MPCDIFF: Testing and Repairing MPC-Hardened Deep Learning Models

Qi Pang[†]
Carnegie Mellon University
qpang@andrew.cmu.edu

Yuanyuan Yuan
HKUST
yyuanaq@cse.ust.hk

Shuai Wang[‡]
HKUST
shuaiw@cse.ust.hk

***Abstract*—Secure multi-party computation (MPC) has recently become prominent as a concept to enable multiple parties to perform privacy-preserving machine learning without leaking sensitive data or details of pre-trained models to the other parties. Industry and the community have been actively developing and promoting high-quality MPC frameworks (e.g., based on TensorFlow and PyTorch) to enable the usage of MPC-hardened models, greatly easing the development cycle of integrating deep learning models with MPC primitives.**

Despite the prosperous development and adoption of MPC frameworks, a principled and systematic understanding toward the correctness of those MPC frameworks does not yet exist. To fill this critical gap, this paper introduces MPCDIFF, a differential testing framework to effectively uncover inputs that cause deviant outputs of MPC-hardened models and their plaintext versions. We further develop techniques to localize error-causing computation units in MPC-hardened models and automatically repair those defects.

We evaluate MPCDIFF using real-world popular MPC frameworks for deep learning developed by Meta (Facebook), Alibaba Group, Cape Privacy, and OpenMined. MPCDIFF successfully detected over one thousand inputs that result in largely deviant outputs. These deviation-triggering inputs are (visually) meaningful in comparison to regular inputs, indicating that our findings may cause great confusion in the daily usage of MPC frameworks. After localizing and repairing error-causing computation units, the robustness of MPC-hardened models can be notably enhanced without sacrificing accuracy and with negligible overhead.

I. INTRODUCTION

A traditional MLaaS workflow involves a data pipeline, which uses a central server that hosts the pre-trained DL model to make predictions. Thus, all data collected by users are sent to the central server in plaintext for use. However, when confidential data is processed by an outsourced DL service, user privacy may be jeopardized [68], [8].

Secure multi-party computation (MPC) allows parties to collaboratively perform computations on data while keeping the data private. Recently, we have seen a prosperous adoption of MPC in securing DL applications: it facilitates making predictions over private data of one party (data provider) using a pre-trained model maintained by another party (model provider) without leaking data or models to the other parties [59], [48]. Industrial giants, including Meta (Facebook), Microsoft, and Alibaba, are actively developing and promoting their MPC frameworks for DL, greatly spurring real-world DL applications with privacy considerations [19], [57], [48].

Despite the significant development of MPC-hardened DL models and MPC frameworks available on the market, it is unclear about their quality and potential attack surfaces. The design of MPC protocols are often very subtle and error-prone. Also, modern MPC frameworks for DL often aim to natively support complex DNN operators, tensor computations, and imperative programming [57], [59]. Various optimization and approximation schemes (e.g., fixed-point value representation [81] and Newton-Raphson approximation [33]) are involved to convert standard computations into MPC-hardened forms. While these efforts substantially reduce the difficulty to integrate DL applications with MPC primitives, they may also increase the difficulty of implementing bug-free MPC frameworks. In fact, our preliminary studies have shown that MPC-hardened DL models may yield deviant outputs comparing to their plaintext versions given identical inputs.

This work designs MPCDIFF, the first testing & repairing tool to detect and fix mis-predictions made by MPC-hardened DL models during the inference stage. MPCDIFF uses feedback-driven differential testing to detect inputs that result in largely deviant outputs between MPC-hardened DL models and their plaintext versions. We show that these inputs are of high (visual) quality comparing with normal inputs, thereby uncovering defects that may cause users of these MPC frameworks substantial confusion in their daily usage. These deviation-triggering inputs may represent practical yet overlooked attack vectors to deceive MPC-hardened models and likely manipulate their predictions. Moreover, MPCDIFF can automatically localize MPC-transformed computation units in DL models that primarily result in the output deviations (due to either truncation errors or non-linear function approximations). We further design techniques to repair the localized defect-causing units, thereby enhancing the robustness and mitigating mis-predictions without incurring much overhead.

Our evaluation encompasses three cutting-edge MPC-hardened DL frameworks, CrypTen [4], TF-Encrypted [3] and PySyft [1], which are all developed by industry companies and actively maintained in the community. We use prominent DL models as the targets, including convolutional neural network (CNN) and multilayer perceptron (MLP) with diverse

[†]Majority of the work is done when Qi Pang was at HKUST.
[‡]Corresponding author.

Network and Distributed System Security (NDSS) Symposium 2024
26 February - 1 March 2024, San Diego, CA, USA
ISBN 1-891562-93-2
https://dx.doi.org/10.14722/ndss.2024.23380
www.ndss-symposium.org

activation functions. MPCDIFF generates $135,000$ mutated inputs in total to test those MPC-hardened models. During approximately 10 days of testing, we detected $1,055$ inputs that resulted in greatly deviant outputs, where the prediction labels are inconsistent between MPC-hardened models and their plaintext versions. With these $1,055$ inputs, we are able to examine the root causes and localize the error-causing computation units. We repair those localized root causes, largely increasing the robustness of MPC-hardened models. About half of the error-triggering inputs are mitigated with a negligible extra overhead (0.02%), and MPCDIFF finds much fewer error-triggering inputs in the repaired models. Developers can use MPCDIFF to test and repair their MPC-hardened models before releasing to end users. In sum, we make the following contributions.

- We for the first time focus on flaws particularly in MPC-hardened DL models. We reveal model inputs that result in deviant prediction outputs, which can cause great confusions or adversarial manipulations in daily usage of MPC-hardened models.
- MPCDIFF uses feedback-driven differential testing to gradually explore inputs that maximize output deviations. MPCDIFF localizes and repairs error-causing computation units in DL models. MPCDIFF incorporates a set of design principles and optimizations to expedite testing and repairing at a low cost.
- Our large-scale evaluation of three popular MPC-hardened DL frameworks exposes a substantial number of error-triggering inputs. We further repair the exposed defects, making them considerably more robust.

We maintain MPCDIFF to benefit future research at [2].

II. PRELIMINARIES

This section introduces how MPC enables secure computation (Sec. II-A), and then explains typical MPC-protected DL scenarios in Sec. II-B. We then present preliminary exploration on attack vectors of MPC-hardened DNN models in Sec. II-C.

A. Secure Multi-Party Computation

Secret sharing is a fundamental MPC primitive that refers to a set of well-designed algorithms that distribute a secret among a group of parties, with each party receiving a share of the secret. The secret can only be reconstructed by combining a certain number of shares; individual shares leak no information on their own. This way, sensitive information can be stored without exposure. This work focuses on studying the widely-used and efficient secret sharing-based MPC protocols. There are also other MPC primitives such as Yao's Garbled Circuit [97], [13], homomorphic encryption [45], [58], and oblivious transfer [51], [55], which are not the focus of our work. We omit their introduction and refer interested readers to this review [43]. We now introduce how arithmetic and binary additive secret sharing form secure DNNs.

Arithmetic and Binary Additive Secret Sharing. Arithmetic additive secret sharing [25] aims to share a scalar l-bit value $x \in \mathbb{Z}_{2^l}$ across parties $p \in \mathcal{P}$, where $\mathbb{Z}_{2^l}$ denotes a ring with 2^l elements. Let the sharing of x be $[\![x]\!] = \{[\![x]\!]_p\}_{p\in\mathcal{P}}$, where $[\![x]\!]_p \in \mathbb{Z}_{2^l}$ represents party p's share of x. Usually, shares are built such that their sum reconstructs the original value x. That is, $x \equiv \sum_{p\in\mathcal{P}} [\![x]\!]_p \mod 2^l$. To share a value x, the parties often generate a pseudorandom zero-share [21] with $|\mathcal{P}|$ random numbers whose sum is 0. The party that possesses the value x adds x to its share and then discards x.

Binary secret sharing [35] operates within the binary field $\mathbb{Z}_2$. A binary secret share, $\langle x\rangle$, of a value x is formed by arithmetic secret shares of the bits of x, setting $2^l = 2$. Each party $p \in \mathcal{P}$ holds a share, $\langle x\rangle_p$, such that $\bigoplus_{p\in\mathcal{P}}\langle x\rangle_p$ yields x. Binary sharing can be obtained from arithmetic sharing, by creating binary secret share, $\langle [\![x]\!]_p\rangle$, of all the bits in $[\![x]\!]_p$. Similarly, arithmetic sharing can be obtained from binary sharing, by computing $[\![x]\!] = \sum_{b=1}^{l} 2^b [\![\langle x\rangle^{(b)}]\!]$, where $\langle x\rangle^{(b)}$ denotes the b-th bit of the binary share $\langle x\rangle$ and l is the total number of bits in the shared secret, $\langle x\rangle$.

1) Secure Computation: Note that well-established arithmetic and binary secret shares have *homomorphic property* [15]: that is, mathematical operations can be directly performed on the encrypted data, whose output can be further converted back to plaintext. Overall, in the context of DL, binary secret sharing often facilitates common operators like rectified linear units and argmaxes. Other operations, including standard matrix multiplications and convolutions, can be implemented using other arithmetic secret sharing forms. Following, we introduce how private computations can be done under the usage of MPC.

Private Addition. Private addition of two arithmetically secret shared values, $[\![z]\!] = [\![x]\!] + [\![y]\!]$, can be implemented by having each party p locally sum his shares of $[\![x]\!]$ and $[\![y]\!]$. That is, each party $p \in \mathcal{P}$ computes $[\![z]\!]_p = [\![x]\!]_p + [\![y]\!]_p$.

Private Multiplication. This operation is trivial (with private addition) if we multiply a secret shared values with public values. Nevertheless, for private multiplication of two secret shared values $[\![x]\!]$ and $[\![y]\!]$, the common practice is to use random Beaver triples [12], which involves extra secret shared values to recast multiplying two private values into decryption then multiplying private values with public values. Beaver triples shift most of the communication and computation cost into a preprocessing phase, which does not require the knowledge of inputs and can be done offline. Here, we illustrate the private multiplication in a two-party scenario. In the offline phase, the parties hold the secret share values $[\![a]\!], [\![b]\!], [\![c]\!]$, such that $c = ab$. Then, in the online phase, the parties compute locally $[\![\alpha]\!] = [\![x]\!] - [\![a]\!]$ and $[\![\beta]\!] = [\![y]\!] - [\![b]\!]$. The parties then reconstruct α, β by exchanging the shares. The secret shared product can be calculated with the Beaver triples: $[\![xy]\!] = [\![c]\!] + \alpha[\![b]\!] + \beta[\![a]\!] + \alpha\beta$.

Linear Functions. Combining the aforementioned private addition and multiplication can easily form linear functions used in conventional DL models, such as the dot products, matrix multiplications, and convolutions.

Non-linear Functions. Non-linear functions in commonly-used DNNs, such as Sigmoid, Tanh, GELU [44], softmax logistic-loss function, and batch normalization, can be *approximated* using private addition and multiplication. For instance, Newton-Rhapson and Householder [33], [47] iterations and Chebyshev polynomials approximation [83] have been adopted in MPC frameworks [4], [3], [1] to approximate private non-linear computations with a reasonable degree of accuracy.

We list approximation strategies for every non-linear function encountered in this study in Appx. A.

In general, approximation strategies of non-linear functions are configured by a hyper-parameter, such as the "iteration number" in the Newton-Raphson method and the term n in the limit approximation for exponential function, $exp(x) = \lim_{n\to\infty}(1+x/n)^n$. When the iteration number or the term n is large, the approximations should coverage to more accurate approximation results. However, in MPC, the computation cost will increase and become unacceptably high, if the iteration number (or term) is too large. We clarify that this is particularly undesirable in DL scenarios where there are numerous approximation operations in typical DNN models. To simplify the presentation, *we refer to such hyper-parameters as "term" throughout the rest of the study,* even if they have different names in various approximation methods.

Security Guarantee. To our best knowledge, today's MPC-DL frameworks primarily consider *semi-honest* parties, such that parties will not deviate from the standard MPC protocols when making joint computation. Nevertheless, each party may be curious to explore revealing the plaintext of other parties. Following the standard notions of secret sharing [16], [17], [24], [27], we present the security guarantee widely offered by de facto MPC frameworks as follows:

> MPC protocols are secure against information leaking against any passive static adversary corrupting up to $|\mathcal{P}|-1$ of the $|\mathcal{P}|$ parties participating in the computation.

B. Private-Preserving Model Inference

Scenario. Following, we introduce how MPC enables privacy-preserving model inference across multiple parties. While we use two parties (Alice and Bob), the scenarios are extensible into multiple parties. Typically, Alice, the model provider, possesses a *pre-trained* plaintext model that cannot be revealed, while Bob, the data provider, would like to use Alice's model to evaluate on his private data samples. Privacy is important in this scenario, given that parameters of the pre-trained plaintext model is proprietary to the model provider, and data providers may have sensitive data.

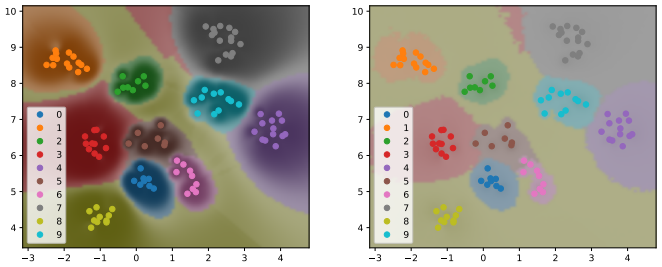
To use MPC, the model provider first locally trains his plaintext model using a plaintext training dataset. The model provider then sends the secret sharing of the plaintext model parameters to the data providers. Accordingly, one or several data providers (i.e., users) send the secret sharing of their test inputs to the model provider. The model provider and data provider will jointly perform computation according to the MPC protocol to get the inference outputs. In this scenario, the data/model providers may also send the secret sharing of their data to multiple servers, who would perform the computations on their behalf, so that data providers do not need to perform the computation themselves, which might be too costly for normal users. For audiences with little background in this field, Appx. B visualizes the steps of conducting MPC-hardened privacy-preserving DL inference in a diagram.

