Is MPC Secure? Leveraging Neural Network Classifiers to Detect Data Leakage Vulnerabilities in MPC Implementations

Guopeng Lin*, Xiaoning Du†, Lushan Song*, Weili Han*, Jin Tan‡, Junming Ma‡, Wenjing Fang‡, Lei Wang‡
*Fudan University, †Monash University, ‡Ant Group
*{17302010022, 19110240022, wlhan}@fudan.edu.cn, †xiaoning.du@monash.edu,
‡{tanjin.tj, jimi.mjm, bean.fwj, shensi.wl}@antgroup.com

***Abstract*—Due to the emerging privacy-protection laws and regulations (e.g. GDPR in the EU) in recent years, dozens of multi-party computation (MPC for short) protocols have been proposed and widely applied by companies and institutions. These MPC protocols enable companies and institutions to perform joint analyses and machine learning on their private data while protecting their data's privacy. However, due to the complexity of MPC protocols, their implementations often contain data leakage vulnerabilities, which can critically undermine the intended privacy protection. Additionally, most existing security analyses of MPC protocols rely on theoretical proofs, neglecting to detect possible vulnerabilities in MPC implementations. Therefore, detecting data leakage vulnerabilities in MPC implementations is an urgent necessity.**

In this paper, we propose `MPCGuard`, a practical framework for detecting data leakage vulnerabilities in MPC implementations. Different from traditional memory vulnerabilities, data leakage vulnerabilities in MPC implementations cannot be identified by existing sanitizers. To resolve this challenge, we first establish a leakage identifier in `MPCGuard` with two neural network classifiers to identify whether an MPC implementation contains data leakage vulnerabilities. To enhance identification effectiveness, the structures of neural network classifiers are designed according to the characteristics of MPC protocols. After identifying a data leakage vulnerability, we employ a delta method to assist in locating the vulnerability.

To demonstrate the effectiveness of `MPCGuard`, we apply `MPCGuard` to test 29 commonly-used MPC implementations in three main-stream MPC frameworks, i.e. `Crypten`, `TF-Encrypted`, and `MP-SPDZ`. We discover that 12 out of 29 implementations contain data leakage vulnerabilities, some of which can lead to the reconstruction of raw data. Until the moment this paper is written, all vulnerabilities, two of which have been assigned with CVE-IDs, have been confirmed. To the best of our knowledge, these two CVE-IDs are the first CVE-IDs assigned for data leakage vulnerabilities in MPC implementations.

1. Introduction

In recent years, countries and regions across the world have enacted laws and regulations (e.g. GPDR [1]) to protect data security and privacy. These laws and regulations impose strict penalties for data leakage, compelling companies and institutions to employ privacy-preserving technologies, such as multi-party computation (MPC for short), federated learning, and trusted execution environment, to prevent data leakage. Among these technologies, MPC has emerged as a key solution due to its rigorous theoretical security guarantees.

MPC enables multiple parties to perform joint analyses and machine learning on their private data to obtain valuable statistical results and machine learning models while preserving data privacy. For instance, the Italian National Institute of Statistics and the Bank of Italy employ MPC protocols to conduct socio-economic analyses while protecting citizens' financial privacy [2]. Such a collaborative approach ensures that parties' data remains private, thus adhering to privacy-protection laws and regulations.

However, due to the complexity of MPC protocols, their implementations often contain data leakage vulnerabilities, which can critically undermine the intended privacy protection. For instance, our testing results (shown in Section 5.2) reveal that `Crypten` [3], a main-stream MPC framework developed by Facebook, contains a data leakage vulnerability in the implementation of the multiplication protocol. This vulnerability arises from the omission of a crucial step, where zero-random shares are generated to mask the result shares, and allows an adversary to reconstruct raw data. Additionally, another main-stream MPC framework, `TF-Encrypted`, contains a data leakage vulnerability in the implementation of the 'Less Than Zero' protocol due to an incorrect parameter type in the random number generation function—a subtle mistake that is difficult to notice. This vulnerability allows an adversary to reconstruct some bits of the raw data.

Moreover, most existing security analyses [4], [5], [6], [7] for MPC protocols rely on theoretical proofs, neglecting potential vulnerabilities in their implementations. As a result, vulnerabilities in practical MPC implementations are often overlooked. Therefore, there remains an urgent need to develop a dedicated framework for detecting data leakage vulnerabilities in MPC implementations.

In this paper, we are motivated to propose `MPCGuard`, a practical framework designed to detect data leakage vulnerabilities in MPC implementations. Different from traditional memory vulnerabilities, data leakage vulnerabilities in MPC implementations cannot be identified by existing sanitizers (e.g. [8], [9]). To resolve this challenge, we establish a leakage identifier in `MPCGuard` with two neural network classifiers to identify whether an MPC implementation contains data leakage vulnerabilities. Specifically, we first collect lots of adversary's views (each consists of the views of its corrupted parties) in both the real and ideal worlds by repeatedly executing the MPC implementation and its ideal functionality, respectively. We then split these views into training and test datasets for both worlds. Subsequently, we train and test two neural network classifiers independently: one on the real-world datasets and the other on the ideal-world datasets to simulate the adversary's inference capabilities in the real world and the ideal world, respectively. If the neural network classifier trained on real-world datasets significantly outperforms the one trained on ideal-world datasets, it indicates that the adversary can gain more information about the private data of honest parties in the real world than in the ideal world, which indicates a data leakage vulnerability. Moreover, since neural networks are not inherently well-suited for identifying data leakage vulnerabilities in MPC implementations, we further design the structure of the neural network classifiers according to the characteristics of MPC protocols to enhance detection effectiveness.

After identifying a data leakage vulnerability, we further employ a delta method to assist in locating the vulnerability. Specifically, we recursively truncate the length of the adversary's views in half until we find the smallest length that can detect data leakage. We then re-execute the MPC implementation and record the call stack when the adversary view reaches the smallest length. The call stack should contain the code location responsible for the data leakage.

To demonstrate the effectiveness of `MPCGuard` [1], we apply `MPCGuard` to test 29 commonly-used MPC implementations in three main-stream MPC frameworks, i.e. `Crypten` [3], `TF-Encrypted` [10], and `MP-SPDZ` [11]. Among these three frameworks, `Crypten` and `TF-Encrypted` are developed by industry companies. During the test, we discover that 12 out of 29 implementations contain data leakage vulnerabilities, some of which can lead to direct reconstruction of raw data. Until the moment this paper is written, all vulnerabilities have been confirmed, and two have been assigned with CVE-IDs. To the best of our knowledge, these two CVE-IDs are the first CVE-IDs assigned for data leakage vulnerabilities in MPC implementations. These results highlight the significant risks present in MPC implementations.

We summarize the main contributions of this paper as follows:

- We design `MPCGuard`, a practical framework to detect data leakage vulnerabilities in MPC implementations. `MPCGuard` first leverages neural network classifiers whose structures are designed according to the characteristics of MPC protocols to identify data leakage vulnerabilities, and then employs a delta method to assist in locating the vulnerability.
- Leveraging `MPCGuard`, we discover that 12 commonly-used MPC implementations in three main-stream MPC frameworks contain data leakage vulnerabilities, and report these vulnerabilities to the developers.

Roadmap: In Section 2, we introduce the background knowledge about MPC involved in this paper. Section 3 provides an overview of `MPCGuard`, followed by a detailed design explanation in Section 4. The performance of `MPCGuard` is evaluated in Section 5. Section 6 discusses the limitations of `MPCGuard`, suggestions for MPC implementations, and future work. We investigate related work in Section 7 and conclude this paper in Section 8.

2. Background of MPC

2.1. Brief Overview

MPC is a cryptographic technique that enables multiple parties to jointly compute a function over their data while keeping their data private. Unlike traditional cryptographic technologies, which focus on protecting the security and integrity of communication and typically assume that adversaries are external eavesdroppers, MPC is designed to protect the privacy of each party's data even when the adversary may be one or more of the participating parties. For example, the well-known MPC protocol, the millionaire protocol, allows two millionaires to determine who is wealthier while keeping their actual wealth private. This protocol operates under the assumption that either of the two millionaires may act as an adversary.

Formally, given a set of parties $\{P_1, P_2, \cdots, P_n\}$, each holding a private input x_i, an MPC protocol enables these parties to jointly compute a function f over their inputs $x_1, x_2, \cdots, x_n$, resulting in outputs $y_1, y_2, \cdots, y_n = f(x_1, x_2, \cdots, x_n)$. The protocol ensures that, aside from each party's output y_i and any information that can be inferred from y_i, no party learns any additional information about the inputs or outputs of other parties.

To combine multiple MPC protocols to achieve complex computation tasks (e.g. training machine learning models) while keeping data private throughout the whole computation process, MPC protocols are usually built on secret sharing [12]. Secret sharing is a cryptographic technique used to distribute a secret among a group of parties. To secret share a secret x, x is first split into shares $[\![x]\!]_1, \cdots, [\![x]\!]_n$, where each share individually reveals no information about x. These shares $[\![x]\!]_i$ are then distributed to the parties. The secret can only be reconstructed when a sufficient number of parties combine their shares. For example, in additive secret sharing (ASS for short), each party P_i receives a share $[\![x]\!]_i$, where $(\sum_{i=1}^{n} [\![x]\!]_i) \mod p = x$ (p is the size of the computation domain). In ASS, the secret can be reconstructed if all parties combine their shares. Besides, in replicated secret sharing (RSS for short), which is often used in three-party protocols, each party receives a pair of

1. We have open-sourced the implementation of `MPCGuard` at https://github.com/FudanMPL/MPCGuard.

ASS shares: P_1 receives $[\![x]\!]_1$ and $[\![x]\!]_2$, P_2 receives $[\![x]\!]_2$ and $[\![x]\!]_3$, and P_3 receives $[\![x]\!]_3$ and $[\![x]\!]_1$. In RSS, the secret can be reconstructed if any two parties combine their shares.

To further elucidate how an MPC protocol operates, we introduce a commonly used RSS-based multiplication protocol. As shown in Protocol 1, this protocol takes shares $[\![x]\!]_i$, $[\![x]\!]_{i+1}$, $[\![y]\!]_i$, and $[\![y]\!]_{i+1}$ from each party and outputs shares $[\![z]\!]_i$ and $[\![z]\!]_{i+1}$, where $z = x * y$, to each party. By leveraging RSS to represent inputs and outputs, the protocol can seamlessly accept inputs from other protocols and provide outputs that serve as inputs for subsequent protocols. In this protocol, the parties first generate zero-sum random values, r_1, r_2, and r_3, where $(r_1 + r_2 + r_3) \mod p = 0$ (Line 1-3), using seeds negotiated during the setup phase. They then multiply the shares of x and y, and mask the results with the zero-sum randoms to get the shares of z (Line 4-6). Finally, the parties exchange the shares of z, so that each party holds a pair of shares (Line 7).

Protocol 1: *RSSMul*

Input:

- P_1 inputs $[\![x]\!]_1$, $[\![x]\!]_2$, $[\![y]\!]_1$ and $[\![y]\!]_2$. P_2 inputs $[\![x]\!]_2$, $[\![x]\!]_3$, $[\![y]\!]_2$, and $[\![y]\!]_3$. and P_3 inputs $[\![x]\!]_3$, $[\![x]\!]_1$, $[\![y]\!]_3$, and $[\![y]\!]_1$.

Output:

- P_1 obtains $[\![z]\!]_1$ and $[\![z]\!]_2$. P_2 obtains $[\![z]\!]_2$ and $[\![z]\!]_3$. P_3 obtains $[\![z]\!]_3$ and $[\![z]\!]_1$, such that $z = x * y$.

