 2017.

[78] Fnu Suya, Jianfeng Chi, David Evans, and Yuan Tian. Hybrid batch attacks: Finding black-box adversarial examples with limited queries. USENIX Security, 2020.

[79] Keke Tang, Dingruibo Miao, Weilong Peng, Jianpeng Wu, Yawen Shi, Zhaoquan Gu, Zhihong Tian, and Wenping Wang. Codes: Chamfer out-of-distribution examples against overconfidence issue. ICCV, 2021.

[80] Yuchi Tian, Kexin Pei, Suman Jana, and Baishakhi Ray. DeepTest: Automated testing of deep-neural-network-driven autonomous cars. ICSE '18, 2018.

[81] Jonas Wagner, Volodymyr Kuznetsov, George Candea, and Johannes Kinder. High system-code security with low overhead. IEEE S&P, 2015.

[82] Haoqi Wang, Zhizhong Li, Litong Feng, and Wayne Zhang. ViM: Out-Of-Distribution with Virtual-logit Matching. CVPR, 2022.

[83] Jiannan Wang, Thibaud Lutellier, Shangshu Qian, Hung Viet Pham, and Lin Tan. EAGLE: Creating equivalent graphs to test deep learning libraries. 2022.

[84] Peisong Wang, Qinghao Hu, Yifan Zhang, Chunjie Zhang, Yang Liu, and Jian Cheng. Two-step quantization for low-bit neural networks. CVPR, 2018.

[85] Run Wang, Felix Juefei-Xu, Lei Ma, Xiaofei Xie, Yihao Huang, Jian Wang, and Yang Liu. FakeSpotter: A Simple yet Robust Baseline for Spotting AI-Synthesized Fake Faces. volume 4, pages 3444–3451.

[86] Shuai Wang and Zhendong Su. Metamorphic object insertion for testing object detection systems. In *ASE*, 2020.

[87] Xi Wang, Nickolai Zeldovich, M Frans Kaashoek, and Armando Solar-Lezama. Towards optimization-safe systems: Analyzing the impact of undefined behavior. SOSP, 2013.

[88] Xiaosong Wang, Yifan Peng, Le Lu, Zhiyong Lu, Mohammadhadi Bagheri, and Ronald M. Summers. Chestx-ray8: Hospital-scale chest x-ray database and benchmarks on weakly-supervised classification and localization of common thorax diseases. CVPR, 2017.

[89] Zan Wang, Ming Yan, Junjie Chen, Shuang Liu, and Dongdi Zhang. Deep learning library testing via effective model generation. FSE, 2020.

[90] Sally Ward-Foxton. Google and Nvidia Tie in MLPerf; Graphcore and Habana Debut. https://www.eetimes.com/google-and-nvidia-tie-in-mlperf-graphcore-and-habana-debut, 2021.

[91] Dongwei Xiao, Zhibo Liu, Yuanyuan Yuan, Qi Pang, and Shuai Wang. Metamorphic testing of deep learning compilers. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 6(1):1–28, 2022.

[92] Xiaofei Xie, Lei Ma, Felix Juefei-Xu, Hongxu Chen, Minhui Xue, Bo Li, Yang Liu, Jianjun Zhao, Jianxiong Yin, and Simon See. Coverage-guided fuzzing for deep neural networks. *arXiv preprint arXiv:1809.01266*, 2018.

[93] Xilinx. Xilinx support TVM on DPU. https://www.xilinx.com/html_docs/xilinx2019_2/vitis_doc/deploying_running.html, 2020.

[94] Zhichen Xu, Barton P Miller, and Thomas Reps. Safety checking of machine code. *ACM SIGPLAN Notices*, 35(5):70–82, 2000.

[95] Jiancheng Yang, Yangzhou Jiang, Xiaoyang Huang, Bingbing Ni, and Chenglong Zhao. Learning black-box attackers with transferable priors and query feedback. 2020.

[96] Jiwei Yang, Xu Shen, Jun Xing, Xinmei Tian, Houqiang Li, Bing Deng, Jianqiang Huang, and Xian-sheng Hua. Quantization networks. CVPR, 2019.

[97] Jianhe Yuan and Zhihai He. Consistency-sensitivity guided ensemble black-box adversarial attacks in low-dimensional spaces. ICCV, 2021.