Significance. MPC denotes a promising solution that provides privacy guarantee to boost the adoption of DL in highly regulated, privacy-sensitive sectors such as credit scoring, insurance, government sectors, and healthcare [91], [89], [75], [22]. We have seen that providers of cloud service and machine learning as a service (MLaaS) are incorporating MPC into their applications [80], [54], [88]. For instance, Meta (Facebook) provides MPC-enabled DL frameworks whose APIs resemble that of PyTorch, thus easing the usage and migration for PyTorch users [57]. Overall, we see that MPC frameworks are increasingly vital to offer privacy-preserving DL solutions in the era of cloud computing and MLaaS, and it is important to assess and enhance the security of MPC-based DL scenarios.

Training vs. Predictions. This work considers the scenario where the predictions of MPC-hardened, well-trained DL models may be manipulated with untrusted user inputs. While recent efforts have also been made to use MPC in model training, studying attack vectors in privacy-preserving model training is orthogonal to this paper. It is worth noting that although a few MPC frameworks support privacy-preserving model training, it is often *computation- and communication-heavy*, potentially undermining its adoption in practice. In contrast, MPC-hardened, pre-trained DL models are generally efficient to make predictions (see Table I).

C. Preliminary Vulnerability Exploration

MPC protects the parties from inferring each other's data (test inputs or model weights). However, it cannot improve the robustness of MPC-hardened DL models against adversarial examples (AEs) [36]. AEs, as special test inputs, are typically generated by adding small perturbations to normal inputs. While such small perturbations preserve the inputs' visual appearance (to a human being), the attacked DL model makes a wrong prediction on the AE input. To motivate this work, we analyze the shifted decision boundaries of MPC-hardened models and characterize AE generated over MPC-hardened models. To clarify, existing blackbox AE generation techniques can be smoothly generalized in our setting, as the inference outputs of MPC-hardened DL models are known to the (adversarial) test data providers.



(a) Decision boundaries of the original model.

(b) Decision boundaries of the MPC-hardened model.

Fig. 1: Depicting decision boundaries of LeNet and its MPC-hardened version for classifying MNIST images. A darker hue denotes a higher confidence of model prediction.

Decision Boundary Disctinction. Our preliminary study shows that plaintext DNN models and the corresponding MPC-hardened versions have distinct decision boundaries. In Fig. 1(a), we visualize the decision boundaries of a classification model, LeNet [60], trained on MNIST [61] dataset using one of the state-of-the-art NN visualization tools [86]; we then

harden the trained model using MPC (we use the CrypTen [4] framework; see details in Sec. V) and also visualize the MPC model's decision boundary in Fig. 1(b). We note that the prediction accuracy of both models are sufficiently high, which is 98.65% for the plaintext model and 97.25% for the MPC-hardened model. We use Shannon's Entropy of the output probabilities to quantify the confidence of model predictions, and the darker the color, the higher the confidence. In short, a substantial difference between the boundaries can be observed in Fig. 1. Moreover, we find that for the MPC-hardened model, predictions typically become less confident. Soon Sec. III clarifies that these decision boundary differences root from MPC-specific optimizations and conversions.

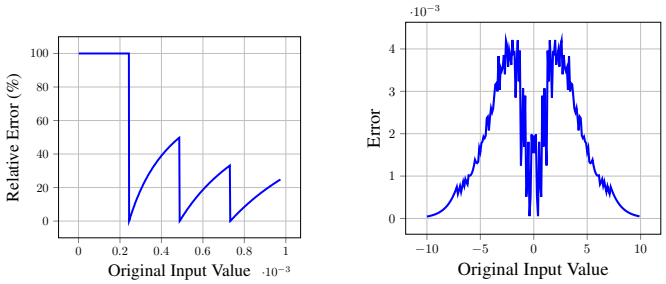
Adversarial Examples (AEs). Given the decision boundary difference between the plaintext model and the MPC-hardened model in Fig. 1, we foresee that malicious data providers can exploit the boundary difference to generate AEs; note that this is distinct from the conventional AEs generated from plaintext DNN models, whose root causes are believed to be the model's inadequate training and unsmooth classification boundaries [62]. To study this, we launch the standard procedure [70], [37] to generate AEs using the above MPC-hardened model. Note that we need to launch black-box AE generation, as the MPC-hardened model is not available to provide gradients. With 275,700 queries, we successfully generate 100 AEs, among which 57 AEs are *not* misclassified by the corresponding plaintext model. That is, there are about 57% of AEs that particularly exploit the MPC-hardened model; Sec. III will explain that these AEs are the result of MPC-specific optimizations and conversions. Overall, these AEs are subtle issues that are specifically toward modern MPC-DL frameworks, creating attack vectors for malicious MPC clients that are under-explored by today's AE study.

MPCDIFF Design Goal. Our focus is to expose hidden defects in MPC-protected *pre-trained* models. Developers can test the MPC-protected models before releasing to users for use. This way, MPCDIFF effectively exposes certain subtle inputs that lead to ill-predictions of MPC-hardened models. We argue that such error-triggering inputs are undesirable, given it could mislead normal users during daily usage of privacy-preserving DL service. Moreover, malicious users may leverage these error-triggering inputs to intentionally manipulate the model predictions. We demonstrate that these inputs are (visually) meaningful compared with regular inputs, illustrating attack vectors that are presumably realistic yet neglected by existing research (Sec. VI-A). In short, we deem that MPC-protected models are likely exploited by such inputs, in a way that is similar to how black-box AEs are leveraged [18].

MPCDIFF also features localizing MPC-transformed DNN operators which are the *root causes* to induce the mis-predictions. MPCDIFF can automatically tune those operators to alleviate the mis-predictions. This way, MPCDIFF can help developers test, debug, and repair their MPC-protected models, achieving high robustness without undermining the MPC-offered privacy guarantee; see empirical results in Sec. VI.

III. Analysis of MPC-Hardened Models

We have introduced how MPC frameworks perform private addition and multiplication on integers, as well as how it approximates non-linear functions in Sec. II-A. Following our preliminary exploration on attack vectors of MPC-hardened DNN models (Sec. II-C), we elaborate on how MPC encodes floating-point numbers, and then clarifies two root causes, $\mathbf{RC}_1$ and $\mathbf{RC}_2$, that result in the prediction boundary deviation (Fig. 1) in MPC-hardened models.



(a) $\mathbf{RC}_1$: fixed-point value error. (b) $\mathbf{RC}_2$: Sigmoid approx. error.

Fig. 2: Sample relative errors incurred from $\mathbf{RC}_1$ and $\mathbf{RC}_2$.

$\mathbf{RC}_1$: Fixed-Point Values. As stated in Sec. II-A, different parties will share integers in secret to perform private computations. To represent a floating point number $\tilde{x} \in \mathbb{R}$, parties often use the fixed-point arithmetic, encoding $\tilde{x}$ by multiplying it with a large integer 2^m: $x = \lfloor \tilde{x} 2^m \rfloor \in \mathbb{Z}$, where m is the *precision bit number*. The multiplication result is then truncated into an integer to use integer sharing $[\![x]\!]$. During decoding, $[\![x]\!]$ is transformed back to the plaintext integer before being divided by 2^m to recover the floating number $\tilde{x}'$. Thus, the encoding & decoding error is $|\tilde{x}' - \tilde{x}|$. Deciding a proper m may be subtle: ideally, a larger m suggests more precise encoding. Nevertheless, an overly large m potentially causes overflow, so compromising the accuracy, especially when subsequent computations amplify the inaccurately-encoded values. The overflow problem is also studied by recent research [81]. On the other hand, a negligible m may also undermine accuracy, as multiplying a small 2^m means most digits in the output value remain in the fractional part and are truncated. Consequently, the encoding & decoding schemes have representation errors when using improper m. An enhancement to avoid overflow is to accompany a large m with a large bit length l in arithmetic secret sharing (see Sec. II-A). However, this adds MPC protocol overhead. In practice, MPC frameworks use a fixed l to encode floating point numbers, such as $l = 64$ in CrypTen and TF-Encrypted. Accordingly, this study analyzes how different m may introduce new attack surfaces of MPC-hardened DL models under fixed values of l. Note that tuning m will *not* incur extra communication and computation cost of the MPC protocols under fixed l.

Fig. 2a illustrates the relative error $\frac{|\tilde{x}' - \tilde{x}|}{\tilde{x}} \times 100\%$ when $m = 12$ in CrypTen. If $\tilde{x} \in [0, 1/2^{12}]$, the relative error would always be 100%, because all of digits in $\lfloor \tilde{x} 2^m \rfloor$ remain in the fractional part and are truncated. When $\tilde{x} \in [1/2^{12}, 2/2^{12}]$, the relative errors are smaller as $\tilde{x}$ grows than when $\tilde{x} \in [0, 1/2^{12}]$. Nevertheless, the errors increase when $\tilde{x}$ is closer to $2/2^{12}$ due to truncating the fractional part of the results after multiplying by 2^{12}. Similar phenomena can be observed for $\tilde{x} \in [2/2^{12}, 3/2^{12}]$, albeit with smaller relative errors as $\tilde{x}$ increases.

Multiplication Truncation. We have discussed using fixed-point integers to encode floating-point numbers. The outcome

of multiplying two fixed-point values is a fixed-point value with $2m$-bit precision. Therefore, truncation is again required to scale down to m-bit precision, and following computations can proceed. As expected, this truncation will likewise add inaccuracies, as also noted by prior research [48]. The root cause is aligned to encoding & decoding, as truncation will inevitably undermine fractional precision (by trimming digits in the fractional part off). Increasing m reduces the truncation error but increases the likelihood of overflow.

$\mathbf{RC}_2$: Non-linear Function Approximation. Sec. II-A1 explains that non-linear functions, such as those commonly-used activation functions, Sigmoid, Tanh, and GELU, are approximated. The approximation will inevitably lead to errors. Sec. II-A1 also clarified that approximation methods for these non-linear functions are uniformly configured by a hyper-parameter, dubbed as "term" in this paper. Fig. 2b depicts the output deviation between the MPC-approximated Sigmoid and the ground truth Sigmoid. We use the default Sigmoid implementation provided by CrypTen, which approximates $Sigmoid(x)$ as the reciprocal of $1 + e^{-x}$. As a common approach, the Newton-Raphson method is used to approximate the reciprocal, whereas the exponential function is approximated using a limit approximation, $exp(x) = \lim_{n\to\infty}(1 + x/n)^n$. We also present the approximation strategies for other non-linear functions in Appx. A. Generally, a larger term n will achieve more accurate results. However, this will induce large computation overhead if we increase the terms for all the non-linear functions, which is unacceptable for DL models with multiple layers, as mentioned in Sec. II-A1. We use the default configuration in CrypTen and it is observed that when the input is in the range $[-5, 5]$, the error is significant, i.e., greater than $1e-3$. For DL models with multiple layers, we expect that the errors propagate and may cause a wrong prediction result; see the following paragraph for error propagation.

Bounds for Error Propagation. We have the following theorem to bound the error propagation:

Theorem 1 (Bounds for Error Propagation)**.** *Considering a DNN with multiple layers, the error induced in layer L_i is ϵ_i, and the error in layer L_{i+1} is ϵ_{i+1}.*

If layer L_{i+1} is a linear layer (including fully connected layers and convolutional layers), and we can treat this layer as multiplications with weights w_{i+1}. Then, $\epsilon_{i+1} = w_{i+1}\epsilon_i$ is bounded:

$$\sigma\|\epsilon_i\|_2 \leq \|\epsilon_{i+1}\|_2 \leq \|w_{i+1}\|_2 \cdot \|\epsilon_i\|_2,$$

where σ is the minimum singular value of w_{i+1}. The bounds generalize to all layers that need matrix multiplications.

If layer L_{i+1} is non-linear activation layer, the error is bounded by:

$$\|\epsilon_{i+1}\|_2 \leq L_c\|\epsilon_i\|_2,$$

where L_c is the Lipschitz constant of layer L_{i+1}.

Proof for Theorem 1: We first prove the bounds for linear layer L_{i+1}. The upper bound $\|\epsilon_{i+1}\|_2 \leq \|w_{i+1}\|_2 \cdot \|\epsilon_i\|_2$ is a direct application of Cauchy-Schwarz inequality in the Euclidean space $\mathbb{R}^n$:

$$\left(\sum_{i=1}^{n} u_i v_i\right)^2 \leq \left(\sum_{i=1}^{n} u_i^2\right) \cdot \left(\sum_{i=1}^{n} v_i^2\right)$$

To get the lower bound, we first perform the singular value decomposition on matrix w_{i+1}. $w_{i+1} = U\Sigma V^T$, where U and V are orthogonal and the diagonal elements of Σ are the singular values of w_{i+1}. Thus,

$$\begin{aligned}\|\epsilon_{i+1}\|_2^2 &= \|w_{i+1}\epsilon_i\|_2^2 = \|U\Sigma V^T\epsilon_i\|_2^2 = \|\Sigma\left(V^T\epsilon_i\right)\|_2^2 \\ &= \sum_k \sigma_k^2 (V^T\epsilon_i)_k^2 \geq \sigma^2\|V^T\epsilon_i\|_2^2 = \sigma^2\|\epsilon_i\|_2^2,\end{aligned}$$

where σ is the minimum singular value of w_{i+1}: $\sigma_k \geq \sigma$, $\forall k$. And we have the lower bound $\|\epsilon_{i+1}\|_2 \geq \sigma\|\epsilon_i\|_2$.