Setup Phase:

- P_1 and P_2 negotiate a random seed s_{12}, P_1 and P_3 negotiate a random seed s_{13}, P_2 and P_3 negotiate a random seed s_{23}

Computation:

1. P_1 generate a random number r_{12} with s_{12} and a random number r_{13} with s_{13}, and compute $r_1 = (r_{12} + r_{13}) \mod p$.
2. P_2 generate a random number r_{12} with s_{12} and a random number r_{23} with s_{23}, and compute $r_2 = (-r_{12} + r_{23}) \mod p$.
3. P_3 generate a random number r_{13} with s_{13} and a random number r_{23} with s_{23}, and compute $r_3 = (-r_{13} - r_{23}) \mod p$.
4. P_1 computes $[\![z]\!]_1 = ([\![x]\!]_1 * [\![y]\!]_1 + [\![x]\!]_1 * [\![y]\!]_2 + [\![x]\!]_2 * [\![y]\!]_1 + r_1)\ mod\ p$.
5. P_2 computes $[\![z]\!]_2 = ([\![x]\!]_2 * [\![y]\!]_2 + [\![x]\!]_2 * [\![y]\!]_3 + [\![x]\!]_3 * [\![y]\!]_2 + r_2)\ mod\ p$.
6. P_3 computes $[\![z]\!]_3 = ([\![x]\!]_3 * [\![y]\!]_3 + [\![x]\!]_3 * [\![y]\!]_1 + [\![x]\!]_1 * [\![y]\!]_3 + r_3)\ mod\ p$.
7. P_1 sends $[\![z]\!]_1$ to P_3, P_2 sends $[\![z]\!]_2$ to P_1, and P_3 sends $[\![z]\!]_3$ to P_2.

2.2. Data Leakage Vulnerability

In this subsection, we first introduce the real-ideal paradigm, which is widely used to define and prove the security of MPC protocols. Then, we introduce the semi-honest adversary, which is commonly used in MPC. Finally, we present the definition of data leakage vulnerabilities in MPC protocols.

Note that this paper focuses on detecting data leakage vulnerabilities in MPC implementations in the presence of a semi-honest adversary. Preliminary results on extending our approach to simulate a malicious adversary for testing MPC implementations are discussed in Section 6.

Real-Ideal Paradigm. The real-ideal paradigm provides a framework to define and analyze the security of an MPC protocol by comparing its behavior in an ideal world with a completely trusted party to that in a real world without such trust. In the ideal world, the parties securely compute a function F with the help of a completely trusted party, referred to as the functionality $\mathcal{F}$. Concretely, each party P_i privately sends its inputs x_i to $\mathcal{F}$. And the functionality$\mathcal{F}$ computes $y_1, \ldots, y_n = F(x_1, \ldots, x_n)$ and returns each y_i to each party P_i. In the real world, there is no trusted party. Instead, all parties communicate with each other using a protocol π. The protocol π specifies a function π_i for each party P_i. This function takes as input a security parameter κ, the party's private input x_i, a random tape, and the list of messages P_i has received so far. Then, π_i outputs either a message to send along with its destination or instructs P_i to terminate with a specific output.

In the ideal world, the view of a party only consists of its input x_i and the output y_i received from $\mathcal{F}$, while in the real world, the view of a party consists of its private input, its random tape, and all messages it received during the protocol.

Besides, the view of an adversary consists of the views of its corrupted parties.

Semi-Honest Adversary. A semi-honest adversary is one who controls its corrupted parties to correctly execute the protocol but tries to learn as much as possible from its view.

For a protocol to be secure in the presence of a semi-honest adversary, it must be possible for the adversary in the ideal world to generate a view indistinguishable from the real-world adversary's view. That is because, in this case, any information about the private data (inputs and outputs) of honest parties that the adversary can infer from its view in the real world can also be inferred from its view in the ideal world. We refer to such an ideal-world adversary as a simulator, since it generates a "simulated" real-world view.

Formally, let π be a protocol and $\mathcal{F}$ be a functionality. Let C be the set of corrupted parties, and $SimAlg$ be a simulator algorithm. We define the following distributions of random variables:

- $Real(\kappa, \pi, C; x_1, \ldots, x_n)$: Execute the protocol π with security parameter κ, where each party P_i executes the protocol honestly using its private input x_i. Let V_i denote the view of party P_i, output $\{V_i \mid i \in C\}$.
- $Ideal_{\mathcal{F},SimAlg}(\pi, C; x_1, \ldots, x_n)$: Compute $(y_1, \ldots, y_n) = \mathcal{F}(x_1, \ldots, x_n)$, and output $SimAlg((x_i, y_i) \mid i \in C)$.

An MPC protocol is secure against semi-honest adversaries if there exists a simulator algorithm $SimAlg$

such that, for every subset of corrupt parties C and all inputs $x_1, \ldots, x_n$, the distributions $Real(\pi, C; x_1, \ldots, x_n)$ and $Ideal_{\mathcal{F},SimAlg}(\pi, C; x_1, \ldots, x_n)$ are indistinguishable.

However, proving the complete correctness and security of implementation through software testing is fundamentally impossible [13]. Therefore, in this paper, we focus on detecting whether an MPC implementation contains a data leakage vulnerability.

Given an implementation $\mathcal{I}$ of an MPC protocol π that realizes a functionality $\mathcal{F}$, we say that $\mathcal{I}$ contains a data leakage vulnerability if an adversary can obtain more information about the inputs or outputs of honest parties with the view from executing $\mathcal{I}$ than with the view from executing $\mathcal{F}$.

Formally, let

- $Real_{\mathcal{I},C}(\kappa, x_1, \ldots, x_n)$: Execute the implementation $\mathcal{I}$ with security κ and the input of each party x_i. Let V_i denote the view of party P_i, output $\{V_i \mid i \in C\}$.
- $Ideal_{\mathcal{F}}(\pi, C; x_1, \ldots, x_n)$: Compute $(y_1, \ldots, y_n) = \mathcal{F}(x_1, \ldots, x_n)$, and output $\{x_i, y_i \mid i \in C\}$.

Let O be the set of honest parties, and C be the set of corrupted parties. An implementation $\mathcal{I}$ is said to contain a data leakage vulnerability if there exists a polynomial-time probabilistic algorithm $\mathcal{A}$, a value $t \in \{x_i, y_i \mid i \in O\}$, a computable function ϕ, and a non-negligible function $\epsilon(k)$ such that, for any polynomial-time probabilistic algorithm $\mathcal{A}'$, the following inequality holds:

$$|\Pr[\mathcal{A}(Real_{\mathcal{I},C}(x_1, \ldots, x_n)) = \phi(t)] - \Pr[\mathcal{A}'(Ideal_{\mathcal{F}}(\pi, C; x_1, \ldots, x_n)) = \phi(t)]| \geq \epsilon(k)$$

2.3. Vulnerability Example

To demonstrate how data leakage vulnerabilities can arise and their severe impact, we present a representative vulnerability identified by our proposed framework, MPCGuard. This vulnerability exists in the RSS-based multiplication protocol implementation in Crypten [3], and arises from the omission of a critical step: generating zero-random shares to mask the result shares.

According to the design (shown in Protocol 1, can be treated as the correct implementation) of the protocol, in order to keep the results totally private, it is essential to generate the zero-sum random values and use them to mask the multiplication result. However, as is shown in Figure 1, this step is missing in the wrong implementation of Crypten, which results in a data leakage vulnerability. This vulnerability allows an adversary controlling a corrupted party (e.g. P_1) to reconstruct the raw value of one input when it knows the shares of the other input. For example, if P_1 knows $[\![y]\!]_1$, $[\![y]\!]_2$ and $[\![y]\!]_3$ are all equal to 1. When a multiplication of $x*y$ is performed, P_1 can directly reconstruct x. Specifically, P_1 initially knows $[\![x]\!]_1$ and $[\![x]\!]_2$. During the multiplication, P_2 sends $[\![z]\!]_2 = ([\![x]\!]_2 * 1 + [\![x]\!]_2 * 1 + [\![x]\!]_3 * 1) \mod p$ to P_1, which enables P_1 to calculate $[\![x]\!]_3$. Consequently, P_1 can compute x by $x = ([\![x]\!]_1 + [\![x]\!]_2 + [\![x]\!]_3) \mod p$. Note that it is reasonable to assume that an adversary might know the shares of one input (x or y), as one of these inputs could be directly provided by the adversary.

```
Initial State:
• $P_1$ holds $[\![x]\!]_1$, $[\![x]\!]_2$, $[\![y]\!]_1$ and $[\![y]\!]_2$
• $P_2$ holds $[\![x]\!]_2$, $[\![x]\!]_3$, $[\![y]\!]_2$ and $[\![y]\!]_3$
• $P_3$ holds $[\![x]\!]_3$, $[\![x]\!]_3$, $[\![y]\!]_3$ and $[\![y]\!]_1$
Computation Process:
/*
Here, the steps for generating and using zero-sum
randomness are missing.
*/
1. $P_1$ computes $[\![z]\!]_1 = ([\![x]\!]_1 * [\![y]\!]_1 + [\![x]\!]_1 * [\![y]\!]_2 + [\![x]\!]_2 * [\![y]\!]_1)\, mod\ p$
2. $P_2$ computes $[\![z]\!]_2 = ([\![x]\!]_2 * [\![y]\!]_2 + [\![x]\!]_2 * [\![y]\!]_3 + [\![x]\!]_3 * [\![y]\!]_2)\, mod\ p$
3. $P_3$ computes $[\![z]\!]_3 = ([\![x]\!]_3 * [\![y]\!]_3 + [\![x]\!]_3 * [\![y]\!]_1 + [\![x]\!]_1 * [\![y]\!]_3)\, mod\ p$
4. $P_1$ sends $[\![z]\!]_1$ to $P_3$
5. $P_2$ sends $[\![z]\!]_2$ to $P_1$
6. $P_3$ sends $[\![z]\!]_3$ to $P_2$
```

Figure 1: The simplified code of the wrong implementation of the multiplication protocol based on RSS in Crypten.

3. Overview of MPCGuard

3.1. Main Idea

Leakage Identifier. According to the definition of the data leakage vulnerability in Section 2.2, if there exists a data leakage vulnerability in an MPC implementation, there will be a polynomial-time probabilistic algorithm (given the adversary's view) that can predict the inputs or outputs of honest parties with greater accuracy in the real world than any polynomial-time probabilistic algorithm in the ideal world. However, exhaustively testing all polynomial-time probabilistic algorithms is infeasible. To resolve this challenge, we leverage the powerful capabilities of neural networks, which have been shown to learn complex tasks [14] to establish a leakage identifier. Specifically, we first split the adversary's views into a training dataset and a test for both worlds. Then, we train two neural network classifiers separately on the real and ideal training datasets to simulate the adversary's inference capabilities in the real world and the ideal world, respectively. These classifiers are then tested on the test datasets. If the classifier trained on the real-world datasets noticeably outperforms the classifier trained on the ideal-world datasets, it indicates that a data leakage vulnerability exists.

Note that, by definition, proving the existence of a data leakage vulnerability requires demonstrating that a neural network classifier trained on the adversary's view in the real world outperforms any polynomial-time probabilistic algorithm given the adversary's view in the ideal world. In practice, however, the adversary's view in the ideal world is generally smaller and simpler than in the real world, making it easier to learn from. Thus, unless a data leakage vulnerability exists, a neural network classifier should typically achieve better performance in the ideal world than in the real world.

