FaultyGarble: Fault Attack on Secure Multiparty Neural Network Inference

Mohammad Hashemi, Dev Mehta, Kyle Mitard, Shahin Tajik, Fatemeh Ganji
Worcester Polytechnic Institute
{*mhashemi,dmmehta2,krmitard,stajik,fganji*}@*wpi.edu*

***Abstract*—The success of deep learning across a variety of applications, including inference on edge devices, has led to increased concerns about the privacy of users' data and deep learning models. Secure multiparty computation allows parties to remedy this concern, resulting in a growth in the number of such proposals and improvements in their efficiency. The majority of secure inference protocols relying on multiparty computation assume that the client does not deviate from the protocol and *passively* attempts to extract information. Yet clients, driven by different incentives, can act *maliciously* to actively deviate from the protocol and disclose the deep learning model owner's private information. Interestingly, faults are well understood in multiparty computation-related literature, although fault attacks have not been explored. Our paper introduces the very first fault attack against secure inference implementations relying on garbled circuits as a prime example of multiparty computation schemes. In this regard, laser fault injection coupled with a model-extraction attack is successfully mounted against existing solutions that have been assumed to be secure against active attacks. Notably, the number of queries required for the attack is equal to that of the best model-extraction attack mounted against the secure inference engines under the semi-honest scenario.**

***Keywords*-Multiparty Computation, Garbled Circuits, Malicious Adversary, Neural Network Inference, Laser Fault Attack.**

I. INTRODUCTION

Machine learning (ML), and in particular deep neural networks (NNs), is widely adopted to create models that perform image recognition, identify fraudulent transactions, natural language processing, and even drug discovery- to name a few [1], [2]. Similar to other ML tasks, training and inference are two typical phases involved in NN applications. While a large amount of data is used to determine the best parameter values of an NN during the training phase, the already trained NN is applied to a new input in the inference phase. Here we focus on inference tasks at the edge, where the existing solutions either need clients to send potentially sensitive data to servers or the model owner stores their proprietary NN model on clients' devices. The latter is of great importance due to growing needs for edge computing, which provides avenues to reduce network congestion through computing near users as well as reducing communication needed to reach resource-hungry services cf. [3]. This can be achieved, of course, at the cost of possible IP infringement, and consequently, harming the NN owners' business model or revealing information about the training data and model weights [2]. In view of this, it is not surprising that *physical attacks* have been mounted against edge devices embodying NN models [4], [5], [6], [7], [8], [9], [10]. Secure inference is the response to this challenge, where the client and the NN owner interact so that the client obtains the prediction result without disclosing any other information about the client's input or the model weights (*inputs' privacy*) [11].

To realize secure inference, numerous studies have been devoted to *secure multiparty computation* (MPC), especially secure two-party computation [11], [12], [13], [14], [15], [16], [17], [18], [19], [20]. Thanks to their competitive performance in terms of online latency and accuracy, the application of MPC in NN inference would continue to gain momentum. Albeit the fact that MPC-based NN inference can guarantee the security of the user's data and NN model (see Figure 1 for an overview), implementation attacks can compromise that. The question is whether these attacks are beyond the adversary models of MPC protocols.

It is arguably inaccurate if one assumes that such attacks are out of the scope of adversary models in secure two-party computation. When it comes to passive implementation attack, e.g., side-channel analysis, or as called in [21] side-information leakage, has been considered in the literature. Concretely, it is argued that "Privacy is rarely absolute," [21], meaning that some side channels leak during the execution of the protocols. Such side channels include the size of the circuit, its topology, or even the circuit itself cf. [21], [22], [23]; nevertheless, the parties' inputs should not be disclosed, although recent work has posed a serious challenge to this [24], [25]. Specifically, the *passive* side-channel analysis in these works has led to the private information leakage through power consumption of the device or the execution time of the protocol [24], [25].