For non-linear layer L_{i+1} whose Lipschitz constant is L_c:

$$\sup_{x_i \neq x_j} \frac{\|L_{i+1}(x_i) - L_{i+1}(x_j)\|_2}{\|x_i - x_j\|_2} \leq L_c$$

Thus, we have $\|\epsilon_{i+1}\|_2 = \|L_{i+1}(x + \epsilon_i) - L_{i+1}(x)\|_2 \leq L_c\|\epsilon_i\|_2$. ■

Implication. With Theorem 1, we conclude that errors, due to either $\mathbf{RC}_1$ or $\mathbf{RC}_2$, will *not* vanish given the presence of a non-trivial number of matrix multiplications in a DNN. Rather, errors will propagate via model layers and influence the model outputs, potentially causing certain inputs to be mis-predicted. This is because of the lower bound. When the error ϵ_i is negligible, however, this error propagation will have a limited effect on the model precision. This is reflected by the upper bounds $\|w_{i+1}\|_2 \cdot \|\epsilon_i\|_2$ and $L\|\epsilon_i\|_2$. This implication is further examined below.

Pervasiveness and Importance. We have illustrated two key root causes, $\mathbf{RC}_1$ and $\mathbf{RC}_2$, that induce inaccuracy in MPC-protected models. According to our observation and analysis under Thm. 1, when computation resources are sufficient, modern MPC-protected models will only manifest negligible accuracy drop comparing to the plaintext models. In other words, the defects MPCDIFF aims to capture are *subtle and rarely occurs* in daily usage. However, this should not impede the importance of this research: in reliability-sensitive scenarios such as credit prediction, a single ill-prediction made by the MPC-protected models is not desirable, undermining the brand reputation and inducing financial losses. Furthermore, when the user has limited computing resource, such as when using a mobile device, the precision bit number m and terms are usually reduced, resulting in larger ϵ_i, increased mis-predictions, and lower accuracy. In Sec. VI, we show that MPCDIFF can effectively uncover many of such errors by mutating model inputs to gradually maximize output deviations of MPC-hardened and plaintext models.

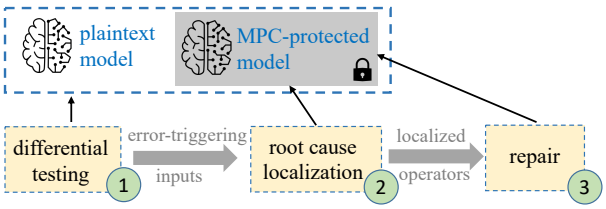


Fig. 3: The workflow of MPCDIFF.

IV. DESIGN OF MPCDIFF

Overview. This work designs MPCDIFF, the first testing framework to detect mis-predictions of MPC-protected models.

Fig. 3 depicts the pipeline of MPCDIFF. Given a target MPC-protected model, developers are anticipated to also prepare the corresponding plaintext model. As introduced in Sec. II-B, to harden a pre-trained plaintext model with MPC, parameters need to be encrypted. Therefore, developers can keep the plaintext models at this phase, together with its MPC-hardened version, as the inputs of MPCDIFF.

① **Differential Testing.** MPCDIFF employs feedback-driven differential testing to explore inputs that result in deviant outputs of MPC-protected models and their plaintext models (Sec. IV-B). That is, MPCDIFF aims to find an input i which can result in a considerably large output deviation δ of the MPC-protected model M_m and its plaintext version M_p.

$$\underset{i}{\text{maximize: }} \delta = |M_p(i) - M_m(i)| \tag{1}$$

We define a threshold T such that when δ, denoting the deviation of model outputs, is greater than T, we collect the inputs as *error-triggering* inputs. Sec. IV-B presents the technical details. Further in Sec. VI-A, we show that most error-triggering inputs are stealthy, hardly distinguishable comparing with normal inputs. This illustrates the attack vectors toward MPC-hardened models that are practical in real-world scenarios.

② **Root Cause Localization.** In Sec. III, we have introduced two key root causes, $\mathbf{RC}_1$ and $\mathbf{RC}_2$, that result in accuracy loss of MPC-protected models. We clarify that $\mathbf{RC}_1$, encoding & decoding errors, can be fixed globally (see ③). Nevertheless, given typical production DNN models can have thousands of non-linear functions, we aim to localize and tune the terms associated with a few critical neurons (non-linear functions like Sigmoid) that primarily contribute to the output deviations. This way, we fix $\mathbf{RC}_2$ with a moderate cost. Tuning every non-linear function would be too costly and unrealistic. Given error-triggering inputs $i \in I$ found in ①, Sec. IV-C discusses how MPCDIFF localizes a set of MPC-transformed non-linear functions that are critical to output deviations.

③ **Repairing.** We present two repairing approaches. For $\mathbf{RC}_1$, we tune the precision bit number m. To repair $\mathbf{RC}_2$, we tune the terms associated with error-causing neurons localized in ②. The end result would be another MPC-hardened model M_m^+ with better robustness while retaining stable accuracy and moderate extra overhead compared to M_m. Developers, as the model provider, can release M_m^+ for users to use. Before discussing the technical details, we first clarify the application scope of MPCDIFF in Sec. IV-A.

A. Application Scope and Clarification

Study Focus: Accuracy vs. Robustness. Following Sec. II-C, we clarify that MPCDIFF is designed to enhance the robustness of MPC-hardened models, not the accuracy. Overall, DL model robustness conceptually differs from accuracy. Accuracy measures how well a model performs on standard test data, while robustness measures model's resistance to adversarial example (AE) attacks. High accuracy does not mean high robustness. As in Sec. II-C, MPC-hardened models maintain high accuracy but their decision boundaries change significantly, making them vulnerable (less robustness) to AE attacks. MPCDIFF is designed to fill the gap between the model's decision boundary with and without MPC to improve its robustness, *not* its accuracy. See Sec. VI on the encouraging empirical results of robustness improvement.

Main Audiences. The main audiences of this work are model owners who want to use MPC frameworks to hide their models from data providers in daily usage. Our work helps model owners to assess and augment the robustness of their models before release. This would eliminate potential attack vectors of MPC-protected models. MPCDIFF is the first automated framework in this field.

Malicious Model Owners. MPCDIFF is designed for normal developers (model owners) to benchmark the quality of their MPC-hardened models in an in-house setting. That is, MPCDIFF is *not* intended to be used against an active adversary. For instance, a malicious developer may want to add a backdoor in his MPC-enabled models to control the model predictions regarding inputs with backdoor triggers. Detecting such injected backdoors in MPC-hardened models is an area for future work.

Distinguishing from AEs. Following our empirical exploration in Sec. II-C, we further clarify how MPCDIFF's findings distinguish from AEs. Overall, similar with AEs, MPCDIFF's findings, error-triggering inputs, can manipulate the prediction outputs of MPC-hardened models. Real-world black-box AE attacks often denote an online setting [50], [39], [87], where they require attackers to iteratively query a remote model (e.g., a cloud service) with mutated inputs to control the model predictions at their will. Nevertheless, we clarify that findings of MPCDIFF are *distinct* with AEs found by prior techniques. MPCDIFF uses differential testing to find inputs that maximize the output deviations of MPC-protected models and plaintext models. Note that the deviation-triggering inputs will retain the *same prediction labels in the plaintext models*, whereas they largely alter the predictions on the MPC-hardened models. In contrast, conventional AEs change the predictions of the plaintext models. Sec. III also shows that a large portion (57%) of AEs generated from MPC-hardened models won't alter the predictions of corresponding plaintext models. In short, root causes of MPCDIFF's findings are domain specific to MPC (noted in Sec. III), whereas conventional AEs are pervasive in DNNs and are believed to root from inadequate training and unsmooth classification boundaries [62].

White-Box vs. Black-Box. We consider a black-box testing scenario where we only rely on the outputs of MPC-hardened models to guide our testing and input mutation. We refrain from using gradients to form the testing guidance. The reason is that for MPC-protected models, it is very slow, if at all possible, to obtain gradients. Also, offering the black-box solution facilitates testing third-party MPC-hardened models in remote APIs, suppose the corresponding plaintext model (or a similar alternative model) is available.

Testing vs. Verification. To our best knowledge, MPCDIFF is the first *testing* approach toward MPC-protected DL models. MPCDIFF shares a similar focus with most security testing tools (e.g., fuzzers [99], side-channel analyzers [76], differential privacy analyzers [30]) to detect errors instead of proving their absence. Obviously, MPCDIFF cannot offer static verification: when MPCDIFF reports no errors, we cannot conclude that the MPC-protected model is free from output deviations. Nevertheless, the testing approach delivered by MPCDIFF is

Algorithm 1 Feedback-driven differential testing.

```
1: function OutputDeviation(M_p, M_m, i', T)
2:     PRED_plaintext ← PREDICTVECTOR(M_p(i'))
3:     PRED_mpc ← PREDICTVECTOR(M_m(i'))
4:     δ ← ||PRED_plaintext − PRED_mpc||_2
5:     if δ > T then                ▷ Output deviation larger than the threshold.
6:         return true
7:     return false
8: function DT(Corpus of Seed Inputs S, M_p, M_m)
9:     Q ← S, O ← ∅
10:    while #total mutations< 15,000 do
11:        i ← CHOOSENEXT(Q)
12:        PRED_plaintext ← PREDICTVECTOR(M_p(i))
13:        PRED_mpc ← PREDICTVECTOR(M_m(i))
14:        T ← ||PRED_plaintext − PRED_mpc||_2
15:        p ← ASSIGNENERGY(i)
16:        for 1 ... p do
17:            i' ← MUTATE(i)
18:            if M_p(i') ≠ M_m(i') then        ▷ Model prediction changes.
19:                add i' in O
20:            else if OutputDeviation(M_p, M_m, i', T) = true then
21:                add i' in Q
22:    return O
```

precise and it generates no false positives. More importantly, as a testing tool, MPCDIFF provides defect-triggering inputs, which enable "debugging" MPC-enabled models, localizing root causes, and further repairing the models. We show that the repaired models manifest much higher robustness (Sec. VI-C). We leave conducting formal verification to check the absence of deviation outputs in MPC-enabled models as future work; we envision extending recent advances in DNN verification [32].

B. MPCDIFF — *Differential Testing*

Alg. 1 depicts the workflow of our testing. Function ***DT*** is the main entry point, and ***OutputDeviation*** checks whether the output deviation of the plaintext model M_p and its MPC-encrypted version M_m is larger than a threshold T. ***DT*** accepts a collection of seed inputs $\mathcal{S}$, and a pair of models M_p and M_m. We use $\mathcal{O}$ to store all deviation-triggering inputs, and $\mathcal{Q}$ as a queue to maintain test seeds. The entire campaign is subject for a sufficiently large number of seed mutations (in our current implementation it is 15,000). For each iteration, we pick one input i by popping $\mathcal{Q}$ (line 11). We first compute the baseline output deviation (lines 12–14) as T.[1] T is calculated from the L2 distance of prediction vectors of two models. We determine #mutations by calling ASSIGNENERGY(i); each seed currently has a fixed "energy" p of 10.

We generate a new variant i' by mutating i (line 17). Mutating inputs in a totally arbitrary manner may not be expedient, because DNN inputs are generally well formed. In the current implementation, we consider both tabular data and images as the inputs. Therefore, in addition to apply random noise over i, we take input constraints into consideration to form a bounded mutation: for the current implementation, we limit the mutation over i within the variance in the standard input dataset that was used to train M_p. This way, we observe that the mutated output i' is (visually) meaningful in comparison to regular inputs, indicating that i' denotes a practical, yet overlooked issue to mislead M_m; see evaluation on quality of i' in Sec. VI-A.

We check if prediction outputs are inconsistent (line 18). If so, we add i', as a deviation-triggering input, to $\mathcal{O}$ (line 19). Moreover, i' is deemed as "interesting," in case it increases the output deviation over T (line 20). For this case, we will keep i' in $\mathcal{Q}$ (line 21) for further mutations. Overall, ***OutputDeviation*** serves as a feedback; it guides MPCDIFF to gradually find inputs with the goal of increasing the output deviation until the deviation is large enough to change model predictions.

Alg. 1 returns a set of deviation-triggering inputs $\mathcal{O}$. Users (model owners) can use $\mathcal{O}$ to debug and repair their M_m; see details in Sec. IV-C and Sec. IV-D. Given that said, if MPCDIFF has limited findings, it could mean that M_m is already robust, or that MPCDIFF was not used long enough. We would recommend developers to run MPCDIFF sufficiently long. Nevertheless, as clarified in Sec. IV-A, MPCDIFF is by no means a verifier, and finding no flaws does not necessarily indicate that M_m is defect-free.

Avalability of Seed $\mathcal{S}$. MPCDIFF testing requires a collection of seeds $\mathcal{S}$. We underline that we have no specific requirements for the seeds. In our evaluation, we randomly select 1K or 2K inputs from each model's test dataset.