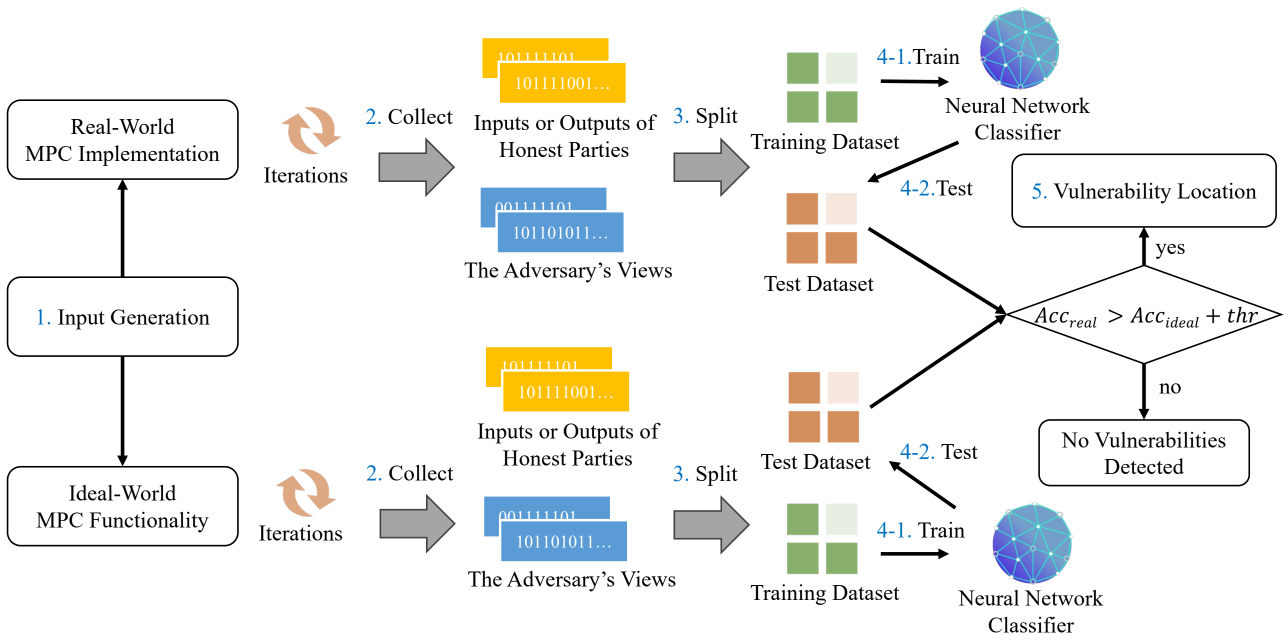

Real-World MPC Implementation
1. Input Generation
Ideal-World MPC Functionality
Iterations
2. Collect
Inputs or Outputs of Honest Parties
The Adversary's Views
101111101
101111001...
001111101
101101011...
3. Split
Training Dataset
Test Dataset
4-1.Train
4-2.Test
Neural Network Classifier
5. Vulnerability Location
$Acc_{real} > Acc_{ideal} + thr$
yes
no
No Vulnerabilities Detected

Figure 2: Workflow of MPCGuard

Vulnerability Location. Once a data leakage vulnerability is identified, we employ a delta method to assist in locating the vulnerability. Specifically, we recursively truncate the length of the adversary's views in half until we find the smallest length that can still detect data leakage. We then re-execute the MPC implementation and record the call stack when the adversary view reaches the smallest length.

As is shown in Figure 2, the workflow of MPCGuard consists of the following five key stages, with the first four stages forming the leakage identifier:

1) **Input Generation:** MPCGuard begins by generating a series of inputs for the MPC protocol, using a combination of heuristic rules and randomization.
2) **Data Collection:** MPCGuard then repeatedly executes the MPC implementation and the corresponding functionality multiple times with the generated inputs. During these executions, the views of the adversary, along with the inputs or outputs of the honest parties are collected.
3) **Dataset Split:** The collected data from both the real world and the ideal world are split into training and test datasets. The features of the training datasets and test datasets consist of the views of the adversary. The labels are set based on the inputs or outputs of the honest parties.
4) **Train and Test:** MPCGuard trains and tests two neural network classifiers separately on the training and test datasets from the real and ideal world. If the accuracy of the neural network classifier trained on real-world datasets is higher than that of the one trained on ideal-world datasets by a threshold ($Acc_{real} > Acc_{ideal} + thr$), it suggests that the adversary's views in the real world leak more information than intended. Note that the accuracy on the training dataset is not a reliable indicator of data leakage vulnerabilities, because neural network classifiers may achieve high accuracy on training data due to overfitting, even if the adversary's views do not actually leak information about the inputs or outputs of honest parties.
5) **Vulnerability Location:** Finally, MPCGuard employs a delta method to assist in locating the vulnerability.

3.2. Scope and Clarification

Main Audiences. The main audiences of this paper are the developers of MPC protocols. They can use MPCGuard to detect potential data leakage vulnerabilities in their MPC implementations and repair them. Previously, detecting such vulnerabilities should be difficult, because it requires the developers to be sufficiently careful and have sufficient knowledge of cryptography.

Gray-Box Tesing. MPCGuard adopts a gray box testing schema. Specifically, MPCGuard is not required to have complete access to the internal structure, code, and design of the MPC implementations. However, it requires instructing some code to the MPC implementations to collect the views of the adversary, including the inputs, outputs, received messages, and random tapes of its corrupted parties.

User's Expectation From MPCGuard. MPCGuard shares a similar focus with most security testing tools (e.g. fuzzers [15], side-channel analyzers [16], differential privacy analyzers [17]) to detect vulnerabilities instead of proving their absence. I.e. MPCGuard cannot offer static verification: when MPCGuard reports no vulnerabilities, we cannot conclude that the MPC implementation is free from data leakage vulnerabilities. Nevertheless, the testing approach provided by `MPCGuard` is precise and is designed to avoid generating false positives by using reasonable running parameters (discussed in Section 4.1).

4. Design of `MPCGuard`

4.1. Leakage Identifier for Input

In this subsection, we present the algorithm of the leakage identifier that identifies leakage to the input of an honest party in an MPC implementation. For the algorithm of the leakage identifier that identifies leakage to the output of an honest party, we present it in Appendix A.

As is shown in Algorithm 1, the algorithm inputs an MPC implementation $\mathcal{I}$ and its corresponding functionality $\mathcal{F}$. The algorithm iteratively generates shares of a secret pair (consists of two secrets) to execute $\mathcal{I}$ and $\mathcal{F}$ to collect the adversary's views (Line 2-7). These views are then split into training and test datasets for both real and ideal worlds, and two neural network classifiers are trained and tested with the datasets (Line 8-11). If the accuracy gap between the real-world and ideal-world classifiers exceeds a threshold, the algorithm returns 'True' (Line 12-14), indicating the presence of a data leakage vulnerability. After all secret pairs are used, if no vulnerability is identified, the algorithm returns 'False' (Line 16).

Below, we present the idea of some key steps of this algorithm.

Input Generation: A simple approach to generate input shares is directly generating multiple random values in the computation domain of MPC protocols. Then, we train neural networks to predict these values. However, this approach can be ineffective because the computation domain of MPC protocols spans a wide range of values, which makes it difficult for the neural networks to converge.

To resolve this challenge, we draw inspiration from the approach proposed by Ma et al. [18] and design a refined input generation approach. Specifically, we first generate a secret pair and then iteratively generate shares of the secret pair to execute the MPC implementation. Using this approach, we only need to perform a binary classification task on the collected views, which improves neural network convergence and the ability to detect data leakage.

Note that the inputs (shares) of honest parties can be deduced from the secrets. Therefore, identifying leakage of the secrets is equivalent to identifying leakage of the inputs of honest parties.

To further increase the likelihood of identifying leakage, we generate secret pairs using the following two methods. The intuition of these methods is that secrets with greater distance are more likely to produce distinguishable views. Therefore, these two methods generate secret pairs based on different distance criteria.

- **Bit Flip:** This method generates pairs with completely different bit patterns. It begins by sampling a random number within the computational domain of the MPC protocol, and then generating another secret by flipping all the bits of the first secret.
- **Value Flip:** This method generates pairs with the maximum value difference. It samples a random number within the computation domain of the MPC protocols and then generates another secret by computing the value most distant from the first secret within the computation domain.

According to our evaluation, if a data leakage vulnerability exists, these two methods usually produce leakage-triggering secret pairs within the first few generated pairs.

Train and Test: During testing neural network classifiers, the randomness of neural network prediction may cause deviations in accuracy even when no data leakage occurs. In order to prevent the false positive in reporting a vulnerability due to the randomness, we use a predefined threshold, thr, to determine when the gap between Acc_{real} and Acc_{ideal} is significant enough to indicate data leakage. Below, we derive how the false positive probability, denoted as P, depends on the threshold thr and the number of samples n in each test dataset.

Assume that, in the absence of leakage, the network behaves like a random classifier with a 0.5 probability of correctly predicting each sample. We can treat the number of correct predictions in each test as a random variable following a binomial distribution. Let X, Y be $\mathrm{Binomial}(n,\, 0.5)$ random variables denoting the numbers of correct predictions in the real and ideal test datasets, respectively.

Because both X and Y have mean $n/2$ and variance $n/4$, and they are independent, their difference $D = X - Y$ satisfies $\mathbb{E}[D] = 0$ and $\mathrm{Var}(D) = \mathrm{Var}(X) + \mathrm{Var}(Y) = n/4 + n/4 = n/2$. For large n, D approximately follows a normal distribution with mean 0 and variance $n/2$, denoted $D \sim \mathcal{N}(0,\, n/2)$.

We report data leakage if $(X/n) - (Y/n) \geq thr$, which is equivalent to $D \geq n\,thr$. Let $Z = D/\sqrt{n/2}$, so $Z \sim \mathcal{N}(0, 1)$. Thus,

$$\begin{aligned} P\big((X/n) - (Y/n) \geq thr\big) &= P\big(D \geq n\,thr\big) \\ &= P\big(Z \geq \sqrt{2n}\,thr\big). \end{aligned}$$

Hence, the false positive probability is

$$P\big(Z \geq \sqrt{2n}\,thr\big).$$

For example, in our evaluation, we set the threshold $thr = 0.1$ and set the round $t = 2000$ to collect a total of 4000 samples. Besides, we split the dataset into training and test datasets with a ratio of 8:2, so each test dataset contains $n = 800$ samples. Using these parameters, the false positive probability is approximately 0.0032%, indicating that false positives are unlikely to occur.

To prevent false positives, in addition to using high thresholds, which may overlook some vulnerabilities, the following two methods can also be adopted: (1) We can perform training and testing multiple times with different

Algorithm 1 Leakage Identifier for Input

Input: An MPC implementation $\mathcal{I}$, and its corresponding functionality $\mathcal{F}$.
Output: `True` if a data leakage vulnerability is identified, and `False` otherwise.

```
1: for $i = 1$ to $k$ do
2:     Generate a pair of secrets $s_1$ and $s_2$.
3:     for $j = 1$ to $t$ do
4:         Generate shares for both $s_1$ and $s_2$.
5:         Execute $\mathcal{I}$ with the shares to collect the adversary's views in the real world.
6:         Execute $\mathcal{F}$ with the shares to collect the adversary's views in the ideal world.
7:     end for
8:     Split the adversary's views in the real world into a training dataset $\mathcal{D}_{train\_real}$ and test dataset $\mathcal{D}_{test\_real}$, where
       each sample is labeled with '0' if its view is generated from $s_1$, otherwise '1'.
9:     Split the adversary's views in the ideal world into training dataset $\mathcal{D}_{train\_ideal}$ and test dataset $\mathcal{D}_{test\_ideal}$, where
       each sample is labeled with '0' if its view is generated from $s_1$, otherwise '1'.
10:    Train a neural network classifier on $\mathcal{D}_{train\_real}$ and test it on $\mathcal{D}_{test\_real}$ to obtain the accuracy $Acc_{real}$.
11:    Train a neural network classifier on $\mathcal{D}_{train\_ideal}$ and test it on $\mathcal{D}_{test\_ideal}$ to obtain the accuracy $Acc_{ideal}$.
12:    if $Acc_{real} > Acc_{ideal} + thr$ then
13:        return True
14:    end if
15: end for
16: return False
```

initial model parameters, then calculate the mean accuracy across these runs to determine Acc_{real} and Acc_{ideal}. This approach helps to mitigate the impact of randomness on the accuracy. (2) We can execute the MPC implementation more times to collect more data. Large training and testing datasets typically reduce the randomness of accuracy. However, these two methods require more time to train and test the models or to execute the MPC implementation.