Regarding susceptibility of schemes to *active implementation attacks*, i.e., fault attacks, MPC-related literature has understood faults as corruption either in the circuit or the inputs cf. [26], [27], [28], [29], [30], [31]. Faults -or as sometimes referred to Byzantine faults- differ from unintentional machine failures and are concerned with the attacks mounted by an external entity or a subset of the participating parties [30]. Fault attacks aim to disclose parties' private inputs or cause the result of the computation to be incorrect, violating privacy and correctness, respectively. To deal with faults, the notions of "robust" and "fair" execution were introduced [26], [32], which have found application in secure NN inference. Fairness ensures that either all parties or

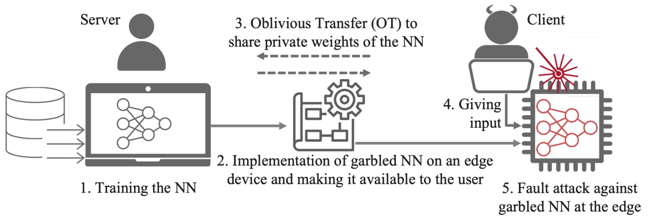
Server
3. Oblivious Transfer (OT) to share private weights of the NN
Client
4. Giving input
1. Training the NN
2. Implementation of garbled NN on an edge device and making it available to the user
5. Fault attack against garbled NN at the edge

Figure 1: Overview of our attack scenario. The client has physical access to the device at the edge running the garbled NN to perform inference. The server represents the NN owner whose private inputs are NN weights.

none learn the output [33], [34]. Besides, robustness, aka guaranteed output delivery, has been another crucial aspect of running a protocol in the face of active attacks. In this context, the malicious parties should not be able to disrupt outputs' delivery to other parties, i.e., carrying out a "denial of service" attack [30]. What can be concluded from this is that *physical security* has been overlooked in the implementation of MPC protocols. The question that naturally follows this discussion is whether *there is any fault attack that targets inputs' privacy without violating privacy and correctness, which cannot be avoided even after considering fairness and robustness.*

Contributions. Our paper gives a positive answer to the question above for secure two-party NN inference realized by garbled circuits (GCs), e.g., [12], [11] (see Section II for their protocol). To this end, the following contributions can be enumerated.

1) Our paper introduces the first fault attack against maliciously secure NN inference engines protected through garbled circuits. The fault attack reduces the security of the scheme to that of an unprotected one, allowing the adversary to mount a model-extraction attack. Our attack cannot be stopped by usual countermeasures ensuring robust and fair execution. The complexity of our attack depends only on the number of network parameters to be extracted, and not on other factors, e.g., the depth of the network. The number of faults is also only linear in the number of parameters.
2) The attack is launched against an implementation of NN inference engines on an FPGA with a general-purpose processor. Such implementation has been widely adopted in the literature [12], [35], [36], [37], [38], [39], [40]. Besides the attack vector exploited in our paper, other possible points of interest from fault analysis' point of view are discussed.
3) Finally, we discuss possible countermeasures and future research direction in that regard.

II. Background and Adversary Model

A. Background on Garbled Circuits

Yao's GC. Yao's GC is a predominant example of MPC with two parties, garbler and evaluator, which is also referred to as the secure function evaluation (SFE) method for Boolean circuits [41], [42], [43]. We highlight the primary building blocks and optimizations within this scheme.

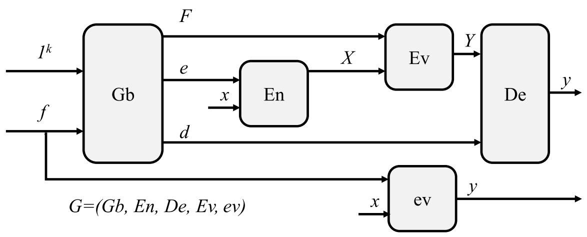

Figure 2: A generic garbling scheme $G = (Gb, En, De, Ev, ev)$ cf. [21]. Note that capital letters on the arrows represent garbled (protected) values/functions while lowercase represents raw (unprotected) ones. ev denotes the typical, unprotected evaluation of the function f against the input x to obtain the output y. F, X, e and d are the counterparts of these in the garbling scheme G that yields y after decoding Y.

Oblivious transfer (OT). In the 1-out-of-2 OT protocol, a sender P_1 has two messages m_0 and m_1. A receiver P_2 with a selection bit $i \in \{0, 1\}$ learns m_i, but not m_{1-i}, while P_1 does not learn i.