C. MPCDIFF — *Defect Localization*

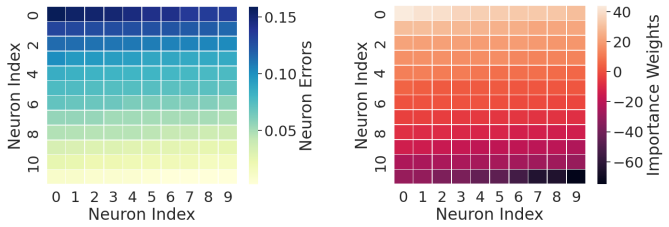
To repair defects due to $\mathbf{RC}_2$, non-linear function approximation, this section introduces a defect localization strategy. At this step, we assign each neuron n_i an importance weight ω_i, which is initialized as zero. ω_i quantifies the overall contribution of neuron n_i toward ill-predictions of M_m.

We update the importance weights in accordance with deviation-triggering inputs $\mathcal{O}$ found in Sec. IV-B. Particularly, we feed each input $o \in \mathcal{O}$ to M_p and M_m, and quantify the output difference of a neuron in M_p and its counterpart in M_m. Let $\delta_i(o)$ be the output difference of neuron n_i in two models under input o, the importance weight ω_i of n_i is incremented by 1, in case $\delta_i(o)$ is greater than a threshold τ_+. Similarly, if $\delta_i(o)$ is smaller than a threshold τ_-, ω_i is decremented by 1. As a result, a collection of neurons with higher influence to the deviation outputs are gradually localized, whose associated importance weights are relatively higher than the rest.

We adaptively configure the thresholds τ_+ and τ_- to scope one-third of neurons with importance weights exceeding τ_+ and another one-third with importance weights below τ_-. The former neurons are the primary contributors of M_m's ill-predictions. Considering Fig. 4a, τ_+ and τ_- are configured to separate the first and last four rows of the heatmap of neurons' output difference $\delta_i(o)$.

After iterating over all the inputs in $\mathcal{O}$, the neuron importance weights ω_i draw a statistical picture of the neurons' likelihood of causing errors. Thus, we treat the neurons with large importance weights as important, and mark the neurons with small importance weights as trivial, as shown in Fig. 4b. The output of this step would be a list of neurons ranked according to their importance weights.

[1] At this step, we check if the seed i can already lead to a deviant prediction between M_m and M_p. If so, we skip mutating this seed and put it in $\mathcal{O}$. We omit this in Alg. 1 to simplify the presentation. Our empirical evaluation shows that this case rarely occurs (less than 2%); see Sec. VI.



(a) Heatmap of the neurons' difference $\delta_i(o)$, the neurons are ranked w.r.t. their $\delta_i(o)$. The first four rows of the neurons will increase their importance weights ω_i, and the last four rows will decrease their ω_i.

(b) Heatmap of the neurons' importance weights ω_i, the neurons are ranked w.r.t. ω_i. The neurons with large weights are marked as important, and the neurons with small weights are marked as trivial.

Fig. 4: Heatmap of neuron difference $\delta_i(o)$ and neuron importance weights ω_i.

D. MPCDIFF — *Error Fixing*

With the localized error-causing neurons, we are able to repair M_m. Recall Sec. III presents two root causes, $\mathbf{RC}_1$ and $\mathbf{RC}_2$, that induce inaccuracy in M_m. Below, we discuss methods to fix these defects.

Fixing $\mathbf{RC}_1$. As introduced in Sec. III, the precision bit m affects the encoding & decoding precision of floating-point numbers and multiplication, thereby influencing the accuracy of M_m. A too large or too small precision bit m can both downgrade the accuracy and incur ill-predictions. We clarify that deciding a proper m is critical yet hardly possible for a universal solution. The "sweet spot" of m may depend on both model structure and MPC framework implementation details. Though deciding a proper m mainly requires manual tuning, we argue that MPCDIFF is particularly *beneficial*: in addition to use the standard test inputs to access m that achieves the best accuracy and low cost, MPCDIFF enables users to tune m from the robustness perspective: users can tune m to minimize the error-triggering inputs $\mathcal{O}$ found by MPCDIFF.

Fixing $\mathbf{RC}_2$. MPC approximates non-linear function computations using various approximation methods (see full details in Appx. A). As noted in Sec. II, approximations are configured using *terms*. In general, larger terms can converge to more accurate results. However, in MPC, especially for DNNs, using too large terms for all the computation units may result in prohibitively high cost, which is undesirable in practice.

Recall some key neurons that primarily contribute to M_m's ill-predictions have been recognized in Sec. IV-C. Thus, we advocate the following scheme to repair ill-predictions and simultaneously retain minimal extra overhead: given a list of neurons ranked by their importance weights (output of Sec. IV-C), MPCDIFF will selectively raise terms following a *meta strategy* (see below) of α neurons beginning with the most important one, where α is decided by users according to their computation cost budget. In the current evaluation, we explore α in the range of 0 to 250. For users with higher cost budget (e.g., using powerful cloud services), α can be increased to fix more neurons accordingly.

This phase requires manually increasing the accuracy levels of certain important neurons. Nevertheless, with MPCDIFF's findings on hand, users can take *robustness* (not just accuracy) into consideration during tuning. We show that after tuning, models become notably more robust in Sec. VI-C. Also, we clarify that the aforementioned meta strategy subsumes detailed approaches to tuning terms for various approximation methods. Overall, different approximation methods often have distinct and subtle tactics to tweak terms. These tactics are domain specific and challenging for MPCDIFF users (who are often DL model creators and providers) to handle. We thus hardcode these tactics in MPCDIFF, and only expose a unified interface for users to configure the number of neurons to be tuned via α. This eases MPCDIFF usage from the user's perspective. We list our meta strategy in Appx. C.

V. IMPLEMENTATION & EVALUATION SETUP

MPCDIFF is written primarily in Python with approximately 3K LOC. All experiments are launched on a machine with one AMD Ryzen CPU, 256GB RAM, and one Nvidia GeForce RTX 3090 GPU.

MPC-Hardened DL Frameworks. MPCDIFF can be integrated to test different MPC frameworks. As listed in Table I, our current evaluation subsumes three popular MPC-hardened DL frameworks: CrypTen [57], [4], TF-Encrypted [6], and PySyft [1]. CrypTen, developed and maintained by Meta (Facebook), is a MPC framework for machine learning built on PyTorch. It enables privacy preserving deep learning in a way that is closely similar to how common PyTorch programs are written. CrypTen, as an industry-quality tool, is carefully optimized to adopt PyTorch's highly optimized tensor computations and use high-speed communication libraries. It can also off-load intensive computations to GPUs. TF-Encrypted is another MPC framework that is tightly integrated with TensorFlow, which allows non-crypto experts to quickly leverage MPC in protecting their private data and models. The primary contribution of TF-Encrypted is also from industry (i.e., Cape Privacy, Alibaba Group, and OpenMined). PySyft is a full-fledged privacy-preserving machine learning framework, which implements various secure computation techniques like federated learning [96], differential privacy [31], MPC, and homomorphic encryption. This library is maintained primarily by OpenMined. Overall, these three frameworks are highly visible in the community, and our observation shows that they all manifest a high engineering quality. We thus believe our evaluation can reflect common defects and improvements over MPC-hardened DL frameworks.

MPC Protocols. CrypTen is primarily built on top of arithmetic and binary secret sharing [25], [23], [35]. It supports two-party secure computation, with the presence of a trusted third party to generate the Beaver triple. For CrypTen, we set up a two-party computation environment to test the framework in which one party provides the private data and the other party provides the well-trained private model. CrypTen enables these two parties to perform private inference without exposing their plaintext model or data.

TF-Encrypted supports several common MPC protocols like ABY3 [73], SecureNN [92], and SPDZ [25]. It allows three-party private model inference. As the default setup of TF-Encrypted, we use its modified SPDZ protocol, known as Pond, for the three-party private inference. Two parties, the model provider and the data provider, perform computations using Pond, while one party delivers the Beaver triple. Similar to

TABLE I: Evaluation setup and statistics.

Framework	Model	Datasets	Plaintext Accuracy	Encrypted Accuracy	#Non-linear Operations	#Multiplication Excluding Non-linear Operations	#Initial Seeds	#Initial Error-Triggering Seeds	Avg. Inference Time Per Input
CrypTen	**LeNet**	**MNIST**	98.65%	97.25%	6,734	20,844,064	2,000	32	1.57s
	MLP-Sigmoid	**Credit**	82.93%	80.70%	120	92,280	1,000	2	0.50s
	MLP-GELU	**Bank**	90.00%	89.90%	250	225,000	1,000	1	0.56s
TF-Encrypted	**LeNet**	**MNIST**	98.20%	96.90%	6,734	20,844,064	2,000	41	0.27s
	MLP-Sigmoid	**Credit**	82.93%	80.10%	120	92,280	1,000	12	0.04s
	MLP-GELU	**Bank**	90.10%	90.10%	250	225,000	1,000	2	0.05s
PySyft	**LeNet**	**MNIST**	97.95%	97.35%	6,530	20,844,064	2,000	18	2.12s
	MLP-Sigmoid	**Credit**	82.93%	80.70%	120	92,280	1,000	2	0.27s
	MLP-GELU	**Bank**	90.10%	89.40%	250	225,000	1,000	1	0.35s

CrypTen, the inference is performed securely without leaking plaintext data or model parameters.

PySyft, like TF-Encrypted, is built on top of SecureNN and SPDZ. To demonstrate the generalization of MPCDIFF with regard to the number of involved parties, we use the three-party setting and SecureNN protocol in PySyft, in which three parties are responsible for the computation. In SecureNN, the Beaver triples are generated using pseudorandom function family (PRF) [92]. In this setting, the data and model parameters are first shared in secret with these three servers, and then the servers will together perform the inference on the encrypted data and the model.

Inference Speed. It is worth noting that these MPC frameworks can introduce a noticeable amount of overhead to the use of large DNN models. For instance, it is disclosed [57] that when using CrypTen-hardened large DNNs for image analysis tasks, it is about 10 to 100 times slower than using the native PyTorch. Nevertheless, we clarify that this overhead does *not* primarily undermine the efficiency of MPCDIFF. The key reason is that MPCDIFF is used in an in-house, localhost setup. For our setup, the communication speed is negligible, as we configure all frameworks using the localhost network. The computation speed is also acceptable on our AMD CPU. As a reasonable assumption, MPCDIFF users (DL model providers) can follow our setup to use MPCDIFF for testing purpose using their machines and localhost networks. Table I (last column) reports the average inference time in our experiments, which is generally speedy (much faster than the normal usage). This justifies the usability of MPCDIFF. We present further discussion about how the computation/communication cost would change after model repairing in Sec. VI-C.

Datasets. We evaluate MPCDIFF using several data formats, including image and tabular data. We choose popular DL tasks like image classification, credit score prediction, and deposit subscription prediction. Specifically, we select representative datasets MNIST [61], Credit [98], and Bank [74]. MNIST is a prominent dataset for classifying handwritten digits, and it is commonly used to benchmark DL models. Credit is a dataset frequently used for secure computation tasks. It contains 23 features such as clients' age, the amount of bill statement, and their repayment status in various months. The task of Credit is to predict whether the client will miss a payment next month, which reflects the client's credit level. Bank is related to the direct marketing campaigns (phone calls) of a Portuguese banking institution; this dataset is often used for benchmarking private computing scenarios. It contains 20 features for each sample, such as consumer price index and consumer confidence index. Its task is to predict if the client has subscribed a term deposit or not.

We clarify that benchmarking tabular datasets (Credit and Bank) and a relatively simple image dataset (MNIST) should *not* undermine MPCDIFF's applicability. MPCDIFF's technical pipeline is orthogonal to particular data types. Moreover, we underline that in typical MPC scenarios (i.e., computation over private data), tabular data such as financial record are the *mainstream*. To show the generality, we also use images (MNIST) to evaluate MPCDIFF. Nevertheless, from the MPC ecosystems, we have yet to observe real-world activities or needs for securely processing complex media data like ImageNet-quality photos [28], audios, or videos.

Models. In accordance with the datasets, we use three popular DNN models as our test models. For MNIST, we train a five-layer LeNet [60] with two convolutional layers and three fully-connected layers. We use Sigmoid as the activation function, and we perform batch normalization and average pooling before and after each activation layer. For Credit, we train a two-layer MLP to predict the credit level of the clients, with Sigmoid as the activation functions. For the Bank dataset, we train a two-layer MLP with GELU as the activation functions. GELU is effective and commonly-used in many models like BERT [29]. We clarify that all of these models are smoothly supported by the evaluated MPC frameworks, and we set the precision bit number of each model as $m = 12$. Moreover, they are all well-trained, whose accuracies are reported in Table I. See Appx. D for each model's full architecture.

VI. EVALUATION

In this section, we evaluate the performance of MPCDIFF following the setup introduced in Sec. V. In accordance with three major components of MPCDIFF, we first evaluate finding deviation-triggering inputs in Sec. VI-A. We then evaluate localizing error-causing neurons and model repairing in Sec. VI-B and Sec. VI-C, respectively.

A. Finding Deviation-Triggering Inputs

Table I reports the evaluated MPC frameworks, datasets, and models. All the MPC-hardened models have good performance, with an accuracy of about 97% for MNIST, 80% for Credit, and 90% for Bank. Their accuracies are comparable to those of the plaintext models. In other words, deviation-triggering inputs found by MPCDIFF are *not* the results of poorly trained models. Instead, they are derived from the two root causes $\mathbf{RC}_1$ and $\mathbf{RC}_2$ noted in Sec. III.