4.2. Neural Network Classifier

Although we establish a leakage identifier with two neural network classifiers to identify the data leakage vulnerabilities, identifying such vulnerabilities remains challenging due to the massive random numbers in the adversary's view. For example, when testing the equality protocol in `TF-Encrypted`, a single sample consists of 9664 features. Among these features, only a few features contribute to the leakage. However, with so many features, the neural network classifiers struggle to identify the leakage.

We argue that this challenge cannot be resolved by feature selection methods based on feature importance due to the random characteristics of the features. For instance, suppose the input data of an honest party is s, and the features are $f_1 \sim f_{1000}$, all randomly distributed in $[-2^{63}, 2^{63}-1]$. If the leakage pattern is $(f_3 + f_{999}) \mod 2^{63} = s$, evaluating the importance of each individual feature would reveal that none of the single features are related to the secret. Moreover, evaluating the importance of every possible combination of features is unacceptable due to the excessive computational time.

In order to effectively identify data leakage vulnerabilities from massive random numbers, we design the structures of neural network classifiers according to the characteristics of MPC protocols. MPC computations are generally divided into arithmetic computations and Boolean logic computations. Arithmetic computations operate over data in $\mathbf{Z}_p$ (a field of size p), while Boolean logic computations typically deal with bits. Only in a few cases do data over $\mathbf{Z}_p$ and bits mix (such as conversions between Boolean shares and ASS shares). Therefore, as is shown in Figure 3, we first structure the classifier with two distinct branches: one dedicated to learning arithmetic computations and the other dedicated to learning Boolean logic computations. Finally, we combine the outputs from both branches through a fusion layer, enabling the model to learn the mixed operations.

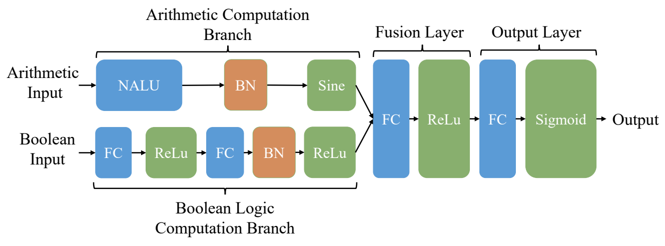


Figure 3: Neural network classifier in `MPCGuard`.

In the arithmetic computation branch, which involves addition, multiplication, and modulo operations, we fit the operations as follows: (1) Addition and multiplication. To fit these operations, we use the neural arithmetic logic unit (NALU for short) [19], which is specifically designed for basic arithmetic functions such as addition and multiplication. (2) Modulo operation. To fit the modulo operation, we preprocess the data by mapping the field values to the range $[-1, 1]$, transforming the modulo operation into a periodic function within this interval. We then use the sine function as the activation function, which has been proven to fit periodic functions more effectively [20].

In the Boolean logic computation branch, which involves XOR, NOT, and AND operations, we fit these operations using two fully connected layers with ReLU activation functions to capture the non-linearities of these operations.

We also add batch normalization to the outputs of both the arithmetic and Boolean logic computations. This step prevents the gradients in one part from becoming too large and causing information from the other part to be ignored.

After obtaining outputs from both the arithmetic and Boolean logic branches, we fuse the outputs to model the mixed computations. This fusion is achieved by concatenating the outputs of both branches and feeding them into a fully connected fusion layer with a ReLU activation function. Finally, we feed the output from the fusion layer into a final output layer to produce a value between 0 and 1 indicating the probability of a data leakage vulnerability.

4.3. Vulnerability Location

To assist in locating the vulnerability, we proceed in two stages: (1) Identify the smallest length of the adversary's views leading to data leakage. To achieve this, we employ a delta method based on binary search. Specifically, we iteratively truncate the length of the collected adversary views in half and employ the approach in Section 4.1 to check if the truncated views can still detect data leakage. This process repeats until we find a length that can detect data leakage, while any smaller length cannot. (2) Locate the code generating the adversary views with the smallest length. To achieve this, we re-execute the MPC implementation, and record the call stack when the adversary view reaches the smallest length. The call stack should contain the code location responsible for the data leakage.

As is shown in Algorithm 2, the vulnerability location algorithm takes as input an MPC implementation, the views of the adversary, the corresponding labels, and the accuracy from the ideal world. It outputs the call stack, which should contain the code location responsible for the data leakage.

The algorithm begins by using a binary search approach to identify the smallest length. It initializes two parameters: l_{succ}, set to the original length of the adversary views, and l_{fail}, set to 0 (Line 1-2). The algorithm then iteratively adjusts these parameters. In each iteration, it calculates the midpoint l_{mid} between l_{succ} and l_{fail} and creates a new view list by truncating the views to the length l_{mid} (Line 4-8). It then creates a training dataset and a test dataset using this new view list and the input labels (Line 9). A neural network classifier is then trained on the training dataset, and evaluated on the test dataset to get the new accuracy Acc_{new} (Line 10). If Acc_{new} exceeds the ideal-world accuracy Acc_{ideal} by a predefined threshold, it indicates that the truncated views still lead to data leakage, so l_{succ} is updated to l_{mid}. Otherwise, l_{fail} is updated to l_{mid} (Line 11-15). This process continues until the difference between l_{succ} and l_{fail} is no greater than 1, at which point l_{succ} indicates the smallest length that still leads to data leakage. Finally, the algorithm re-executes the MPC implementation and records the call stack when the length of the adversary's view reaches l_{succ} (Line 17). This recorded call stack is returned as the output (Line 18).

Algorithm 2 Vulnerability Location

Input: An MPC implementation $\mathcal{I}$, the views of the adversary $views$, the corresponding labels ys, and the accuracy Acc_{ideal} in the ideal world.
Output: The call stack of the data leakage vulnerability.

```
1: $l_{succ} = len(views[0])$
2: $l_{fail} = 0$
3: while $l_{succ} - l_{fail} > 1$ do
4:     $l_{mid} = (l_{succ} + l_{fail})/2$
5:     $views_{new} = \emptyset$
6:     for each $v$ in $views$ do
7:         $views_{new}.add(v[0 : l_{mid}])$
8:     end for
9:     Create a training dataset $\mathcal{D}_{train}$ and a test dataset $\mathcal{D}_{test}$ using $views_{new}$ and $ys$.
10:    Train a neural network classifier on $\mathcal{D}_{train}$ and test it on $\mathcal{D}_{test}$ to obtain the accuracy $Acc_{new}$
11:    if $Acc_{new} > Acc_{ideal} + thr$ then
12:        $l_{succ} = l_{mid}$
13:    else
14:        $l_{fail} = l_{mid}$
15:    end if
16: end while
17: Execute $\mathcal{I}$ again and record the call stack $callstack$ once the length of the adversary's view reaches $l_{succ}$.
18: return $callstack$
```

5. Evaluation

In this section, we evaluate MPCGuard by investigating the following research questions:

- **RQ1:** How effective is MPCGuard in detecting data leakage vulnerabilities in MPC implementations?
- **RQ2:** Compared to other commonly used models, how effective is the neural network classifier designed according to the characteristics of MPC protocols in identifying these vulnerabilities?
- **RQ3:** How efficient is MPCGuard in terms of runtime when detecting data leakage vulnerabilities in MPC implementations?

5.1. Implementation and Experiment Settings

Implementation. We implement MPCGuard in Python 3.10 with more than 1k lines of Python Code, and implement the neural network classifiers using PyTorch [21]. Additionally, for all MPC implementations, we configure the adversary to control the first party, while all secrets are provided by the second party,

Experiment Environment: We conduct experiments on a Linux server equipped with a 32-core 2.4 GHz Intel Xeon CPU, 512GB of RAM, and an NVIDIA GeForce RTX 3080 GPU with 10GB of memory. The system is running CUDA version 12.4 and NVIDIA driver version 550.90.07.

Model Parameters: We set the model parameters of our classifier as follows: Both the Arithmetic and Boolean computation branches contain 128 neurons in each fully connected layer, while the fusion layer contains 256 neurons. The output layer contains a single neuron to produce the final classification result.

Experimental Parameters: We use five secret pairs to test the implementations, and set the threshold $thr = 0.1$ and the round $t = 2000$ to collect a total of 4000 samples. Besides, we split the dataset into training and test datasets with a ratio of 8:2, so each test dataset contains $n = 800$ samples. With these parameter settings, the probability of producing a false positive in vulnerability identification is negligible (approximately 0.0032% detailed in Section 4.1). We further show the evaluation for different thresholds in Appendix C.

Testing Protocols: In this paper, we evaluate the implementations of the following five fundamental MPC protocols based on ASS and RSS.

- **Multiplication Protocol** (*Mul*): This protocol computes the product of two secret values. Given shares of x and y, it outputs shares of $z = x * y$.
- **Linear Protocol** (*Linear*): This protocol computes a linear combination of two secrets with three public constants. Given shares of x and y and three public constants a, b, and c, it outputs shares of $z = a{*}x{+}b{*}y{+}c$.
- **Less-Than-Zero Protocol** (*LTZ*): This protocol performs a comparison between a secret and zero. Given shares of x, it outputs shares of $z = (x < 0)$.
- **Equal-Zero Protocol** (*EQZ*): This protocol checks equality between a secret and zero. Given shares of x, it outputs shares of $z = (x == 0)$.
- **Probabilistic Truncation Protocol** (*Truncpr*): This protocol performs probabilistic truncation to a secret. Given shares of x, it outputs shares of $z = (x \gg f){+}b$, where $b = 1$ or 1.

Testing Frameworks: In this paper, we test the implementations of the above protocols in three widely used open-sourced frameworks, `Crypten` [3], `TF-Encrypted` [3], and `MP-SPDZ` [11].

- `Crypten`, developed by Facebook, is a flexible and efficient MPC framework specifically designed for privacy-preserving machine learning tasks. It offers a high-level interface compatible with PyTorch, enabling users to easily build privacy-preserving machine learning models. With significant support from Facebook, `Crypten` has been widely adopted across diverse application domains. Therefore, it is crucial to ensure the security of `Crypten`.
- `TF-Encrypted`, primarily developed by Alibaba, OpenMined, and Cape, is an open-source MPC framework built upon TensorFlow. It allows developers to securely train and evaluate machine learning models, maintaining data privacy throughout the process. With significant support from organizations such as Alibaba, `TF-Encrypted` has also been widely adopted across diverse application domains. Therefore, it is crucial to ensure the security of `TF-Encrypted`.
- `MP-SPDZ` is a versatile, general-purpose MPC framework supporting multiple protocols and adversarial models. It provides extensive capabilities for secure computation with arithmetic and binary circuits, facilitating broad usage across diverse application domains. Due to its broad applicability, it is crucial to ensure the security of `MP-SPDZ`.

5.2. Detected Vulnerabilities

As is shown in Table 1, we use `MPCGuard` to test 29 commonly used MPC implementations in `Crypten`, `TF-Encrypted`, and `MP-SPDZ`. Among these 29 implementations, 12 are detected to contain data leakage vulnerabilities. Until the moment this paper is written, all vulnerabilities have been confirmed, and two have been assigned with CVE-ids.

TABLE 1: Vulnerability detection results in 29 MPC implementations in three main-stream MPC frameworks, `Crypten`, `TF-Encrypted`, and `MP-SPDZ`. ✗ means a vulnerability is detected, while - means no vulnerability is detected, and '/' means the protocol is not implemented. 12 out of 29 implementations are detected to contain data leakage vulnerabilities.

Framework	ASS				
	Mul	*Linear*	*LTZ*	*EQZ*	*Truncpr*
`Crypten`	✗	-	✗	✗	✗
`TF-Encrypted`	-	-	-	/	✗
`MP-SPDZ`	-	-	-	-	✗
Framework	**RSS**				
	Mul	*Linear*	*LTZ*	*EQZ*	*Truncpr*
`Crypten`	✗	-	✗	✗	-
`TF-Encrypted`	-	-	✗	✗	✗
`MP-SPDZ`	-	-	-	-	-

Below, we present a detailed analysis of several representative vulnerabilities identified by `MPCGuard`, including their identification and location process, potential impact, and corresponding repair methods. The remaining vulnerabilities are detailed in Appendix B.