[98] Xiaoyong Yuan, Pan He, Qile Zhu, and Xiaolin Li. Adversarial examples: Attacks and defenses for deep learning. *IEEE transactions on neural networks and learning systems*, 30(9):2805–2824, 2019.

[99] Tomofumi Yuki, Gautam Gupta, DaeGon Kim, Tanveer Pathan, and Sanjay Rajopadhye. Alphaz: A system for design space exploration in the polyhedral model. In *International Workshop on Languages and Compilers for Parallel Computing*, pages 17–31. Springer, 2012.

[100] Michał Zalewski. American Fuzzy Lop. https://lcamtuf.coredump.cx/afl/, 2021.

[101] Jiang Zhang, Shuai Wang, Manuel Rigger, Pinjia He, and Zhendong Su. {SANRAZOR}: Reducing redundant sanitizer checks in {C/C++} programs. OSDI, 2021.

[102] Mengshi Zhang, Yuqun Zhang, Lingming Zhang, Cong Liu, and Sarfraz Khurshid. DeepRoad: GAN-based Metamorphic Testing and Input Validation Framework for Autonomous Driving Systems. In *ASE*, 2018.

[103] Yuchen Zhang, Chengbin Pang, Georgios Portokalidis, Nikos Triandopoulos, and Jun Xu. Debloating address sanitizer.

[104] Yuhao Zhang, Yifan Chen, Shing-Chi Cheung, Yingfei Xiong, and Lu Zhang. An empirical study on tensorflow program bugs. ISSTA 2018, 2018.

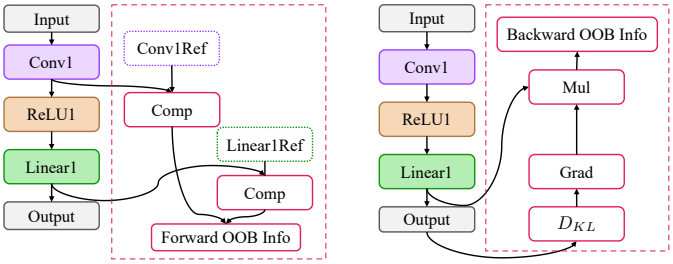
[105] Ritchie Zhao, Yuwei Hu, Jordan Dotzel, Chris De Sa, and Zhiru Zhang. Improving neural network quantization without retraining using outlier channel splitting. ICML, 2019.

[106] Jun-Yan Zhu, Philipp Krähenbühl, Eli Shechtman, and Alexei A Efros. Generative visual manipulation on the natural image manifold. In *European conference on computer vision*, pages 597–613. Springer, 2016.

[107] Fei Zuo and Qiang Zeng. Exploiting the sensitivity of l2 adversarial examples to erase-and-restore. AsiaCCS, 2021.

TABLE XV: Implementations of inverted operators for backward propagation, where x and y are the input and output of each operator/layer, and x^* and y^* are the input and output of each inverted operator/layer.

Operator/Layer	Description	Inverted Implementation
Fully connected layer	$y = Wx + b$. W and b are the parameters.	The inverted layer can be implemented as another fully connected layer $y^* = W^T x^* + b$
Convolution layer	$y = \sum_{kernel} W \star x + b$. W and b are the parameters. $\star$ is the 2D cross-correlation operator [19, 60].	The inverted layer can be implemented as a transposed convolutional layer of parameters W and b.
ReLU	$y = \max(0, x)$. Activation function w/o parameters.	$g = \begin{cases} 1, & x^* > 0 \\ 0, & x^* \leq 0 \end{cases}$ $y^* = gx^*$.
LeakyReLU	$y = \begin{cases} x, & x > 0 \\ \alpha x, & x \leq 0 \end{cases}$ α is a hyperparameter.	$g = \begin{cases} 1, & x^* > 0 \\ \alpha, & x^* \leq 0 \end{cases}$ $y^* = gx^*$.
Max Pooling	Record $y[\ldots, w, h] = \max x[\ldots, w + k[0], h + k[1]]$ and $(i, j) = \arg\max x[\ldots, w + k[0], h + k[1]]$. for each (w, h) pair. k is a 2D kernel.	1) Let $y^*[\ldots, w + k[0], h + k[1]] = 0$. 2) For each (w, h, i, j) pair, set $y^*[\ldots, i, j] = x^*[\ldots, w, h]$.
Average Pooling	$y[\ldots, w, h] = \frac{1}{k[0]} \frac{1}{k[1]} \sum x[\ldots, w + k[0], h + k[1]]$. k is a 2D kernel.	1) Let $g = 0$ and for each (w, h), set $g[\ldots, w : w + k[0], h : h + k[1]]$ $= g[\ldots, w : w + k[0], h : h + k[1]] + \frac{1}{k[0]} \frac{1}{k[1]}$ 2) $y^* = gx^*$.
Sigmoid	$y = \sigma(x) = \frac{1}{1+e^{-x}}$. Activation function w/o parameters.	$y^* = \sigma(x^*)(1 - \sigma(x^*))$
Tanh	$y = tanh(x) = \frac{2}{1+e^{-2x}} - 1$	$y^* = 1 - tanh(x^*)^2$