We randomly select 2K or 1K inputs from the test split of each dataset to form the seeds of MPCDIFF. As seen in Table I,

only a small number of seeds can cause deviated outputs. We interpret this as consistent with our discussion in Theorem 1, in the sense that the error is upper bounded. Since PySyft does not support the 1D-Batchnorm, we remove the batchnorm layer between fully-connected layers in LeNet. Thus, PySyft has a fewer amount of non-linear operations than the other two frameworks for LeNet. Consequently, fewer seeds can trigger output deviations for LeNet/PySyft.

We launch MPCDIFF to test each MPC-hardened model with maximal $15,000$ mutation iterations over the seeds. As fuzzing tools are often configured, we use reasonably large mutation iterations for evaluation, and we anticipate to find more Deviation-triggering inputs by increasing this setup. We report the findings in Fig. 5. Overall, MPCDIFF detects a great number of deviation-triggering inputs. We report several deviation-triggering input samples in Fig. 6, for which plaintext models can yield correct predictions. However, when using the MPC-hardened models, the predictions are incorrect. Note that the dimensions of the Credit and Bank datasets are much smaller than those of MNIST, and the MLP-Sigmoid and MLP-GELU models are relatively simpler with fewer non-linear operations. Therefore, MPCDIFF finds less amount of deviation-triggering inputs for these two datasets compared to the results on MNIST. We deem that MPCDIFF is still effective for simple datasets, as it can find 2 to 17 times more deviation-triggering inputs than the number of deviation-triggering seeds.

Deviation-Triggering Input Quality. As discussed in Sec. IV-B, MPCDIFF bounds the total number of mutations toward each input to ensure that the inputs are meaningful. Moreover, we apply projection after each mutation to guarantee that the inputs are within a pre-defined value range, e.g., the image pixels will be projected to $[0, 1]$, and the mutation on tabular data is within its pre-defined variance. As a common setup, we calculate the $L2$ distance between the mutated deviation-triggering inputs and the original inputs to quantify the distance between these inputs.

Fig. 6 shows that the deviation-triggering inputs retain meaningful contents. Moreover, we find that *the plaintext model M_p yields correct predictions on all these "deviation-triggering" inputs*, which in turn demonstrates that they are perceptually meaningful from the well-trained DNNs' perspective. The average $L2$ distance over the input features reflects that our mutation is small: about 0.002 divergence per pixel on average for MNIST. We summarize the key findings below:

> MPCDIFF can effectively detect a great number of deviation-triggering inputs on various datasets and models. These inputs have high quality, with close distance to normal data and hard to distinguish.

B. Error-Causing Neuron Localization

As a pre-requist for fixing $\mathbf{RC}_2$, this section evaluates the defect localization undertaken by MPCDIFF. We first underline that it is generally difficult to acquire the "ground truth" in this defect localization evaluation. It is inherently obscure to decide which neurons in the MPC-hardened models primarily cause output deviations. Nevertheless, we notice that the AI community have explored eXplainable AI (XAI) techniques [11], [85] to explain the decisions of neural networks.

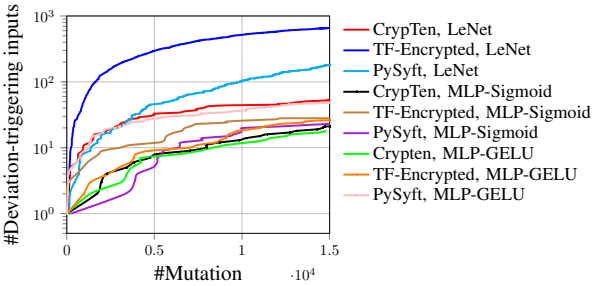


Fig. 5: #Deviation-triggering inputs found by MPCDIFF. Deviation-triggering inputs retain correct predictions in plaintext models but trigger wrong predictions in MPC-hardened models.

MPC Framework	Datasets	Error-Inducing Inputs	Avg. L2-Distance
CrypTen	MNIST	 ···	0.0018
	Credit	[0.000, 1.000, ..., 0.014, 0.039] ···	0.019
	Bank	[0.494, 0.454, ..., 0.957, 0.860] ···	0.018
TF-Encrypted	MNIST	 ···	0.0029
	Credit	[0.010, 0.000, ..., 0.276, 0.009] ···	0.0022
	Bank	[0.197, 0.636, ..., 0.000, 0.170] ···	0.032
PySyft	MNIST	 ···	0.0034
	Credit	[0.802, 0.000, ..., 0.846, 0.297] ···	0.023
	Bank	[0.049, 0.727, ..., 0.060, 0.106] ···	0.015

Fig. 6: Examples of deviation-triggering inputs found by MPCDIFF.

Particularly, they are able to identify the neurons that make significant contributions to the model's prediction.

Though XAI and MPCDIFF's defect localization procedures are conceptually distinct, we seek to use XAI, as a reflection, to assess the findings of MPCDIFF. To do so, we use the idea in [10] to calculate the second order derivative of the neurons with respect to the inputs in test datasets. A larger second order derivative value indicates that the neuron's first order derivative changes quickly and it is more sensitive during the model training.

As shown in Table II, the neurons marked as important by MPCDIFF and the second order derivation have an overlap of at least 30.1%. We observe that the overlap is generally higher for complex models like LeNet. A probable explanation is because LeNet contains a batch normalization layer before the activation function, which maps the value to a range with 0 mean and 1 variance, allowing the activation functions to work properly and reducing the likelihood of gradients vanishing. Thus, compared with MLP models, neurons with medium values in LeNet are more likely to contribute to the inference results. Recall that in Fig. 2b, medium inputs can result in larger errors in Sigmoid. This presumably explains why that LeNet has a greater degree of overlap.

We clarify that the objectives of XAI and MPCDIFF's defect localization are *distinct*, resulting neuron localization differences as shown in Table II. This implies that certain neurons, though yield substantial output differences between plaintext models and MPC-hardened models, may be simply ignored by XAI. In contrast, certain neurons that are treated as important by XAI might be accurate in MPC-hardened models.

TABLE II: Overlap between XAI-marked neurons and findings of MPCDIFF's defect localization.

MPC Framework	Datasets, Models	Neuron Overlap Ratio
CrypTen	MNIST, LeNet	69.93%
	Credit, MLP-Sigmoid	35.29%
	Bank, MLP-GELU	30.12%
TF-Encrypted	MNIST, LeNet	61.49%
	Credit, MLP-Sigmoid	35.46%
	Bank, MLP-GELU	50.60%
PySyft	MNIST, LeNet	61.87%
	Credit, MLP-Sigmoid	63.82%
	Bank, MLP-GELU	44.58%

TABLE III: Models after applying initial repairing.

MPC Framework	Models	Acc. on Test Data Bef. / Aft. Repair	Acc. on Deviation-Triggering Inputs Bef. / Aft. Repair
CrypTen	LeNet	97.25% / 97.10%	0% / 49.45%
	MLP-Sigmoid	80.70% / 80.70%	0% / 86.95%
	MLP-GELU	89.90% / 90.00%	0% / 71.87%
TF-Encrypted	LeNet	96.90% / 96.60%	0% / 15.41%
	MLP-Sigmoid	80.10% / 80.20%	0% / 57.14%
	MLP-GELU	90.10% / 90.20%	0% / 88.88%
PySyft	LeNet	97.80% / 98.00%	0% / 22.05%
	MLP-Sigmoid	80.70% / 80.90%	0% / 68.00%
	MLP-GELU	89.40% / 89.90%	0% / 96.77%

Despite this inherent difference, we nevertheless see that a large fraction of agreement between MPCDIFF's findings and XAI. This indicates a large number of neurons that primarily result in output deviations in MPC are also marked as important in the plaintext models. Our key findings are as follows:

> Our findings adequately justify the effectiveness of defect localization in MPCDIFF. Sec. VI-C will show, form an alternative and practical aspect, that repairing the neurons identified by MPCDIFF at this stage will notably mitigates deviation-triggering inputs. That provides additional support for validity of MPCDIFF-localized neurons.

C. Repairing MPC-Hardened Models

With the detected deviation-triggering inputs and the localized neurons evaluated in Sec. VI-A and Sec. VI-B, we are able to repair MPC-hardened models. As introduced in Sec. IV-D, fixing $\mathbf{RC}_1$ (encoding & decoding errors) requires to tune the precision bit number m, whereas fixing $\mathbf{RC}_2$ will tune terms of α neurons localized by MPCDIFF, beginning with the most important one.

Initial Repairing. To mimic how developers may tentatively repair their models, we first apply an "initial repairing" scheme. Note that for this scheme, we continue to use the original precision bit number $m = 12$ for all models, and we set $\alpha = 40$, meaning that we tune terms of 40 neurons (e.g., Sigmoid, GELU) with the greatest influence on the output deviations. We report the repaired model performance in Table III. Overall, we find that following the initially applied augmentation, all models retain high accuracy on the test dataset and become more robust against the deviation-triggering inputs found by MPCDIFF. Moreover, we report that the average extra cost incurred by the initial repairing scheme is negligible, at approximately 0.003%. We clarify how this is measured when discussing Fig. 7d below.

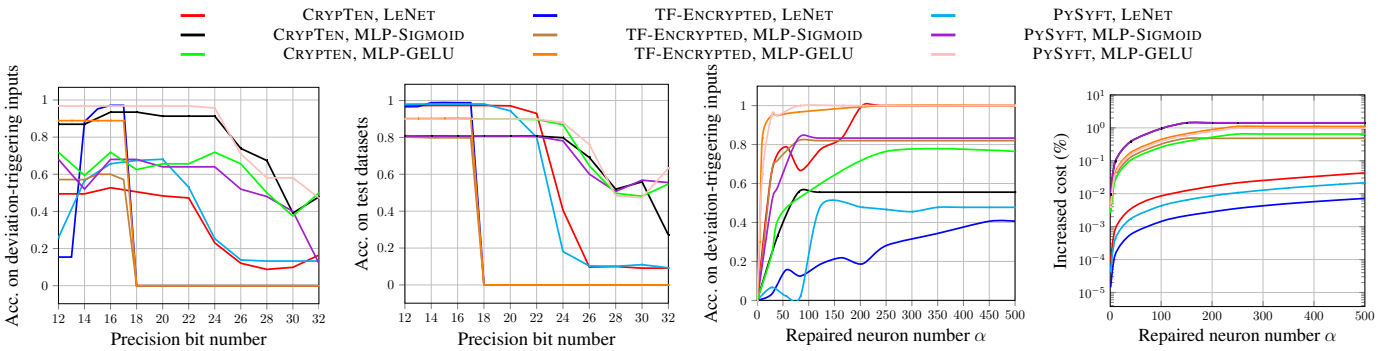
Fixing $\mathbf{RC}_1$ by Tuning m. We now evaluate how precision bit numbers m affect the "initially repaired" models. As noted in Sec. III, too large m can result in overflow errors, while a m that is too small will cause greater encoding & decoding errors. Fig. 7a depicts the performance of several MPC-hardened models with varying m on the deviation-triggering inputs found by MPCDIFF. Note that TF-Encrypted will check for overflow and abort. Thus, it cannot give prediction outputs if m exceeds 18. For CrypTen and PySyft, model accuracy on the deviation-triggering inputs is the highest when $m \in [15, 17]$, and this observation is consistent across different datasets. For TF-Encrypted, the accuracy increases rapidly at the beginning, and when $m \in [14, 17]$, the accuracy is mostly stable. A possible reason is that the base of TF-Encrypted's encoding configuration is 3, rather than of 2. Thus, the scale 3^m is much larger than the other two frameworks' configuration, 2^m. When $m \in [14, 17]$, the scale is sufficiently large to reduce the encoding & decoding errors and will not cause large overflow errors for TF-Encrypted.

We also evaluate the model performance on the test datasets under different m on the "initially repaired" models ($\alpha = 40$). As shown in Fig. 7b, for CrypTen and PySyft, the accuracy on test datasets decreases greatly when $m > 18$. Overall, overflow renders the models non-functional when m is too large. When m is smaller, i.e., from 12 to 14, the accuracy on complex datasets/models like MNIST/LeNet increases slightly.

Besides the well-known issue that an improper m may undermine the accuracy, Fig. 7a and Fig. 7b further suggest that a m primarily affects the robustness of MPC-hardened models. A properly decided m, usually around 15 to 17 (depending on the base of the scale), can result in models with high accuracy and better robustness on the deviation-triggering inputs. This observation is consistent among the three MPC frameworks and their protected models. As already noted in Sec. III, we again clarify that the effect of m is analyzed under fixed bit length $l = 64$. Therefore, tuning m will *not* undermine the communication and computation cost of the MPC protocols.

Fixing $\mathbf{RC}_2$ by Tuning α. We now evaluate the robustness of MPC-hardened models when users can afford extra computation and communication costs, i.e., using a larger α. According to the above analysis, we use the precision bit number $m = 16$. To mitigate the effect of varying m on neuron importance ranking and ensure the accuracy of the neuron importance scores, we re-launch testing with $m = 16$ with the same number of mutations as shown in Fig. 5. Then, we re-run neuron localization as in Sec. VI-B using the obtained deviation-triggering inputs for models with $m = 16$. Fig. 7c shows that users can improve the robustness of all models with even small increase of α; in other words, users are expected to trade small extra computation/communication overhead for robustness increase. We present the relation between the theoretical relative cost increased and the number of repaired neurons α in Fig. 7d.

As stated in Sec. IV-D, we increase the terms in non-linear functions approximation to repair a localized neuron. Overall, the increased terms lead to more communication and computation rounds. In contrast, we clarify that changing the precision bit number m would not influence the cost. Thus, we are able to measure the relative cost from a theoretical standpoint by calculating the ratio of the total iteration rounds before and after repairing. However, we refrain from running the MPC-



(a) Relationship between m and accuracy on deviation-triggering inputs.