***Mul*-RSS-`Crypten`:** When testing the RSS-based *Mul* protocol implementation in `Crypten`, `MPCGuard` identified a significant accuracy gap ($gap > 0.45$) between real-world and ideal-world classifiers, indicating a vulnerability. Utilizing the vulnerability location algorithm of `MPCGuard`, we obtained a call stack containing the function `replicate_shares`. Further examination revealed that this function reshared multiplication results, which are not masked by zero-sum randomness. This omission may allow an adversary to reconstruct the raw data of honest parties, as discussed in Section 2.3. The vulnerability was repaired by introducing zero-sum random values to mask the multiplication results.

***Mul*-ASS-`Crypten`:** When testing the ASS-based *Mul* protocol implementation in `Crypten`, `MPCGuard` identified a significant accuracy gap ($gap > 0.47$) between real-world and ideal-world classifiers, indicating a vulnerability. Utilizing the vulnerability location algorithm of `MPCGuard`, we obtained a call stack containing the function `__beaver_protocol`, which employs Beaver triples [22] for secure multiplication. Further examination showed that the Beaver triples were incorrectly generated solely by the first party, violating the security assumptions of two-party multiplication protocols. Typically, Beaver triples must be generated either by a trusted third party or via secure generation protocols based on homomorphic encryption or oblivious transfer. This vulnerability was repaired by assigning a trusted third party to generate Beaver triples.

***LTZ*-RSS-`TF-Encrypted`:** When testing the RSS-based *LTZ* protocol implementation in `TF-Encrypted`, `MPCGuard` identified a significant accuracy gap ($gap > 0.23$) between real-world and ideal-world classifiers. Utilizing the vulnerability location algorithm of `MPCGuard`, we obtained a call stack containing the function `_and_private_private`. Further examination revealed that randomness generation in the function consistently produced fixed values rather than random values. After discussions with `TF-Encrypted`'s developers, it was determined that this vulnerability resulted from an incorrect variable type declaration in the randomness generation function. As illustrated in Figure 4, the variable `range` was mistakenly declared as `auto` instead of an unsigned type (`uT`). Under specific parameter configurations (e.g., $low = -128$, $high = 126$ used in the function `_and_private_private`), the variable `range` declared as `auto` incorrectly defaults to a signed type. This subtle issue leads to the function consistently outputting fixed values, rather than the intended secure random values. As a result, some bits of honest parties' input are leaked. This kind of vulnerability is particularly challenging to detect without tools like `MPCGuard`, as it occurs only in specific edge-case scenarios. This vulnerability was repaired by changing the type of `range` from `auto` to `uT`.

***Truncpr*-ASS-`MP-SPDZ`:** Different from the other vulnerabilities, the vulnerability in the ASS-based *Truncpr* protocol implementation in `MP-SPDZ` is raised by wrong design rather than wrong implementation. When testing the ASS-based *Truncpr* protocol implementation in `MP-SPDZ`, `MPCGuard` identified a significant accuracy gap ($gap > 0.45$) between real-world and ideal-world classifiers, indicating a vulnerability. However, at this point, `MPCGuard`'s vulnerability location algorithm failed to produce a meaningful call stack, as `MP-SPDZ` employs an additional proxy thread to receive messages. As a result, the call stack only contains code related to the proxy thread, rather than the protocol implementation. We then manually inspected the source code and discovered that the protocol implementation adheres to the original design presented in `SecureML` [23], which has been proven to leak information about parties' output in specific cases [24]. Specifically, as shown in Functionality 1, the *Truncpr* protocol is used to probabilistically truncate the least significant f bits from a secret-shared value x. The expected behavior is that no party should be able to infer information about the parity of another party's output. However, as shown in Protocol 2, the design of this protocol violates this security property in specific cases. If party P_1 already knows the original input value x, it can compute the other party's share $[\![x]\!]_2$ and subsequently deduce the corresponding output share $[\![z]\!]_2$. Given that this protocol is widely used due to its efficiency and is only insecure under specific cases, this vulnerability was repaired by adding explicit warnings to inform users of the potential security implications when employing this protocol.

Functionality 1: $\mathcal{F}_{truncp}$

Inputs:
- The bit length f for truncation.
- $\mathcal{F}_{truncp}$ receives $[\![x]\!]_1$ from P_1, and receives $[\![x]\!]_2$ from P_2, where $([\![x]\!]_1 + [\![x]\!]_2) \mod p = x$. Here, p is the size of the computation domain.

Outputs:
- $\mathcal{F}_{truncp}$ sends $[\![z]\!]_1$ to P_1, and sends $[\![z]\!]_2$ to P_2, where $([\![z]\!]_1 + [\![z]\!]_2) \mod p = z$.

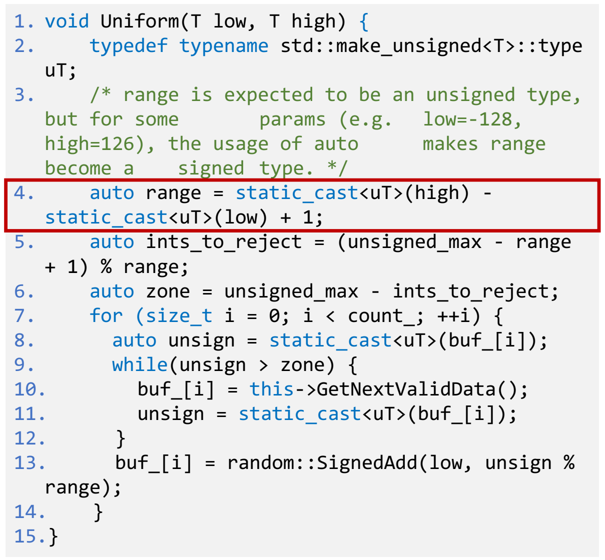
Computation:
1. $\mathcal{F}_{truncp}$ computes $x = ([\![x]\!]_1 + [\![x]\!]_2) \mod p$.
2. $\mathcal{F}_{truncp}$ computes $x' = x \gg f$.
3. $\mathcal{F}_{truncp}$ computes $x'' = x - (x' \ll f)$.
4. $\mathcal{F}_{truncp}$ randomly samples z s.t.
   - With probability $x''/2^f$, $z = x' + 1$.
   - With probability $1 - x''/2^f$, $z = x'$.
5. $\mathcal{F}_{truncp}$ randomly generate $[\![z]\!]_1$ and $[\![z]\!]_2$, such that $([\![z]\!]_1 + [\![z]\!]_2) \mod p = z$.

Protocol 2: $TruncPr$

Inputs:
- The bit length f for truncation
- P_1 inputs $[\![x]\!]_1$, and P_2 inputs $[\![x]\!]_2$, where $([\![x]\!]_1 + [\![x]\!]_2) \mod p = x$. Here, p is the size of the computation domain

Outputs:
- P_1 obtains $[\![z]\!]_1$, and P_2 obtains $[\![z]\!]_2$, where $([\![z]\!]_1 + [\![z]\!]_2) \mod p = z$.

Computation:
1. P_1 computes $[\![z]\!]_1 = [\![x]\!]_1 \gg f$.
2. P_2 computes $[\![z]\!]_2 = -(-[\![x]\!]_2 \gg f)$.



```
void Uniform(T low, T high) {
    typedef typename std::make_unsigned<T>::type uT;
    /* range is expected to be an unsigned type, but for some       params (e.g.   low=-128, high=126), the usage of auto      makes range become a    signed type. */
    auto range = static_cast<uT>(high) - static_cast<uT>(low) + 1;
    auto ints_to_reject = (unsigned_max - range + 1) % range;
    auto zone = unsigned_max - ints_to_reject;
    for (size_t i = 0; i < count_; ++i) {
      auto unsign = static_cast<uT>(buf_[i]);
      while(unsign > zone) {
        buf_[i] = this->GetNextValidData();
        unsign = static_cast<uT>(buf_[i]);
      }
      buf_[i] = random::SignedAdd(low, unsign % range);
    }
}
```

Figure 4: The wrong implementation of the uniform randomness generation function in `TF-Encrypted`. This wrong implementation causes data leakage in *Mul* protocol based on RSS. A correct implementation requires changing the type ‘auto’ to ‘uT’.

5.3. Comparison with Other Models

To the best of our knowledge, `MPCGuard` is the first testing framework designed to detect data leakage vulnerabilities in MPC implementations, and thus no baseline for comparison is available. To further evaluate the effectiveness of `MPCGuard`, we compare our designed neural network classifier (referred to as MPC-NN), which is designed according to the characteristics of MPC protocols, with four commonly used machine learning models: deep neural network (DNN for short), convolutional neural network (CNN for short), recurrent neural network (RNN for short), and XGBoost. To ensure a fair comparison, all the models are tested with the same secret pairs, and the DNN, CNN, and RNN models are configured with a similar number of layers and parameters with MPC-NN.

Below is the detail of each model:

- **DNN:** This model employs a simple feedforward neural network with three fully connected hidden layers, each containing 256 neurons. ReLU is used as the activation function in each hidden layer to introduce non-linearity, while a sigmoid function is applied to the output layer to produce a binary classification prediction.
- **CNN:** This model includes two 1D convolutional layers with 6 and 12 channels to capture sequential dependencies. Each convolution is followed by ReLU activation and max-pooling with a kernel size of 2 to reduce dimensionality and control overfitting. The final pooled output is flattened and passed through two fully connected layers: the first contains 256 neurons with a ReLU as the activation function, and the second contains a neuron with a sigmoid activation to produce a binary classification prediction.
- **RNN:** This model employs an LSTM layer with two recurrent layers, each containing 256 hidden neurons. The input sequence is divided into steps of size 64, and padding is added when necessary to ensure compatibility. The final hidden state from the LSTM is passed through three hidden layers with 256 neurons and a ReLU activation, and an output with a neuron and a sigmoid activation.
- **XGBoost:** The model is set to use 100 trees with a learning rate of 0.01. Each tree has a maximum depth of 6 to prevent overfitting. To further control overfitting, we apply an 80% subsample rate and an 80% column sampling rate. The objective function is set for binary logistic regression, producing probability estimates suitable for binary classification.

TABLE 2: Number of detected vulnerabilities and longest training time of the five models (MPC-NN, DNN, CNN, RNN, and XGBoost) when testing 29 MPC implementations in `Crypten`, `TF-Encrypted`, and `MP-SPDZ`.

Model	#Detected Vulnerability			Training Time
	`Crypten`	`TF-Encrypted`	`MP-SPDZ`	
MPC-NN	7	4	1	7.53s
DNN	0	0	0	6.27s
CNN	0	1	0	23.01s
RNN	0	0	0	41.34s
XGBoost	6	3	1	4.46s

As is shown in Table 2, we can conclude the results as follows:

- Among the tested models, our proposed MPC-NN detects the most data leakage vulnerabilities. In contrast, generic neural networks (NN) fail to detect data leakage vulnerabilities, with only the CNN identifying a single vulnerability. The reason why MPC-NN significantly outperforms other NNs should be attributed to its MPC-specific structure, which constrains the parameter search space and mitigates overfitting to training data. Specifically, identifying data leakage vulnerabilities requires learning complex relationships between the adversary’s view and honest parties’ data. While NNs are well-suited for this task, generic NNs struggle due to their excessively large parameter search space, which makes them prone to overfitting the training data. In other words, the excessively large parameter search space makes them prone to capturing spurious relationships rather than the real relations between the adversary’s view and honest parties’ data. By contrast, MPC-NN leverages domain-specific constraints to narrow down parameter search space, which enhances generalization. Thus, MPC-NN is able to capture the real relations between the adversary’s view and the honest parties’ data.
- Aside from MPC-NN, XGBoost is the next most effective model for vulnerability identification. It identifies 10 vulnerabilities, missing two (RSS-based *Mul* in `Crypten` and RSS-based *Truncpr* in

TF-Encypted). This exception occurs because identifying these two vulnerabilities requires learning a more complex leakage pattern that involves module operations, which poses a challenge for XGBoost. Additionally, XGBoost is the fastest model because, unlike other neural network models, it does not require an extensive iterative training process for convergence. However, despite this speed advantage, identifying as many vulnerabilities as possible is the most critical objective in this task. In contrast, MPC-NN, although slightly slower, achieves comprehensive vulnerability identification and can be trained within an acceptable time. Therefore, we still consider MPC-NN the most suitable model among the five models for this task.