(a) Instrumentation using FOBSAN. ReLU1 is excluded from OOB checks.

(b) Instrumentation using BOBSAN. Only the last layer is hooked with BOBSAN.

Fig. 7: A simple DNN model instrumented by the two variants of OBSAN with layer-wise checks pruning ($\mathcal{O}_2$) enabled. Subgraphs inserted are marked by dashed line boxes.

APPENDIX A
INSTRUMENTATION EXAMPLES

Fig. 7 shows two concrete examples of OBSAN instrumentation on a simple DNN model, where the attached OBSAN subgraphs are highlighted in dashed line boxes and the original models and prediction results are unaltered.

Instrumenting Models for FOBSAN. To instrument a model to detect forward OOB, the instrumentation pass of OBSAN traverses the model's computational graph in Relay IR and hook each interesting layer with checks which compare the layer's outputs with its reference values, as noted above. Fig. 7(a) shows a simple case of FOBSAN instrumentation where the intermediate outputs of the Conv1 and Linear1 layers are compared with their respective references and the results are then aggregated to produce the final OOB information. The outputs of the ReLU1 layer are omitted from the instrumentation as an optimization (optimization $\mathcal{O}_2$; see Sec. V-B). Since the reference values need to be built using the training data, FOBSAN also introduces two range building components, one for Conv1 (Conv1Ref) and the other for Linear1 (Linear1Ref). These components will be gradually updated in the Record Mode (Fig. 5(b)) and allow FOBSAN to capture and store the required reference values.

Instrumenting Models for BOBSAN. The instrumentation is slightly different to enable the backward variant of OBSAN. The model output layer is traversed first, and the traversal continues towards the input layer, carrying along the gradient information as calculated in Sec. IV-B to finish the backward propagation. Although no reference values are needed for BOBSAN and thus storage requirements are lowered, the calculation of gradients can be computationally expensive as it can involve operations like matrix multiplication. We thus optimize BOBSAN in the sense that only the last n layers are instrumented; see $\mathcal{O}_2$ in Sec. V-B for details. Fig. 7(b) illustrates the instrumentation of a simple DNN model using BOBSAN where only the last layer is included in gradients computation. Our experiments show that with such optimizations, BOBSAN typically offers satisfactory OOB detection capability while maintaining low computation costs.

APPENDIX B
EXTENDING FOR GRADIENT COMPUTATION

To enable backward propagation of DNN executable e for gradient computation, we first implement an inverted operator f_i^{-1} for each operator f_i in e. We then compose each inverted operator from the ending layer (f_l^{-1}; following the order in e) to the starting layer (f_1^{-1}) and build a new inverted DNN $e^* = f_1^{-1} \circ f_2^{-1} \circ \cdots \circ f_l^{-1}$. As a result, the backward propagation can be accomplished by feeding e^* with g; see Eq. 2 for the calculation of g.

Table XV lists operators from the DNNs evaluated in this paper and how the correspondingly inverted operator is implemented. We clarify that these operators are representative and cover operators in most popular DNNs. We also make OBSAN extendable for users to implement new inverted operator (see [2]) which should be straightforward given the gradient computations are rigorously defined math formulas.