(b) Relationship between m and accuracy on test datasets.

(c) Relationship between α and accuracy on deviation-triggering inputs.

(d) Relationship between α and increased cost.

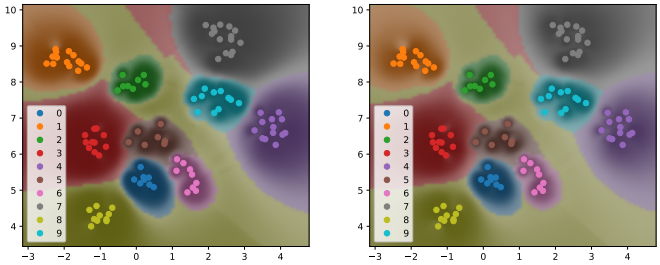
Fig. 7: Results of repairing MPC-hardened models.

hardened models on the physical machines to benchmark cost, as our testing is conducted on an in-house, localhost setup.

The cost increases linearly, but we use the log axis to ease the presentation. For instance, when increasing α from 0 to 150, the relative cost increases for 0.01% in LeNet/CrypTen, whereas the robustness illustrated in Fig. 7c increases for 80%. We observe that all of the evaluated models can achieve better performance with a larger α. And for MNIST/LeNet, when α is small, there are fluctuations on the accuracy of the deviation-triggering inputs. The reason is that the LeNet network and MNIST datasets are relatively more complex compared with other networks and datasets we evaluate. However, after increasing α, we can still get much better robustness, which reflects the effectiveness of MPCDIFF. Also, we omit the accuracy of the models on test datasets because it is stable when we increase α, i.e., the accuracy on test datasets fluctuates within 1%, which reflects that repairing will not sacrifice the model's accuracy on normal test inputs.

AEs and Decision Boundaries of Repaired Models. Recall that in Sec. II-C, we have generated AEs on the CrypTen-hardened models trained on MNIST before repairing utilizing blackbox AE generation techniques. The results show that we can generate 100 AEs using 275,700 queries. We re-launch the AE generation on the CrypTen-hardened LeNet after repairing using $\alpha = 250$ and $m = 16$, and we can only generate 19 AEs using the same number of queries. And among the generated AEs, there are only 3 that are not misclassified by the plaintext model. That is, we only generate 3 deviation-triggering inputs using the same large number of queries. Compared to the results in Sec. II-C, it is much harder to generate derivation-triggering inputs using blackbox AE generation on the repaired model. The results further reflect that the repaired model is becoming more robust. Additionally, we also visualize the decision boundaries of the CrypTen-hardened LeNet after repairing. As shown in Fig. 8, the boundaries of the CrypTen-hardened LeNet after repairing are much closer to that of the original model compared to the results in Fig. 1b. We deem that this result illustrates the effectiveness of repairing in MPCDIFF.

To further demonstrate the effectiveness of the repairing, we launch MPCDIFF to test the repaired models. We use the same configuration and seeds as we launch testing, and we run MPCDIFF to again perform $15,000$ mutations over the seeds. We use $\alpha = 250$ for LeNet, $\alpha = 40$ for MLP-Sigmoid, and



(a) Decision boundaries of the original model.

(b) Decision boundaries of the repaired MPC-hardened model.

Fig. 8: Depicting decision boundaries of LeNet and its repaired MPC-hardened version for classifying MNIST. A darker hue denotes a higher confidence of model prediction.

$\alpha = 80$ for MLP-GELU. The results are shown in Fig. 9. We observe that we can still find some deviation-triggering inputs on the repaired models, which indicates the effectiveness of our testing and the pervasiveness of deviation-triggering inputs in MPC-hardened models. Compared to Fig. 5, however, the number of deviation-triggering inputs found by MPCDIFF is much smaller, indicating that models after repairing are more robust. For CrypTen/MLP-GELU, we can still find comparable deviation-triggering inputs as Fig. 5. We deem the reason is that the model is not comprehensively repaired and MPCDIFF can still exploit the neurons with lower precisions to misclassify the models. Thus, we launch testing on CrypTen/MLP-GELU again using higher $\alpha = 250$. As expected, the number of generated deviation-triggering inputs reduces by 64.7%. In Fig. 5, MPCDIFF finds 17 deviation-triggering inputs, and to compare, we find 6 deviation-triggering inputs after increasing $\alpha = 250$ at this step. For PySyft/LeNet, we also find a relatively large amount of deviation-triggering inputs. Similar to CrypTen/MLP-GELU, defect localization for this case may not comprehensively reflect root cause neurons, hence reducing the efficacy of our augmentation. Nevertheless, the number of deviation-triggering inputs found by MPCDIFF is still significantly smaller than the original results in Fig. 5. We summarize the key findings at this step below:

> The repaired models have comparable, if not better, accuracy than the original models on test data. Moreover, we have encouraging observations that the repaired models are significantly more robust than the original models.

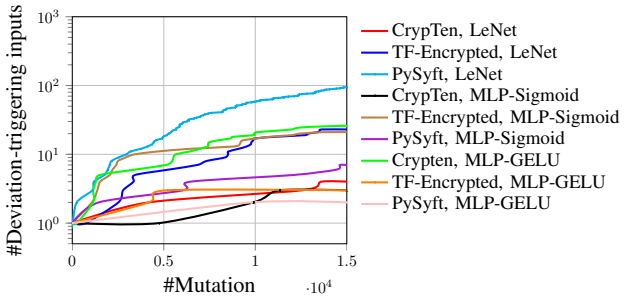


Fig. 9: #Deviation-triggering inputs found by MPCDIFF on repaired models.

VII. Discussion

Fine-Grained Tuning of m. We have evaluated tuning m as a global setting of each MPC-hardened model in Sec. VI-C. Nevertheless, an optimal m depends on the value range of its encoded floating-point number, and apparently different layers in a DL model may provide outputs with distinct ranges. We deem it an interesting future work to tune different precision bit number m for different layers or even different elements in an MPC-hardened model. We envision the major hurdle is on extra overhead, as the parties have to frequently encode & decode floating numbers if they use inconsistent m. Encoding a value from a higher precision bit to a lower one may also introduce extra errors. Moreover, we see that the intermediate outputs of a DNN are typically of the same order of magnitude, implying that determining a proper global precision bit number should be sufficient for most cases. Therefore, tuning different m may have a negligible effect on robustness, but notably undermine speed.

Information Disclosure on Repaired MPC-Hardened Models. The model provider will release the computation paradigm according to MPC protocols, facilitating secure DL inference on data providers' ends. The computation paradigm includes information of the model architecture and how to calculate each (non-linear) neuron, whereas model weights are shared *in secret*. As expected, if the model provider has repaired the MPC-hardened model using MPCDIFF, then the computation paradigm will presumably disclose inconsistent terms for non-linear functions in the model; a higher term implies that its associated neuron is presumably critical for deciding model outputs. This reveals a certain amount of information to the data owners, comparing to the MPC-hardened models before repairing. Nevertheless, we deem the leakage is negligibly concerned, given that the private data (i.e., the model weights and input data) are securely hidden.

Deviation-Triggering Inputs for Specific Labels. The current objective of MPCDIFF is to find deviations between outputs of plaintext models and MPC-hardened models for *debugging purpose*. Therefore, MPCDIFF currently uses distances between these models' outputs to guide testing, and we do not set a specific target label but detecting as many deviation-triggering inputs as possible. We clarify that MPCDIFF can be extended to mutate inputs for specific labels, e.g., mutating MNIST images to mislead the MPC-hardened model to predict "8" rather than "6". With this regard, previous works that generate adversarial examples in the black-box settings [78] can be considered. We deem it is an interesting future work to explore the attack surface of models deployed in MPC scenarios, particularly in the remote, black-box setting.

Alternative Testing Feedback. As noted in Alg. 1, when performing DT, MPCDIFF leverages the distance between outputs of plaintext models and MPC-hardened models to search test inputs; MPCDIFF gradually finds inputs with the goal of increasing the output deviation until the deviation is large enough to flip model predictions. This design of testing feedback abstracts the MPC-DL model internals (e.g., orthogonal to the types of approximations employed for the non-linear functions), making the testing pipeline of MPCDIFF more general and applicable to various MPC-DL settings. We also notice prior DL testing and attacking works that leverage the model internals (layer-wise or neuron-wise outputs) to guide input mutation [79]. While this may offer finer-grained feedback (e.g., by focusing on the most critical neurons), we believe the primary concern is speed. Hooking into the model internals (e.g., every neuron) incur extra overhead, particularly in the MPC-DL setting, where substantial, extra efforts are needed in converting sharings to plaintext. We believe the currently-formed feedback over model outputs is *sufficient*, and we deem it an interesting future work to explore finer-grained testing feedback.

MPC-Aware Training. We note that users can also bridge the gap between the MPC-hardened models and the plaintext models by applying MPC-aware training. Instead of using the accurate non-linear functions and floating-point representations, users can simulate the MPC behavior and take the deviations into consideration during training. However, in MPC, the approximations may not always be differentiable. Using these approximated nonlinear functions during training can greatly affect the model's convergence. Additionally, modern DL frameworks optimize common nonlinear functions like Sigmoid and GELU. Replacing them with approximations considerably increases training costs. Furthermore, most MPC protocols for DL use approximations and fixed-point computations [82], [84], [41]. We deem MPCDIFF targets popular use cases like those.

VIII. Related Work

Secure Machine Learning. In addition to using MPC, recent research have extensively studied federated learning (FL)-based secure machine learning techniques [71], [42], [77], [96]. Note that MPC can also be used in FL frameworks to encrypt parameters. Furthermore, some trusted hardware features like ARM TrustZone and Intel SGX are used to shield machine learning models in cloud computing and edge computing devices [72], [40], [49], [38], [90]. In addition to protect the privacy of the model, another main branch of research focus is to use differential privacy, MPC, or FL to protect private training data [7], [71], [57]. We envision the potential feasibility of extending the current implementation of MPCDIFF to test FL infrastructures, which may share similar root causes. As for secure machine learning based on trusted hardware environments, we believe it should be orthogonal with MPCDIFF.

Testing & Verification of Privacy-Enhancing Protocols. We have noticed an exciting trend in the community to explore the combination of programming language and cryptography [5]. In particular, recent works have been applying soft-

ware testing, program analysis, and verification techniques toward crypto/privacy-enhancing protocols. Ding et al. [30] apply differential testing to detect violations of differential privacy (DP). DP protocols are tested in a white-box setting: accordingly, they leverage symbolic execution and constraint solving to boost the testing, by prioritizing test inputs that can cover divergent code paths. Recent works in this field apply testing, verification, security auditing, or empirically investigation the implementation of DP protocols from a variety of perspectives [52], [53], [94], [67], [56], [93], [100], [14]. DP-Sniper trains a neuron distinguisher to detect violations of DP guarantees. With this regard, we also notice recent advances in the crypto community that leverage neural models to boost linear or differential cryptanalysis [34], [20], [46], [9], [95]. Some research focuses on analyzing oblivious RAM (ORAM) protocols. Existing works have proposed language-based solutions to verify or testing ORAM protocols [64], [63], [26], [66], [65], [69].

IX. Conclusion

We present MPCDiff, a feedback-driven differential testing tool to detect ill-predictions of MPC-hardened models. Deviation-triggering inputs appear meaningful compared to regular model inputs, illustrating that defects found by MPCDiff are practical, yet overlooked by existing works. We further discuss techniques to localize error-causing computation units, and present techniques to repair defects in MPC-hardened models. We show that the repaired models have high robustness without incurring much extra overhead.

Acknowledgment

This work was supported in part by the research fund provided by HSBC and a RGC GRF grant under the contract 16214723. We are grateful to the anonymous reviewers for their valuable comments.

References

[1] "PySyft," https://github.com/OpenMined/PySyft.

[2] "Research artifact," https://github.com/Qi-Pang/MPCDiff.

[3] "TF-encrypted," https://github.com/tf-encrypted/tf-encrypted.

[4] "CrypTen," https://github.com/facebookresearch/CrypTen, 2022.

[5] "Pl/crypto workshop," https://andrewcmyers.github.io/plcrypt, 2022.

[6] "TF Encrypted," https://github.com/tf-encrypted/tf-encrypted, 2022.

[7] M. Abadi, A. Chu, I. Goodfellow, H. B. McMahan, I. Mironov, K. Talwar, and L. Zhang, "Deep learning with differential privacy," in *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, 2016, pp. 308–318.

[8] Y. Bai, Y. Li, M. Xie, M. Fan, and J. Ming, "A defense framework for privacy risks in remote machine learning service," *Sec. and Commun. Netw.*, vol. 2021, jan 2021. [Online]. Available: https://doi.org/10.1155/2021/9924684

[9] A. Baksi, J. Breier, Y. Chen, and X. Dong, "Machine learning assisted differential distinguishers for lightweight ciphers," in *DATE*, 2021.

[10] R. Battiti, "First-and second-order methods for learning: between steepest descent and newton's method," *Neural computation*, vol. 4, no. 2, pp. 141–166, 1992.

[11] D. Bau, J.-Y. Zhu, H. Strobelt, A. Lapedriza, B. Zhou, and A. Torralba, "Understanding the role of individual units in a deep neural network," *Proceedings of the National Academy of Sciences*, vol. 117, no. 48, pp. 30 071–30 078, 2020.

[12] D. Beaver, "Efficient multiparty protocols using circuit randomization," in *Annual International Cryptology Conference*. Springer, 1991, pp. 420–432.