Additionally, although MPC-NN shows the strongest performance among the five models for detecting data leakage vulnerabilities, it does not imply that MPC-NN is the optimal model in all contexts. Designing a more effective detection approach for data leakage vulnerability remains an open question.

5.4. Efficiency Evaluation

To evaluate the efficiency of MPCGuard and identify its performance bottlenecks in detecting data leakage vulnerabilities, we divide the detection process of MPCGuard into four phases: protocol execution (repeatedly execute the MPC implementation and its functionality to collect the adversary's view), vulnerability identification, vulnerability location, and other processes (such as program startup, file read/write operations, etc.). We then record the runtime for each phase, along with the total runtime, for each test.

As is shown in Table 3, we can conclude the results as follows.

- The total runtime ranges from approximately 96 seconds to 8920 seconds, which should be acceptable given the importance of improving security for these implementations. Besides, the total runtime varies significantly across different frameworks due to differences in implementations, including the choice of cryptographic libraries, programming languages, and so on. For example, MP-SPDZ employs oblivious transfer to generate Dabits [25] for implementing the *LTZ* protocol based on ASS, which incurs substantial computational cost. In contrast, *Crypten* achieves a more efficient *LTZ* protocol implementation by using an adder circuit, which reduces the overall runtime significantly.
- Protocol execution is the most time-intensive phase, often accounting for over 90% of the total runtime. This underscores the need for optimized execution in different frameworks to improve efficiency in detecting vulnerabilities.
- Vulnerability identification and location, which are the main processes of our proposed approach, occupy only a small portion of the total runtime. This result indicates the efficiency of our approach.

6. Discussion

Limitations of Our Proposed MPC-NN. Although our proposed MPC-NN performs effectively in our evaluations, there are two categories of vulnerabilities that remain challenging for it to detect.

The first category involves data leakage in complex protocols. Compared to simple protocols, complex protocols typically produce a larger adversary's view during execution, which increases the difficulty of identifying potential data leakage vulnerability. To simulate varying levels of protocol complexity, we pad the adversary's views collected from the 12 vulnerable MPC implementations with random values with different padding rates (defined as the ratio of the padded view length to the original view length). We then evaluate MPC-NN on these padded views to assess whether it can still detect the vulnerabilities. As is shown in Figure 5, the number of identified vulnerabilities decreases as the adversary view length increases. Notably, when the padding rate reaches 10, MPC-NN is able to detect only 7 out of the original 12 vulnerabilities, highlighting its limitations in handling complex protocols.

The second category involves the usage of insecure pseudo-random number generators (PRNGs) that fail to meet cryptographic security standards. For instance, common PRNG implementations such as std::mt19937 in C++ and random in Python fail to meet cryptographic security standards. Consequently, MPC implementations relying on these insecure PRNGs inherently contain vulnerabilities. However, since the outputs of these insecure PRNGs may appear statistically random, MPC-NN struggles to differentiate them from genuinely secure randomness. Thus, vulnerabilities resulting from insecure PRNGs remain a significant detection challenge and represent an important direction for future improvements of MPC-NN.

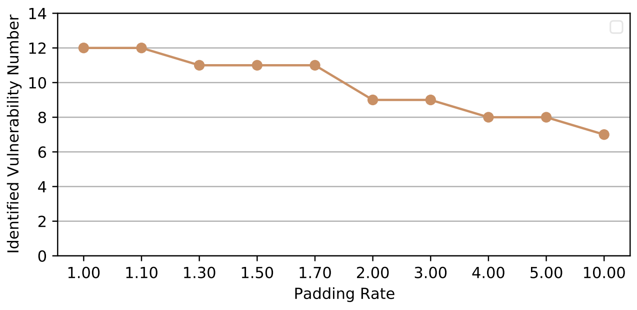


Figure 5: Number of identified vulnerabilities under different padding rates (ratio of padded adversary's view length to original adversary's view length). Experimental parameters: $thr = 0.1$, $t = 2000$, and $n = 800$.

Other Promising Detection Approaches. While this paper adopts neural network classifiers to detect data leakage in MPC implementations, several promising approaches warrant further exploration. First, traditional machine learning methods—such as random forests, symbolic regression, and XGBoost—can be further explored as they can be trained

TABLE 3: Runtime (seconds) of `MPCGuard`, which consists of four phases 'Protocol Execution', 'Vulnerability Identification', 'Vulnerability Location', and 'Other Processes'. The time for 'Vulnerability Location' is zero for some protocols because no vulnerabilities are identified, and thus, there is no need to locate vulnerabilities. '/' means the protocol is not implemented.

Runtime	Crypten									
	ASS					RSS				
	Mul	*Linear*	*LTZ*	*EQZ*	*Truncpr*	*Mul*	*Linear*	*LTZ*	*EQZ*	*Truncpr*
Total	2587.81	6210.38	1384.42	2679.55	1237.86	1508.89	7446.35	1670.82	1728.63	7432.95
Protocol Execution	2490.45	6036.37	1285.71	2552.77	1193.57	1454.39	7269.85	1564.07	1592.40	7256.69
Vulnerability Identification	58.43	172.64	17.74	58.55	25.93	16.35	175.24	18.52	20.60	175.07
Vulnerability Location	38.35	0	80.56	67.61	18.14	37.87	0	87.64	114.59	0
Other Processes	0.59	1.37	0.41	0.62	0.22	0.29	1.26	0.59	1.049	1.194
Runtime	TF-Encrypted									
	ASS					RSS				
	Mul	*Linear*	*LTZ*	*EQZ*	*Truncpr*	*Mul*	*Linear*	*LTZ*	*EQZ*	*Truncpr*
Total	1010.29	630.56	2164.29	/	96.69	1729.76	1032.60	2450.79	1361.60	250.74
Protocol Execution	828.03	450.84	1966.05	/	53.62	1545.66	854.23	2338.97	1263.70	196.29
Vulnerability Identification	180.99	178.46	195.41	/	28.03	182.84	177.01	52.154	21.64	25.69
Vulnerability Location	0	0	0	/	14.81	0	0	59.18	75.93	28.54
Other Processes	1.27	1.26	2.83	/	0.23	1.25	1.36	0.49	0.33	0.22
Runtime	MP-SPDZ									
	ASS					RSS				
	Mul	*Linear*	*LTZ*	*EQZ*	*Truncpr*	*Mul*	*Linear*	*LTZ*	*EQZ*	*Truncpr*
Total	1667.49	1544.08	8920.81	5429.04	298.11	2109.92	2115.03	2757.22	2588.72	2112.15
Protocol Execution	1483.42	1363.10	8729.96	5241.21	271.60	1920.98	1934.28	2530.49	2373.51	1936.82
Vulnerability Identification	182.83	179.71	187.93	185.94	25.80	187.78	179.41	221.66	211.37	174.05
Vulnerability Location	0	0	0	0	0.41	0	0	0	0	0
Other Processes	1.24	1.26	2.92	1.88	0.30	1.15	1.34	5.07	3.84	1.28

much more efficiently. As is shown in Section 5.3, XGBoost outperforms generic neural networks and can be trained much more efficiently, making it a compelling choice for large-scale or resource-constrained testing scenarios. Another promising direction is specification-based testing [26], which verifies whether an implementation faithfully adheres to its specifications. With specification-based testing, we can directly check whether an MPC implementation follows its design without executing the MPC implementation, which is usually inefficient. In addition, techniques such as taint analysis [27] can be employed to trace insecure data flows between parties, offering a complementary perspective on finding data leakage.

Simulation for Malicious Adversaries. While this paper primarily focuses on simulating a semi-honest adversary to detect data leakage vulnerabilities in MPC implementations, we also extend `MPCGuard` to simulate a malicious adversary to detect memory vulnerabilities in MPC implementations. `MPCGuard` simulates a malicious adversary by sending wrong messages to honest parties, and monitors whether the MPC implementations crash to identify memory vulnerabilities. Using `MPCGuard` with malicious adversary simulation, we further detect seven memory vulnerabilities in `MP-SPDZ` and one memory vulnerability in `emp-ot`, which is a widely used library for oblivious transfer. These vulnerabilities have been reported to the developers and assigned five CVE-ids.

Suggestions for MPC Implementations. To mitigate the risks of data leakage and memory vulnerabilities, we provide the following suggestions for developers: (1) Utilize well-established cryptographic and randomness modules rather than implementing custom underlying libraries. Self-implemented cryptographic primitives often introduce subtle vulnerabilities and are also generally less efficient. (2) Implement MPC protocols faithfully and completely according to their original specifications. Avoid omitting seemingly insignificant protocol steps, even if they appear to have no immediate impact on correctness, because these omissions usually introduce vulnerabilities. (3) Regularly write and execute unit tests for your protocol implementations, especially the unit tests that consider the edge cases. (4) Employ existing sanitizers (e.g. AddressSanitizer), particularly when developing in C++. Unit tests combined with sanitizers help identify and resolve vulnerabilities (especially memory vulnerabilities) at the early stages of development.

Functionality Implementation. While it currently requires manually implementing the corresponding functionality when using `MPCGuard` to detect data leakage vulnerabilities in MPC implementations, the process is typically straightforward. The functionalities are often described with pseudocode in the related paper and can usually be implemented in fewer than ten lines of Python code without requiring expertise in MPC protocols.

Future Work. In future work, we aim to enhance `MPCGuard` by incorporating automated functionality generation using large language models, such as ChatGPT. Besides, we will explore how to detect data leakage vulnerabilities in MPC implementations by simulating a malicious adversary. Furthermore, we will focus on designing a more effective model for detecting data leakage vulnerabilities.

7. Related Work

Formal Verification of MPC Security. Some works [4], [5], [6], [7] focus on automated formal verification of MPC protocols. Haagh et al. [4] use EasyCrypt to prove the security of the GMW protocol [7]. Barthe et al. [6] develop a probabilistic reasoning framework to verify additive secret sharing. Gancher et al. [5] introduce a language system that uses protocol simulators to verify security properties, particularly for the GMW protocol.

However, as is shown in our testing results (Section 5.2), MPC implementations often contain data leakage vulnerabilities due to the complexity of MPC protocols, even if the underlying MPC protocols are secure, Therefore, developing a practical framework for detecting vulnerabilities in MPC implementations is necessary, even though many automated formal verification tools for MPC protocols exist.

Other Software Testing for MPC Implementations. `MPCDIFF` [28] employs differential testing to identify inputs that cause deviations between MPC-hardened machine learning models and the plaintext models, and further locates error-prone computation units. `HEDIFF` [29] similarly uses differential testing to uncover inputs that cause deviations between fully homomorphic encryption-based models and the plaintext models, and further analyzes these discrepant inputs to determine whether they exhibit transferable noise patterns. `MT-MPC` [30] uses metamorphic testing to test MPC compilers, helping detect bugs in the conversion from high-level languages to various intermediate representations (e.g., arithmetic or Boolean circuits). `Mtzk` [31] similarly uses metamorphic testing to test zero-knowledge compilers, helping detect bugs in conversion from domain-specific languages into zero-knowledge circuits.