Garbling. Garbled circuits enable two parties to compute the correct output of some agreed-upon function f applied to their private inputs without revealing anything else. We use the notions and definitions provided in [21] to formalize this protocol to support modular and practical analyses. In this context, a garbling algorithm Gb is a randomized algorithm involving a degree of randomness. $Gb(f)$ produces a triple of functions $(F, e, d) \leftarrow Gb(f)$ that accepts the function $f : \{0, 1\}^n \rightarrow \{0, 1\}^m$ and the security parameter k. $Gb(f)$ has the following properties. In practice, F is composed of garbled *tables*, encoded four-entry truth table of circuit gates, whose input wires are $L_G^{0,1}$ and $L_E^{0,1}$. The encoding function e converts an initial input $x \in \{0, 1\}^n$ into a garbled input $X = e(x)$, which is given to the function F to generate the garbled output $Y = F(X)$. Here, e encodes a list of tokens (*labels*), one pair for each bit in $x \in \{0, 1\}^n$: $En(e, \cdot)$ uses the bits of $x = x_1 \cdots x_n$ to select from $e = (X_1^{0,1}, X_2^{0,1}, \cdots, X_n^{0,1})$ and obtain the sub-vector $X = X_1^{x_1}, \cdots, X_n^{x_n}$. By reversing this process, the decoding function d generates the final output $y = d(Y)$, which must equal $f(x)$. In other words, f combines probabilistic functions $d \circ F \circ e$. More precisely, the garbling scheme $G = (Gb, En, De, Ev, ev)$ consists of five algorithms as shown in Figure 2, where the strings d, e, f, and F are used by the functions De, En, ev, and Ev.

B. MIPS Processor Overview

The MIPS (microprocessor without interlocked pipeline stages) [44] architecture is a RISC (reduced instruction set computing) [45] processor architecture known for its simplicity and efficiency. This architecture is widely used when implementing GCs [37], [35], [46], [18]. The MIPS architecture uses 32 general-purpose registers, each 32 bits wide, for data processing [47]. These registers are designated as \$0 to \$31, with register \$0 hardwired to zero for simplifying operations. The other registers serve various purposes, including holding temporary values, function arguments, return values, and addresses [48]. MIPS supports arithmetic and logic operations such as addition, subtraction, multiplication, division, `AND`, `OR`, `XOR`, and `NOT` [47], performed directly on register values. It follows a load/store model, requiring data to be loaded from memory into registers or stored from registers into memory for manipulation [47]. MIPS instructions are uniform in size, each being 32 bits long, aiding in the efficient design of the instruction decoder [44].

There are three primary instruction formats in MIPS [44]: *R-Type (register)*, *I-Type (immediate)*, and *J-Type (Jump)*. R-Type is used for arithmetic and logical instructions and includes opcode (6 bits), source register `rs` (5 bits), second source register `rt` (5 bits), destination register `rd` (5 bits), shift amount `shamt` (5 bits), and function code `funct` (6 bits). I-Type is used for operations with immediate values and includes `opcode` (6 bits), source register `rs` (5 bits), destination register `rt` (5 bits), and a 16-bit immediate value. J-Type is used for jump instructions and includes `opcode` (6 bits) and a 26-bit address.

C. Background on Garbled NN Inference

The implementation of the NN model using GC includes two main modules: (1) the linear layer and (2) the ReLU activation function (AF). In adapting the ReLU function within GC, the binary representation of the input x is a fundamental step. This input is expressed over n bits as $x = \sum_{i=0}^{n-1} x_i 2^i$, where each x_i is a binary digit, either "0" or "1". The most significant bit (`MSB(x)`), x_{n-1}, serves as the sign indicator in a two's complement system, signifying the input as negative when $x_{n-1} = 1$. To accommodate the ReLU function for GCs, similar to the prior GC NN frameworks [16], [49], [11], [14], a garbled multiplexer (MUX) circuit becomes crucial. This `MUX` selectively outputs either the actual input x or zero based on the `MSB`. The operation of the `MUX` is encapsulated by the logic: if the `MSB` is "0" (indicating non-negativity), the output should be x; if the `MSB` is "1" (indicating negativity), the output should default to zero. The functional logic of the `MUX` is as follows:

$$\mathtt{MUX}(x, \mathtt{MSB}(x)) = (1 - \mathtt{MSB}(x)) \cdot x$$

This setup ensures that inputs deemed negative result in a zero output, according to the ReLU function's definition.

The linear layers of NN model implementation include the multiplication functions. The steps for implementing multiplication in GC are as follows: first, the weights and the input of the neuron are encrypted to their corresponding labels. In the next step, a bitwise garbled `AND` operation is executed to create partial products. Then, at the last step, a garbled addition circuit is executed to sum the partial products.