[13] D. Beaver, S. Micali, and P. Rogaway, "The round complexity of secure protocols," in *Proceedings of the twenty-second annual ACM symposium on Theory of computing*, 1990, pp. 503–513.

[14] B. Bichsel, S. Steffen, I. Bogunovic, and M. Vechev, "Dp-sniper: black-box discovery of differential privacy violations using classifiers," in *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2021, pp. 391–409.

[15] D. Bogdanov, "Foundations and properties of shamir's secret sharing scheme research seminar in cryptography," *University of Tartu, Institute of Computer Science*, vol. 1, 2007.

[16] D. Bogdanov, S. Laur, and J. Willemson, "Sharemind: A framework for fast privacy-preserving computations," in *European Symposium on Research in Computer Security*. Springer, 2008, pp. 192–206.

[17] R. Canetti, "Universally composable security: A new paradigm for cryptographic protocols," in *Proceedings 42nd IEEE Symposium on Foundations of Computer Science*. IEEE, 2001, pp. 136–145.

[18] N. Carlini and D. Wagner, "Towards evaluating the robustness of neural networks," in *2017 ieee symposium on security and privacy (sp)*. IEEE, 2017, pp. 39–57.

[19] N. Chandran, D. Gupta, A. Rastogi, R. Sharma, and S. Tripathi, "Ezpc: Programmable and efficient secure two-party computation for machine learning," in *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 2019, pp. 496–511.

[20] Y. Chen and H. Yu, "Neural aided statistical attack for cryptanalysis." *IACR Cryptol. ePrint Arch.*, vol. 2020, p. 1620, 2020.

[21] R. Cramer, I. Damgård, and Y. Ishai, "Share conversion, pseudorandom secret-sharing and applications to secure computation," in *Theory of Cryptography Conference*. Springer, 2005, pp. 342–362.

[22] CWI, "Secure multiparty computation starts to deliver applications," https://www.cwi.nl/news/2021/secure-multiparty-computation-starts-to-deliver-applications, 2021.

[23] I. Damgård, D. Escudero, T. Frederiksen, M. Keller, P. Scholl, and N. Volgushev, "New primitives for actively-secure mpc over rings with applications to private machine learning," in *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019, pp. 1102–1120.

[24] I. Damgård, M. Fitzi, E. Kiltz, J. B. Nielsen, and T. Toft, "Unconditionally secure constant-rounds multi-party computation for equality, comparison, bits and exponentiation," in *Theory of Cryptography Conference*. Springer, 2006, pp. 285–304.

[25] I. Damgård, V. Pastro, N. Smart, and S. Zakarias, "Multiparty computation from somewhat homomorphic encryption," in *Annual Cryptology Conference*. Springer, 2012, pp. 643–662.

[26] D. Darais, I. Sweet, C. Liu, and M. Hicks, "A language for probabilistically oblivious computation," *Proceedings of the ACM on Programming Languages*, vol. 4, no. POPL, pp. 1–31, 2019.

[27] D. Demmler, T. Schneider, and M. Zohner, "Aby-a framework for efficient mixed-protocol secure two-party computation." in *NDSS*, 2015.

[28] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "ImageNet: A Large-Scale Hierarchical Image Database," in *CVPR09*, 2009.

[29] J. Devlin, M. Chang, K. Lee, and K. Toutanova, "BERT: pre-training of deep bidirectional transformers for language understanding," *CoRR*, vol. abs/1810.04805, 2018.

[30] Z. Ding, Y. Wang, G. Wang, D. Zhang, and D. Kifer, "Detecting violations of differential privacy," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 475–489.

[31] C. Dwork, "Differential privacy: A survey of results," in *International conference on theory and applications of models of computation*. Springer, 2008, pp. 1–19.

[32] T. Gehr, M. Mirman, D. Drachsler-Cohen, P. Tsankov, S. Chaudhuri, and M. Vechev, "Ai2: Safety and robustness certification of neural networks with abstract interpretation," in *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2018, pp. 3–18.

[33] A. Gil, J. Segura, and N. M. Temme, *Numerical methods for special functions*. SIAM, 2007.

[34] A. Gohr, "Improving attacks on round-reduced speck32/64 using deep learning," in *Annual International Cryptology Conference*. Springer, 2019, pp. 150–179.

[35] O. Goldreich, S. Micali, and A. Wigderson, "How to play any mental game, or a completeness theorem for protocols with honest majority," in *Providing Sound Foundations for Cryptography: On the Work of Shafi Goldwasser and Silvio Micali*, 2019, pp. 307–328.

[36] I. Goodfellow, J. Shlens, and C. Szegedy, "Explaining and harnessing adversarial examples," in *International Conference on Learning Representations*, 2015. [Online]. Available: http://arxiv.org/abs/1412.6572

[37] C. Grossmann, H.-G. Roos, and M. Stynes, *Numerical treatment of partial differential equations*. Springer, 2007, vol. 154.

[38] K. Grover, S. Tople, S. Shinde, R. Bhagwan, and R. Ramjee, "Privado: Practical and secure dnn inference with enclaves," *arXiv preprint arXiv:1810.00602*, 2018.

[39] C. Guo, J. Gardner, Y. You, A. G. Wilson, and K. Weinberger, "Simple black-box adversarial attacks," ser. ICML, 2019.

[40] L. Hanzlik, Y. Zhang, K. Grosse, A. Salem, M. Augustin, M. Backes, and M. Fritz, "Mlcapsule: Guarded offline deployment of machine learning as a service," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2021, pp. 3300–3309.

[41] M. Hao, H. Li, H. Chen, P. Xing, G. Xu, and T. Zhang, "Iron: Private inference on transformers," *Advances in Neural Information Processing Systems*, vol. 35, pp. 15 718–15 731, 2022.

[42] S. Hardy, W. Henecka, H. Ivey-Law, R. Nock, G. Patrini, G. Smith, and B. Thorne, "Private federated learning on vertically partitioned data via entity resolution and additively homomorphic encryption," *arXiv preprint arXiv:1711.10677*, 2017.

[43] M. Hastings, B. Hemenway, D. Noble, and S. Zdancewic, "Sok: General purpose compilers for secure multi-party computation," in *2019 IEEE symposium on security and privacy (SP)*. IEEE, 2019, pp. 1220–1237.

[44] D. Hendrycks and K. Gimpel, "Gaussian error linear units (gelus)," *arXiv preprint arXiv:1606.08415*, 2016.

[45] W. Henecka, S. Kögl, A.-R. Sadeghi, T. Schneider, and I. Wehrenberg, "Tasty: tool for automating secure two-party computations," in *Proceedings of the 17th ACM conference on Computer and communications security*, 2010, pp. 451–462.

[46] B. Hou, Y. Li, H. Zhao, and B. Wu, "Linear attack on round-reduced DES using deep learning," in *European Symposium on Research in Computer Security*. Springer, 2020, pp. 131–145.

[47] A. S. Householder, *The numerical treatment of a single nonlinear equation*. McGraw-Hill, 1970.

[48] Z. Huang, W.-j. Lu, C. Hong, and J. Ding, "Cheetah: Lean and fast secure two-party deep neural network inference," in *Usenix Security*, 2022.

[49] T. Hunt, C. Song, R. Shokri, V. Shmatikov, and E. Witchel, "Chiron: Privacy-preserving machine learning as a service," *arXiv preprint arXiv:1803.05961*, 2018.

[50] A. Ilyas, L. Engstrom, A. Athalye, and J. Lin, "Black-box adversarial attacks with limited queries and information," in *International Conference on Machine Learning*. PMLR, 2018, pp. 2137–2146.

[51] R. Impagliazzo and S. Rudich, "Limits on the provable consequences of one-way permutations," in *Proceedings of the twenty-first annual ACM symposium on Theory of computing*, 1989, pp. 44–61.

[52] M. Jagielski, J. Ullman, and A. Oprea, "Auditing differentially private machine learning: How private is private sgd?" *Advances in Neural Information Processing Systems*, vol. 33, pp. 22 205–22 216, 2020.

[53] B. Jayaraman and D. Evans, "Evaluating differentially private machine learning in practice," in *28th USENIX Security Symposium (USENIX Security 19)*, 2019, pp. 1895–1912.

[54] M. Kan, "Facebook's ad business to minimize data collection after apple, google privacy changes," https://www.pcmag.com/news/facebooks-ad-business-to-minimize-data-collection-after-apple-google-privacy, 2021.

[55] M. Keller, E. Orsini, and P. Scholl, "Mascot: faster malicious arithmetic secure computation with oblivious transfer," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016, pp. 830–842.

[56] D. Kifer, S. Messing, A. Roth, A. Thakurta, and D. Zhang, "Guidelines for implementing and auditing differentially private systems," *arXiv preprint arXiv:2002.04049*, 2020.

[57] B. Knott, S. Venkataraman, A. Hannun, S. Sengupta, M. Ibrahim, and L. van der Maaten, "CrypTen: Secure multi-party computation meets machine learning," in *arXiv 2109.00984*, 2021.

[58] V. Kolesnikov, A.-R. Sadeghi, and T. Schneider, "A systematic approach to practically efficient general two-party secure function evaluation protocols and their modular design," *Journal of Computer Security*, vol. 21, no. 2, pp. 283–315, 2013.

[59] N. Kumar, M. Rathee, N. Chandran, D. Gupta, A. Rastogi, and R. Sharma, "Cryptflow: Secure tensorflow inference," in *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2020, pp. 336–353.

[60] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel, "Backpropagation applied to handwritten zip code recognition," *Neural computation*, vol. 1, no. 4, pp. 541–551, 1989.

[61] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[62] K. Leino, Z. Wang, and M. Fredrikson, "Globally-robust neural networks," in *International Conference on Machine Learning*. PMLR, 2021, pp. 6212–6222.

[63] C. Liu, A. Harris, M. Maas, M. Hicks, M. Tiwari, and E. Shi, "Ghostrider: A hardware-software system for memory trace oblivious computation," *ACM SIGPLAN Notices*, vol. 50, no. 4, pp. 87–101, 2015.

[64] C. Liu, M. Hicks, and E. Shi, "Memory trace oblivious program execution," ser. CSF, 2013.

[65] C. Liu, Y. Huang, E. Shi, J. Katz, and M. W. Hicks, "Automating efficient ram-model secure computation," in *2014 IEEE Symposium on Security and Privacy, SP 2014, Berkeley, CA, USA, May 18-21, 2014*, 2014, pp. 623–638.

[66] C. Liu, X. S. Wang, K. Nayak, Y. Huang, and E. Shi, "Oblivm: A programming framework for secure computation," ser. IEEE Security and Privacy, 2015.

[67] C. Liu, X. He, T. Chanyaswad, S. Wang, and P. Mittal, "Investigating statistical privacy frameworks from the perspective of hypothesis testing." *Proc. Priv. Enhancing Technol.*, vol. 2019, no. 3, pp. 233–254, 2019.

[68] L. Lyu, H. Yu, and Q. Yang, "Threats to federated learning: A survey," *arXiv preprint arXiv:2003.02133*, 2020.

[69] P. Ma, Z. Liu, Y. Yuan, and S. Wang, "Neurald: Detecting indistinguishability violations of oblivious ram with neural distinguishers," *IEEE Transactions on Information Forensics and Security*, vol. 17, pp. 982–997, 2022.

[70] A. Madry, A. Makelov, L. Schmidt, D. Tsipras, and A. Vladu, "Towards deep learning models resistant to adversarial attacks," *arXiv preprint arXiv:1706.06083*, 2017.

[71] B. McMahan, E. Moore, D. Ramage, S. Hampson, and B. A. y Arcas, "Communication-efficient learning of deep networks from decentralized data," in *Artificial Intelligence and Statistics*. PMLR, 2017, pp. 1273–1282.

[72] F. Mo, A. S. Shamsabadi, K. Katevas, S. Demetriou, I. Leontiadis, A. Cavallaro, and H. Haddadi, "Darknetz: towards model privacy at the edge using trusted execution environments," in *Proceedings of the 18th International Conference on Mobile Systems, Applications, and Services*, 2020, pp. 161–174.

[73] P. Mohassel and P. Rindal, "Aby3: A mixed protocol framework for machine learning," in *Proceedings of the 2018 ACM SIGSAC conference on computer and communications security*, 2018, pp. 35–52.

[74] S. Moro, P. Cortez, and P. Rita, "A data-driven approach to predict the success of bank telemarketing," *Decision Support Systems*, vol. 62, pp. 22–31, 2014.

[75] I. News, "Enveil raises 25 million to expand its footprint in both the commercial and government markets," https://www.helpnetsecurity.com/2022/04/29/enveil-funding/, 2022.

[76] S. Nilizadeh, Y. Noller, and C. S. Pasareanu, "DifFuzz: differential fuzzing for side-channel analysis," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 176–187.

[77] R. Nock, S. Hardy, W. Henecka, H. Ivey-Law, G. Patrini, G. Smith, and B. Thorne, "Entity resolution and federated learning get a federated resolution," *arXiv preprint arXiv:1803.04035*, 2018.

[78] N. Papernot, P. McDaniel, I. Goodfellow, S. Jha, Z. B. Celik, and A. Swami, "Practical black-box attacks against machine learning," ser. AsiaCCS, 2017.

[79] K. Pei, Y. Cao, J. Yang, and S. Jana, "DeepXplore: Automated whitebox testing of deep learning systems," in *Proceedings of the 26th Symposium on Operating Systems Principles*, ser. SOSP '17. New York, NY, USA: ACM, 2017, pp. 1–18. [Online]. Available: http://doi.acm.org/10.1145/3132747.3132785

[80] C. Privacy, "Cape privacy launches self-service enterprise solution to enable secure predictions," shorturl.at/bvEMR, 2022.