While these works also test MPC implementations, their targets differ from those addressed by `MPCGuard`. To the best of our knowledge, `MPCGuard` is the first framework designed to detect data leakage vulnerabilities in MPC implementations. These vulnerabilities are particularly serious because they enable an adversary to infer the private data of honest parties, critically undermining the intended privacy protection of MPC protocols.

Neural Network Aided Cryptanalysis. Several works [18], [32], [33], [34] also leverage neural networks to aid cryptanalysis. Aron Gohr [32] proposes the first deep learning-based key recovery attack on 11-round Speck32/64, pioneering the direction of neural-aided cryptanalysis. Building upon this foundation, Chen et al. [33] propose `NASA`, a method that supports theoretical complexity estimation and does not rely on neutral bits. `NASA` is proved to be more effective than Gohr's attack when neutral bits are insufficient. `Salsa` [34] uses transformers to perform statistical cryptanalysis on lattice cryptography. `NeuralD` [18] employs a neural distinguisher to form a probabilistic testing oracle to determine if a pair of Oblivious RAM (ORAM for short) inputs violates the obliviousness guarantee.

Among these works, `NeuralD` is the most similar to our framework `MPCGuard` and has inspired its design. However, adapting `NeuralD`'s approach to `MPCGuard` faces significant technical challenges. Specifically, it requires a complete redesign of both the leakage identifier and the neural network classifier structure according to the characteristics of MPC protocols to ensure detection accuracy.

Testing Other Privacy-Enhancing Protocols. Several works [17], [18], [35], [36], [37] focus on testing other privacy-enhancing protocols, such as Differential Privacy (DP for short) [38] and ORAM [39]. Ding et al. [17] use differential testing to detect violations of DP. `DPCheck` [35] adapts the well-known 'pointwise' technique from informal proofs to distinguish correct from buggy implementations of PrivTree. `DP-Sniper` [36] automatically finds violations of DP by training a classifier and transforming it into an approximately optimal attack on DP. Fujita et al. [37] analyze the effect of pseudorandom number generators on the randomness of the derived path sequence in Path ORAM by evaluating various PRNGs across a wide range of randomness.

These works all focus on testing and verifying privacy protection capabilities in various protocols. However, they focus on specific domains, such as DP and ORAM, and their techniques are tailored to the characteristics of those protocols. In contrast, `MPCGuard` provides specialized testing techniques tailored to MPC implementations, enabling the detection of data leakage vulnerabilities, which can critically undermine the intended privacy protection of MPC protocols.

8. Conclusion

In this paper, we propose `MPCGuard`, a practical framework for detecting data leakage vulnerabilities in MPC implementations. We first establish a leakage identifier with two neural network classifiers to identify whether an MPC implementation contains data leakage vulnerabilities. To enhance detection effectiveness, we further design the structures of neural network classifiers according to the characteristics of MPC protocols. After identifying a data leakage vulnerability, We employ a delta method to assist in locating the vulnerability. To demonstrate the effectiveness of `MPCGuard`, we apply `MPCGuard` to test 29 commonly-used MPC implementations in three mainstream MPC frameworks, `Crypten`, `TF-Encrypted`, and `MP-SPDZ`, discovering 12 out of 29 implementations contain data leakage vulnerabilities, some of which can lead to the reconstruction of raw data. Until the moment this paper is written, all vulnerabilities have been confirmed and two have been assigned with CVE-IDs.

Acknowledgments

This paper is supported by Natural Science Foundation of China (92370120, 62172100), National Cryptologic Science Fund of China (2025NCSF01010), the National Key R&D Program of China (2023YFC3304400), and Ant Group Research Fund. We thank all anonymous reviewers for their insightful comments. Weili Han is the corresponding author.

References

[1] P. Voigt and A. Von dem Bussche, "The eu general data protection regulation (gdpr)," A Practical Guide, 1st Ed., Cham: Springer International Publishing, 2017.

[2] T. T. T. on Privacy-Enhancing Technologies of the United Nations Committee of Experts on Big Data and D. S. for Official Statistics, "Un guide on privacy-enhancing technologies for official statistics," https://unstats.un.org/bigdata/task-teams/privacy/guide/, 2023.

[3] 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.

[4] H. Haagh, A. Karbyshev, S. Oechsner, B. Spitters, and P.-Y. Strub, "Computer-aided proofs for multiparty computation with active security," in Proceedings of the 2018 IEEE 31st Computer Security Foundations Symposium (CSF). IEEE, 2018, pp. 119–131.

[5] J. Gancher, K. Sojakova, X. Fan, E. Shi, and G. Morrisett, "A core calculus for equational proofs of cryptographic protocols," in Proceedings of the ACM on Programming Languages. ACM New York, NY, USA, 2023, pp. 866–892.

[6] G. Barthe, J. Hsu, and K. Liao, "A probabilistic separation logic," in Proceedings of the ACM on Programming Languages. ACM New York, NY, USA, 2019, pp. 1–30.

[7] 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.

[8] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "{AddressSanitizer}: A fast address sanity checker," in 2012 USENIX annual technical conference (USENIX ATC 12), 2012, pp. 309–318.

[9] E. Stepanov and K. Serebryany, "Memorysanitizer: fast detector of uninitialized memory use in c++," in 2015 IEEE/ACM International Symposium on Code Generation and Optimization (CGO). IEEE, 2015, pp. 46–55.

[10] T.-E. Contributors, "TF-Encrypted: A framework for privacy-preserving machine learning," 2024, accessed: 2024-11-11. [Online]. Available: https://github.com/tf-encrypted/tf-encrypted

[11] M. Keller, "MP-SPDZ: A versatile framework for multi-party computation," in Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security, 2020. [Online]. Available: https://doi.org/10.1145/3372297.3417872

[12] A. Shamir, "How to share a secret," Communications of the ACM, vol. 22, no. 11, pp. 612–613, 1979.

[13] R. A. DeMillo, R. J. Lipton, and A. J. Perlis, "Social processes and proofs of theorems and programs," Commun. ACM, vol. 22, no. 5, pp. 271–280, 1979. [Online]. Available: https://doi.org/10.1145/359104.359106

[14] R. Bommasani, D. A. Hudson, E. Adeli, R. Altman, S. Arora, S. von Arx, M. S. Bernstein, J. Bohg, A. Bosselut, E. Brunskill et al., "On the opportunities and risks of foundation models," arXiv preprint arXiv:2108.07258, 2021.

[15] M. Zalewski, "American Fuzzy Lop (AFL)," 2014, accessed: 2024-07-10. [Online]. Available: http://lcamtuf.coredump.cx/afl/

[16] 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.

[17] 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.

[18] 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.

[19] A. Trask, F. Hill, S. E. Reed, J. Rae, C. Dyer, and P. Blunsom, "Neural arithmetic logic units," Advances in neural information processing systems, vol. 31, 2018.

[20] G. Parascandolo, H. Huttunen, and T. Virtanen, "Taming the waves: sine as activation function in deep neural networks," 2016.

[21] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library," in Proceedings of the 33rd International Conference on Neural Information Processing Systems, 2019, pp. 8026–8037.

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

[23] P. Mohassel and Y. Zhang, "Secureml: A system for scalable privacy-preserving machine learning," in Proceedings of the 2017 IEEE symposium on security and privacy (SP). IEEE, 2017, pp. 19–38.

[24] Y. Li, Y. Duan, Z. Huang, C. Hong, C. Zhang, and Y. Song, "Efficient {3PC} for binary circuits with application to {Maliciously-Secure}{DNN} inference," in Proceedings of the 32nd USENIX Security Symposium (USENIX Security 23), 2023, pp. 5377–5394.

[25] 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 Proceedings of 2019 IEEE Symposium on Security and Privacy, 2019, pp. 1102–1120.

[26] R. M. Hierons, K. Bogdanov, J. P. Bowen, R. Cleaveland, J. Derrick, J. Dick, M. Gheorghe, M. Harman, K. Kapoor, P. Krause et al., "Using formal specifications to support testing," ACM Computing Surveys (CSUR), vol. 41, no. 2, pp. 1–76, 2009.

[27] E. J. Schwartz, T. Avgerinos, and D. Brumley, "All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask)," in 2010 IEEE symposium on Security and privacy. IEEE, 2010, pp. 317–331.

[28] Q. Pang, Y. Yuan, and S. Wang, "Mpcdiff: Testing and repairing mpc-hardened deep learning models," in Proceedings of the 31st Annual Network and Distributed System Security Symposium, NDSS 2024, San Diego, California, USA, February 26 - March 1, 2024. The Internet Society, 2024. [Online]. Available: https://www.ndss-symposium.org/ndss-paper/mpcdiff-testing-and-repairing-mpc-hardened-deep-learning-models/

[29] Y. Peng, D. Wu, Z. Liu, D. Xiao, Z. Ji, J. Rahmel, and S. Wang, "Testing and understanding deviation behaviors in fhe-hardened machine learning models," in 2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE). IEEE Computer Society, 2025, pp. 644–644.

[30] Y. Li, D. Xiao, Z. Liu, Q. Pang, and S. Wang, "Metamorphic testing of secure multi-party computation (mpc) compilers," vol. 1, no. FSE, 2024. [Online]. Available: https://doi.org/10.1145/3643781

[31] D. Xiao, Z. Liu, Y. Peng, and S. Wang, "Mtzk: Testing and exploring bugs in zero-knowledge (zk) compilers," in NDSS, 2025.

[32] A. Gohr, "Improving attacks on round-reduced speck32/64 using deep learning," in Advances in Cryptology–CRYPTO 2019: 39th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 18–22, 2019, Proceedings, Part II 39. Springer, 2019, pp. 150–179.

[33] Y. Chen, Y. Shen, and H. Yu, "Neural-aided statistical attack for cryptanalysis," The Computer Journal, p. 2480–2498, Oct 2023. [Online]. Available: http://dx.doi.org/10.1093/comjnl/bxac099

[34] E. Wenger, M. Chen, F. Charton, and K. E. Lauter, “Salsa: Attacking lattice cryptography with transformers,” Advances in Neural Information Processing Systems, vol. 35, pp. 34 981–34 994, 2022.

[35] H. Zhang, E. Roth, A. Haeberlen, B. C. Pierce, and A. Roth, “Testing differential privacy with dual interpreters,” Proceedings of the ACM on Programming Languages, vol. 4, no. OOPSLA, p. 1–26, Nov 2020. [Online]. Available: http://dx.doi.org/10.1145/3428233

[36] B. Bichsel, S. Steffen, I. Bogunovic, and M. Vechev, “Dp-sniper: Black-box discovery of differential privacy violations using classifiers,” in Proceedings of the 2021 IEEE Symposium on Security and Privacy, May 2021.

[37] H. Fujita, N. Fujieda, and S. Ichikawa, “An analysis on randomness of path oram for light-weight implementation,” in Proceedings of the 2018 Sixth International Symposium on Computing and Networking Workshops (CANDARW). IEEE, 2018, pp. 163–165.

[38] C. Dwork, “Differential privacy,” in Proceedings of the International colloquium on automata, languages, and programming. Springer, 2006, pp. 1–12.

[39] O. Goldreich and R. Ostrovsky, “Software protection and simulation on oblivious rams,” Journal of the ACM (JACM), vol. 43, no. 3, pp. 431–473, 1996.

[40] 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.

Appendix A. Leakage Identifier for Output

As is shown in Algorithm 3, the algorithm for identifying leakage to the outputs of the honest parties takes as input an MPC implementation $\mathcal{I}$ and its corresponding functionality $\mathcal{F}$. It returns ‘True’ if a data leakage vulnerability to the output of an honest party is identified, and ‘False’ otherwise.

Since most steps in this algorithm are similar to Algorithm 1. Hence, below, we focus on the difference of this algorithm to Algorithm 1.

Data Collection: In order to save time for executing the MPC implementation, during identifying the leakage to output, we directly use the secrets, the views of the adversary, and the output of the honest party produced during identifying the leakage to input.