[81] D. Rathee, A. Bhattacharya, R. Sharma, D. Gupta, N. Chandran, and A. Rastogi, "SecFloat: Accurate floating-point meets secure 2-party computation," ser. IEEE S&P, 2022.

[82] D. Rathee, M. Rathee, R. K. K. Goli, D. Gupta, R. Sharma, N. Chandran, and A. Rastogi, "Sirnn: A math library for secure rnn inference," in *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2021, pp. 1003–1020.

[83] T. J. Rivlin, *Chebyshev polynomials*. Courier Dover Publications, 2020.

[84] W. Ruan, M. Xu, W. Fang, L. Wang, L. Wang, and W. Han, "Private, efficient, and accurate: Protecting models trained by multi-party learning with differential privacy," in *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2023, pp. 1926–1943.

[85] W. Samek, T. Wiegand, and K.-R. Müller, "Explainable artificial intelligence: Understanding, visualizing and interpreting deep learning models," *arXiv preprint arXiv:1708.08296*, 2017.

[86] A. Schulz, F. Hinder, and B. Hammer, "Deepview: Visualizing classification boundaries of deep neural networks as scatter plots using discriminative dimensionality reduction," in *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI-20*, C. Bessiere, Ed. International Joint Conferences on Artificial Intelligence Organization, 7 2020, pp. 2305–2311, main track. [Online]. Available: https://doi.org/10.24963/ijcai.2020/319

[87] F. Suya, J. Chi, D. Evans, and Y. Tian, "Hybrid batch attacks: Finding black-box adversarial examples with limited queries," ser. {USENIX} Security, 2020.

[88] SYNCED, "Shared machine learning: Ant financial's solution for data privacy," https://syncedreview.com/2019/08/22/shared-machine-learning-ant-financials-solution-for-data-privacy/, 2019.

[89] D. Taylor, "Making intractable data interactable, bitfount raises 5 million," shorturl.at/gkCQ1, 2022.

[90] F. Tramer and D. Boneh, "Slalom: Fast, verifiable and private execution of neural networks in trusted hardware," *arXiv preprint arXiv:1806.03287*, 2018.

[91] Trofi, "Trofi set to launch its platform for users to access crypto products and services," shorturl.at/lsxQ7, 2022.

[92] S. Wagh, D. Gupta, and N. Chandran, "Securenn: Efficient and private neural network training," *Cryptology ePrint Archive*, 2018.

[93] Y. Wang, Z. Ding, D. Kifer, and D. Zhang, "Checkdp: An automated and integrated approach for proving differential privacy or finding precise counterexamples," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, 2020, pp. 919–938.

[94] Y. Wang, Z. Ding, G. Wang, D. Kifer, and D. Zhang, "Proving differential privacy with shadow execution," in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2019, pp. 655–669.

[95] T. Yadav and M. Kumar, "Differential-ml distinguisher: Machine learning based generic extension for differential cryptanalysis," in *Progress in Cryptology – LATINCRYPT 2021*, P. Longa and C. Ràfols, Eds., 2021, pp. 191–212.

[96] Q. Yang, Y. Liu, T. Chen, and Y. Tong, "Federated machine learning: Concept and applications," *ACM Transactions on Intelligent Systems and Technology (TIST)*, vol. 10, no. 2, pp. 1–19, 2019.

[97] A. C.-C. Yao, "How to generate and exchange secrets," in *27th Annual Symposium on Foundations of Computer Science (sfcs 1986)*. IEEE, 1986, pp. 162–167.

[98] I.-C. Yeh and C.-h. Lien, "The comparisons of data mining techniques for the predictive accuracy of probability of default of credit card clients," *Expert systems with applications*, vol. 36, no. 2, pp. 2473–2480, 2009.

[99] M. Zalewski, "American Fuzzy Lop," https://lcamtuf.coredump.cx/afl/, 2021.

[100] H. Zhang, E. Roth, A. Haeberlen, B. C. Pierce, and A. Roth, "Testing differential privacy with dual interpreters," *arXiv preprint arXiv:2010.04126*, 2020.

APPENDIX A
NON-LINEAR FUNCTION APPROXIMATION STRATEGIES

In this section, we present the commonly used approximation strategies for non-linear functions in DL models.

Exponential Function $exp(x)$. Exponential function is approximated by a limit approximation:

$$exp(x) = \lim_{n\to\infty}(1 + x/n)^n$$

This approximation strategy is the same for CrypTen, TF-Encrypted and PySyft.

Reciprocal Function $1/x$. One strategy to approximate the reciprocal function is Newton-Raphson approximation:

$$y_{i+1} = 2y_i - y_i^2 x,$$

where y_0 is usually set to $y_0 = 3exp(0.5 - x) + 0.003$ to speed up the convergence. Following the recursion formula above, we can get an accurate result of $y_n = 1/x$ if n is large enough. This strategy is adopted in CrypTen and TF-Encrypted to calculate the batch normalization and other non-linear functions' approximations.

PySyft adopts the following iteration-based method to calculate the batch normalization:

$$y_{i+1} = y_i(C + 1 - xy_i^2)/C,$$

where $y_0 = (C + 1 - x)/C$ and C is the maximal value that the input value $|x|$ can be. With large enough iteration number n, we can get an accurate result of $y_n = 1/x$.

Sigmoid Function $Sigmoid(x) = \frac{1}{1+e^{-x}}$. One strategy to approximate the Sigmoid function is to use the aforementioned methods to approximate the exponential function and reciprocal function, and then Sigmoid can be naturally calculated. This strategy is adopted by all the three frameworks.

Another strategy is to approximate function $Tanh(x)$ first by Chebyshev polynomials and then convert the results to Sigmoid as the following holds:

$$Sigmoid(x) = 0.5Tanh(\frac{x}{2}) + 0.5$$

Tanh Function $Tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$. As we mentioned above, $Tanh(x)$ can be approximated by Chebyshev polynomials. We illustrate the procedure as following:

$$Tanh(x) = \sum_{j=1}^{n} c_{2j-1} P_{2j-1}(x),$$

where c_i is the ith Chebyshev series coefficient and P_i is ith polynomial. The result is further truncated to $[-1, 1]$. This strategy is also adopted by all these three frameworks.

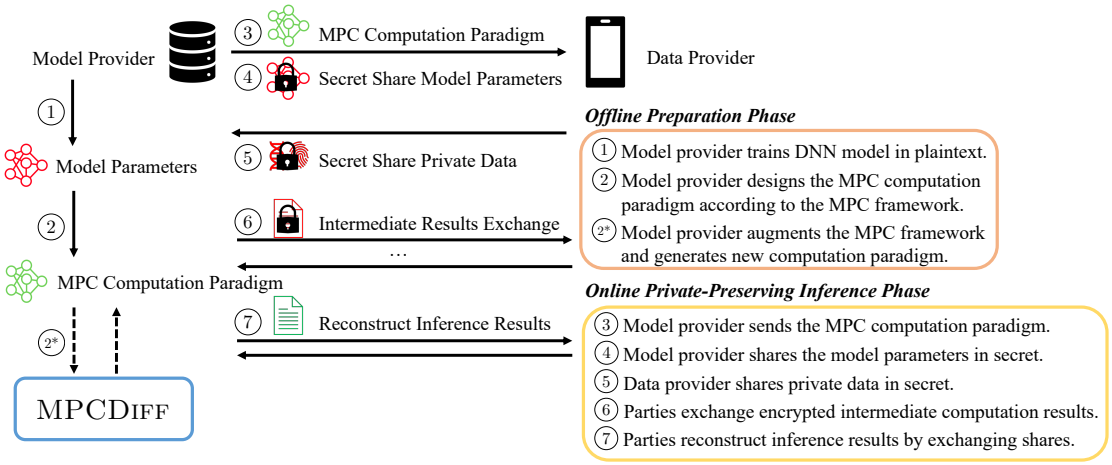


Fig. 10: Pipeline of MPC-protected DL inference over pre-trained models.

Similar to $Sigmoid(x)$, $Tanh(x)$ can also be approximated by $Sigmoid(x)$ as the following always holds:

$$Tanh(x) = 2Sigmoid(2x) - 1$$

GELU Function $Gelu(x) = x\Phi(x)$**.** GELU function is the result of multiplication of the input and the cumulative distribution function of standard Gaussian $\Phi(x)$. This function is approximated by the following strategy:

$$Gelu(x) = 0.5x(1 + Tanh(\sqrt{\frac{2}{\pi}}(x + 0.044715x^3))),$$

where we use $Sigmoid$ to approximate the $Tanh$ function when calculating $Gelu(x)$.

Appendix B
Pipeline of Privacy-Preserving DL Inference

This appendix section depicts the pipeline of launching privacy-preserving DL inference with a pre-trained DL model. As shown in Fig. 10, in the offline stage, the model provider will train a DL model using its training dataset in plaintext (Step ①). Then, the model provider designs the MPC computation paradigm according to the model architecture and the MPC framework (Step ②). MPCDIFF can be leveraged after Step ②, delivering testing and repairing towards the MPC-hardened models. We mark this step as ②* in Fig. 10.

After the offline phase, the model provider will send the (repaired) MPC computation paradigm to the data provider (Step ③). And the model provider will share his model parameters with the data provider in secret (Step ④). Also, the data provider will share his private data with the model provider in secret (Step ⑤). Then, in Step ⑥, the two parties will exchange intermediate sharings and perform computation according to the MPC computation paradigm specificed by the model provider in Step ③. Finally, the parties will reconstruct the inference results by exchanging the sharings in Step ⑦.

Appendix C
Meta Strategy for Non-Linear Function Approximation

As mentioned in our main paper, we adopt a meta strategy in MPCDIFF, which subsumes detailed approaches to tuning terms for various approximation methods. Table IV lists the approximation terms' tuning configurations in MPCDIFF. Developers are suggested to reuse our configuration for similar tasks. Nevertheless, it is also feasible to configure MPCDIFF by updating the meta strategy with new configurations and tuning tactics according to specific computing scenarios, computation/communication cost budgets, and model performance requirements.

Appendix D
Detailed Architecture of the Models

In Table V, we present the detailed architectures of all the models we test in Sec. VI.

TABLE IV: Meta strategy of tuning terms for different non-linear functions.

MPC Frameworks	Non-Linear Functions	Approximation Strategy	Term Before / After Repair
CrypTen	$exp(x)$	limit approximation	$6/9$
	$1/x$	Newton-Raphson approximation	$6/9$
	$Sigmoid(x)$	combine exponential and reciprocal approximation	$6, 6/9, 9$
	$Tanh(x)$	$2Sigmoid(2x) - 1$	same as $Sigmoid(x)$
	$Gelu(x)$	$0.5x(1 + Tanh(\sqrt{\frac{2}{\pi}}(x + 0.044715x^3)))$	same as $Tanh(x)$
TF-Encrypted	$exp(x)$	limit approximation	$3/5$
	$1/x$	Newton-Raphson approximation	$3/4$
	$Sigmoid(x)$	combine exponential and reciprocal approximation	$3, 3/5, 4$
	$Tanh(x)$	$2Sigmoid(2x) - 1$	same as $Sigmoid(x)$
	$Gelu(x)$	$0.5x(1 + Tanh(\sqrt{\frac{2}{\pi}}(x + 0.044715x^3)))$	same as $Tanh(x)$
PySyft	$exp(x)$	limit approximation	$6/9$
	$1/x$	iteration-based method (for batch normalization)	$40, C = 20/80, C = 20$
	$1/x$	Newton-Raphson approximation (for activation functions)	$6/9$
	$Sigmoid(x)$	combine exponential and Newton-Raphson-based reciprocal approximation	$6, 6/9, 9$
	$Tanh(x)$	$2Sigmoid(2x) - 1$	same as $Sigmoid(x)$
	$Gelu(x)$	$0.5x(1 + Tanh(\sqrt{\frac{2}{\pi}}(x + 0.044715x^3)))$	same as $Tanh(x)$

TABLE V: Detailed architectures of the models.

Models	Layers
LeNet (CrypTen, TF-Encrypted)	Conv(input_channel=1, output_channel=6, kernel_size=5, stride=1, padding=2) Batch Normalization Sigmoid Activation AvgPool(kernel_size=2, stride=2) Conv(input_channel=6, output_channel=16, kernel_size=5, stride=1, padding=0) Batch Normalization Sigmoid Activation AvgPool(kernel_size=2, stride=2) Linear(input_shape=400, output_shape=120) Batch Normalization Sigmoid Activation Linear(input_shape=120, output_shape=84) Batch Normalization Sigmoid Activation Linear(input_shape=84, output_shape=10)
LeNet (PySyft)	Conv(input_channel=1, output_channel=6, kernel_size=5, stride=1, padding=2) Batch Normalization Sigmoid Activation AvgPool(kernel_size=2, stride=2) Conv(input_channel=6, output_channel=16, kernel_size=5, stride=1, padding=0) Batch Normalization Sigmoid Activation AvgPool(kernel_size=2, stride=2) Linear(input_shape=400, output_shape=120) Sigmoid Activation Linear(input_shape=120, output_shape=84) Sigmoid Activation Linear(input_shape=84, output_shape=10)
MLP-Sigmoid	Linear(input_shape=23, output_shape=120) Sigmoid Activation Linear(input_shape=120, output_shape=2)
MLP-GELU	Linear(input_shape=20, output_shape=250) GELU Activation Linear(input_shape=250, output_shape=2)