Dataset Split: In this step, the difference between this algorithm and Algorithm 1 lies in the labeling method. In this algorithm, the label of each sample is ‘0’ if the output of the honest party is odd and ‘1’ if it is even. Note that directly using the output of the honest party as the label for each sample makes it difficult for the neural network to converge, because the outputs of MPC protocols are often random values within the computation domain of the MPC protocols and span a wide range of values. Hence, `MPCGuard` uses the parity of the output as the label, converting it into a binary classification task, which helps the neural network converge more efficiently. Additionally, since parity is determined by the least significant bit, it is more sensitive to data changes and easier to identify if leakage occurs.

Algorithm 3 Leakage Identifier for Output

```
Input: An MPC implementation $\mathcal{I}$, and its corresponding functionality $\mathcal{F}$.
Output: True if a data leakage vulnerability is identified, and False otherwise.
 1: Read the secrets, views of the adversary, and output of the honest parties produced in Algorithm 1.
 2: for each secret do
 3:     Split the corresponding adversary's views in the real world into a training dataset $\mathcal{D}_{train\_real}$ and a test dataset $\mathcal{D}_{test\_real}$, where each sample is labeled with ‘0’ if the corresponding output of the honest party is odd, and ‘1’ if the output is even.
 4:     Split the corresponding adversary's views in the ideal world into a training dataset $\mathcal{D}_{train\_ideal}$ and a test dataset $\mathcal{D}_{test\_ideal}$, where the sample is labeled with ‘0’ if the corresponding output of the honest party is odd, and ‘1’ if the output is even.
 5:     Train a neural network classifier on $\mathcal{D}_{train\_real}$ and test it on $\mathcal{D}_{test\_real}$ to obtain the accuracy $Acc_{real}$.
 6:     Train a neural network classifier on $\mathcal{D}_{train\_ideal}$ and evaluate it on $\mathcal{D}_{test\_ideal}$ to obtain the accuracy $Acc_{ideal}$.
 7:     if $Acc_{real} > Acc_{ideal} + thr$ then
 8:         return True
 9:     end if
10: end for
11: return False
```

Appendix B. Remaining Vulnerabilities

***LTZ*-ASS-`Crypten`:** When testing the ASS-based *LTZ* protocol implementation in `Crypten`, `MPCGuard` identified a significant accuracy gap ($gap > 0.47$) between real-world and ideal-world classifiers, indicating a vulnerability. Utilizing the vulnerability location algorithm of `MPCGuard`, we obtained a call stack containing the function `beaver.B2A_single_bit`, which employs daBits [25] for secure conversion between additive secret shares and boolean secret shares. Further examination showed that the daBits were incorrectly generated solely by the first party, violating the security assumptions of two-party *LTZ* protocols. Typically, the daBits must be generated either by a trusted third party or via secure generation protocols based on homomorphic encryption or oblivious transfer. This vulnerability was repaired by assigning a trusted third party to generate the daBits.

***EQZ*-ASS-`Crypten`:** When testing the ASS-based *EQZ* protocol implementation in `Crypten`, `MPCGuard` identified a significant accuracy gap ($gap > 0.46$) between real-world and ideal-world classifiers, indicating a vulnerability. With further examination on this vulnerability, we found that this vulnerability shares the same root cause as the ASS-based *LTZ* implementation—relying on the first party to generate daBits for secure conversion between additive and boolean secret shares.

***Truncpr*-ASS-`Crypten`:** When testing the ASS-based *Truncpr* protocol implementation in `Crypten`, `MPCGuard` identified a significant accuracy gap ($gap > 0.47$) between real-world and ideal-world classifiers, indicating a vulnerability. Utilizing the vulnerability location algorithm of `MPCGuard`, we obtained a call stack containing the code to collect the output of the first party, indicating the first party's output directly leaked information about the honest parties' data. We then manually inspected the source code and discovered that the protocol implementation also adheres to the original design described in `SecureML` [23], which has been proven to leak information about parties' output in specific cases [24].

***LTZ*-RSS-`Crypten`:** When testing the RSS-based *LTZ* protocol implementation in `Crypten`, `MPCGuard` identified a significant accuracy gap ($gap > 0.46$) between real-world and ideal-world classifiers, indicating a vulnerability. With further examination, we found that this vulnerability shares the same root cause as the one in the ASS-based *LTZ* protocol implementation in `Crypten`—relying on the first party to generate daBits.

***EQZ*-RSS-`Crypten`:** When testing the RSS-based *EQZ* protocol implementation in `Crypten`, `MPCGuard` identified a significant accuracy gap ($gap > 0.45$) between real-world and ideal-world classifiers, indicating a vulnerability. With further examination, we found that this vulnerability also shares the same root cause as the one in the ASS-based *LTZ* protocol implementation in `Crypten`—relying on the first party to generate daBits.

***Truncpr*-ASS-`TF-Encrypted`:** When testing the ASS-based *Truncpr* protocol implementation in `TF-Encrypted`, `MPCGuard` identified a significant accuracy gap ($gap > 0.45$) between real-world and ideal-world classifiers, indicating a vulnerability. Utilizing the vulnerability location algorithm of `MPCGuard`, we obtained a call stack containing the code to collect the output of the first party, indicating the first party's output directly leaked information about the honest parties' data. We then manually inspected the source code and discovered that the protocol implementation also adheres to the original design described in `SecureML` [23].

***EQZ*-RSS-`TF-Encrypted`:** When testing the RSS-based *EQZ* protocol implementation in `TF-Encrypted`, `MPCGuard` identified a significant accuracy gap ($gap > 0.11$) between real-world and ideal-world classifiers. With further examination, we found that this vulnerability also shares the same root cause as the one in *LTZ* protocol implementation in `TF-Encrypted` — an incorrect variable type declaration in the randomness generation function.

***Truncpr*-RSS-`TF-Encrypted`:** When testing the RSS-based *Truncpr* protocol implementation in `MP-SPDZ`, `MPCGuard` identified a significant accuracy gap ($gap > 0.35$) between real-world and ideal-world classifiers, indicating a vulnerability. Utilizing the vulnerability location algorithm of `MPCGuard`, we obtained a call stack containing the code to collect the inputs of the first party, indicating the first party's input directly leaked information about the honest parties' data. Further examination revealed that the protocol implementation also adheres to the original design described in `ABY3` [40], which has been proven to leak information about parties' output in specific cases [24].

Appendix C. Different Thresholds

In this section, we present the theoretical false positive probabilities (with $n = 800$) along with experimental results for different thresholds.

As is shown in Figure 6, when the threshold is set to 0.1 or higher, the theoretical false positive probability becomes negligible (close to zero). Although increasing the threshold can further reduce the theoretical false positive probability, it may also impair detection capability. For example, at $thr = 0.3$, `MPCguard` reports only 11 vulnerabilities, indicating that one actual vulnerability is missed. Conversely, lowering the threshold may lead to more false positives: at $thr = 0.03$, 13 vulnerabilities are reported, but only 12 are genuine. Therefore, setting $thr = 0.1$ strikes a good balance—achieving a negligible false positive probability while maintaining high detection accuracy.

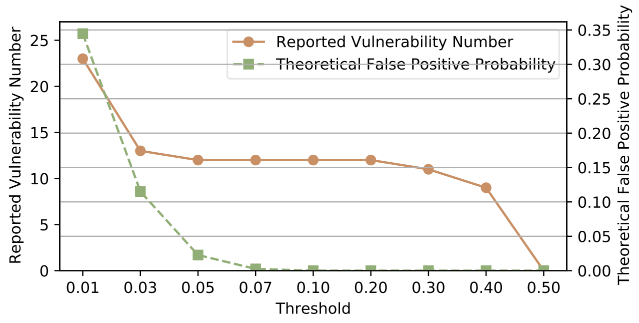


Figure 6: Theoretical false positive probability and reported vulnerability number of different thresholds. Experimental parameters: $t = 2000$, and $n = 800$.

Appendix D. Research Ethics Committee Response

We thank the committee for the detailed review and appreciate the opportunity to clarify these points.

1. Please clarify how many vulnerabilities have been assigned CVE-IDs. In some places the paper says two, one place says five.

Totally, seven vulnerabilities have been assigned CVE-IDs. Among these seven vulnerabilities, two are data leakage vulnerabilities (found in `TF-encrypted` and `Crypten`), which are the PRIMARY contributions of our paper, and are mentioned in the abstract, introduction, evaluation, and conclusion sections. The remaining five are memory vulnerabilities (found in `MP-SPDZ` and `emp-ot`), detected by extending `MPCGuard` to simulate a malicious adversary, and are mentioned only in the discussion section.

2. Specify if the IDs were issued by the developers of the evaluated MPC frameworks or other organizations, e.g., MITRE.

All of the seven CVE-IDs were issued by MITRE.

3. Discuss developer responses and whether they were given sufficient time to address the vulnerabilities.

Totally, we detected 17 vulnerabilities (seven of which were assigned CVE-IDs) across four multi-party computation (MPC) frameworks: four in `TF-Encrypted`, five in `MP-SPDZ`, seven in `Crypten`, and one in `emp-ot`. All of these vulnerabilities have been confirmed and repaired.

Upon detecting the vulnerabilities, we promptly reported them to the developers of `TF-Encrypted`, `MP-SPDZ`, `Crypten`, and `emp-ot` through GitHub issues, Facebook's Whitehat Report website, or email, and received their confirmation on all 17 vulnerabilities. Our reports not only included detailed technical descriptions but also emphasized the security implications of each vulnerability. In particular, the developers of `TF-Encrypted` and `MP-SPDZ` repaired all nine vulnerabilities within one week. Two out of the seven vulnerabilities in `Crypten` were repaired by the development team while the rest five were repaired with the help of our pull requests. The vulnerability in `emp-ot` was also addressed by our pull request.

For all the vulnerabilities, we provided a disclosure window of at least 90 days ahead of publication.

4. Discuss potential impact of the findings on users and data used by MPC frameworks that do not address the vulnerabilities.

For the data leakage vulnerabilities, failure to address them may allow an adversary to infer sensitive information that should remain confidential, thereby compromising user privacy. For the memory vulnerabilities, failure to address them may allow an adversary to launch denial-of-service attacks on the system. Fortunately, all the data leakage and memory vulnerabilities we identified have been repaired.

Additionally, we noticed that another product, `secretflow` (https://github.com/secretflow/secretflow), leverages parts of `emp-ot`'s code. To ensure that the vulnerability in `emp-ot` does not affect `secretflow`, we also reported the vulnerability to the developers of `secretflow`.

Through responsible disclosure of these findings, we can increase awareness of MPC framework security and strengthen the overall security posture of MPC frameworks. Prior to our work, few recognized that many MPC frameworks contain vulnerabilities stemming from implementation errors. Through responsible disclosure of these findings, we can prompt developers to implement remedial measures and build more secure MPC frameworks, and prompt users to be cautious when adopting MPC frameworks that are not sure to be secure.

Appendix E. Meta-Review

The following meta-review was prepared by the program committee for the 2025 IEEE Symposium on Security and Privacy (S&P) as part of the review process as detailed in the call for papers.

E.1. Summary

The paper proposes an automatic detection tool for vulnerabilities in MPC implementations. The tool has identified several, sometimes crucial vulnerabilities in existing implementations. The ethics committees has validated the ethical handling of these vulnerabilities.

E.2. Scientific Contributions

- Creates a New Tool to Enable Future Science
- Identifies an Impactful Vulnerability
- Establishes a New Research Direction

E.3. Reasons for Acceptance

1) This paper opens a new research direction with already many successful, initial results. The expected impact of the work on the security of MPC is also very positively acknowledged. We expect many follow-up works.

E.4. Noteworthy Concerns

1) Solutions other than neural network might be a better choice in detecting these vulnerabilities. While the authors' success speaks for itself, other approaches may be evaluated in the future.
2) The authors have categorized the vulnerabilities. However, the false negatives of the approach remain somewhat unknown. Further investigation, e.g., by deliberately inserting adaptive vulnerabilities that aim to evade detection, is needed by follow-up work.