D. Adversary Model

Adversary model from MPC perspective. When it comes to the adversary models, some schemes assume that both the client and the NN owner follow the protocol rules, i.e., honest-but-curious (HbC) parties. On the other hand, a *malicious* adversary deviates from the protocol arbitrarily [50]. Malicious activities involve using corrupted inputs or circuits to extract information about the other party's inputs. If the NN owner cheats and is caught acting maliciously, the consequences would be very serious due to public accountability [11], [50]. Clients, on the contrary, can use a wide range of setups under their control to act maliciously to extract the NN model's parameters and obtain a similar model.

In this context, the malicious behavior of both parties has been formulated in the literature [51], [52], where the client's malicious behavior has been mainly studied under the notion of input consistency. Constructing an incorrect, garbled circuit maliciously is particularly associated with the malicious party who generates the circuits, i.e., the NN owner in our scenario, not the client.

Our adversary model follows that of a client-server setting, where the client acts maliciously and attempts to extract the model's weights held by the server cf. [53], [12], [11]. In accordance with related studies on secure MPC-based NN inference, the neural network configuration is known to both client and server. In line with this, the client either has knowledge of the processor layout or is able to profile the processor on the chip to target points of interest. We stress that this is a reasonable assumption, as the processor circuit itself is not supposed to be protected through MPC. Still, the NN model's parameters are proprietary to the NN owner, i.e., the server. We further assume that the malicious client is capable of mounting physical fault attacks, such as laser fault injection, against the edge device running garbled NN. In this regard, the client holds the raw value of her inputs and the NN model output. She takes advantage of the intrinsic characteristics of maliciously-secure inference protocols, where the client evaluates the neural network in a layer-by-layer fashion on a general-purpose processor, see, e.g., [11], [12], [35], [36], [37], [38], [39], [40]. This is a key driver of our attack because by injecting fault into instructions, the adversary can change their functionality and extract the weights in a divide-and-conquer way. In this regard, our attack discloses the weights of NNs that are composed of alternating linear (fully connected, convolutional, etc.) and non-linear ReLU layers as typical blocks of NNs.

III. Methodology

A. Model Extraction Attacks

One of the earliest attempts to extract NN models' weights has been presented by Carlini et al. [54], where the NN was not embedded in any MPC protocol. By carefully choosing inputs that differ slightly and observing the changes in the outputs, the attack identifies critical points where the ReLU activation function (AF) changes its behavior. Such critical points reveal information about the model's parameters. The process is repeated layer by layer, allowing the client to reconstruct the entire neural network with high precision and significantly fewer queries than traditional methods. This attack is extended in [11] to target secure inference protocols that rely on additive secret sharing in the HbC setting. Similar to the attack in [54], layer-by-layer evaluation of NNs is the main ingredient in mounting the attack against those protocols. The attack starts with the last layer of the NN and moves toward the first one, where for each layer, the client *malleate* its shares fed to the intermediate layers. Malleating means adding a small, controlled amount to the intermediate layer input, which changes the final output. By observing these changes, the client can extract the model's actual weights after receiving the final output, which is decoded and in plaintext at the final stage of the protocol. While Carlini et al. [54] leverages the ReLU linear part, the attack in [11] exploits weaknesses in the protocol's implementation and forces ReLU to behave linearly. Both attacks achieve efficient model extraction, but focus on different vulnerabilities.

B. Fault Injection for Model Extraction

Our attack can be seen as a fault-assisted cryptanalysis against garbled NN protocols. Concretely, we focus on NN models containing ReLU, where the protocol evaluates the NN model layer-by-layer. The NN models coming under our attack are fully connected, where -for the sake of simplicity and comparison- the linear layers do not have any additive bias value as also considered in [11]. In contrast to the target in [11], the intermediate values in the garbled NN inference engine are either "0" or "1," making the attack even more straightforward, as explained below.

1) Recovering the last layer's weights: Figure 3 illustrates the interactions between the NN owner and the malicious client when computing the k^{th} layer of the garbled NN on the client's inputs x to obtain the output y. As the first step in launching our attack, the malicious client aims to extract the last layer weights, i.e., $k = \ell$. The client sets her input value $x = \{0\}^n$.

Evaluation as usual. At this stage, the client obliviously receives an array of encrypted input labels $X = (X_1^{0,1}, X_2^{0,1}, \cdots, X_n^{0,1})$, the encrypted inputs labels corresponding to $x = \{0\}^n$ (see Figure 3). The client then honestly evaluates the model until the last layer, i.e., the

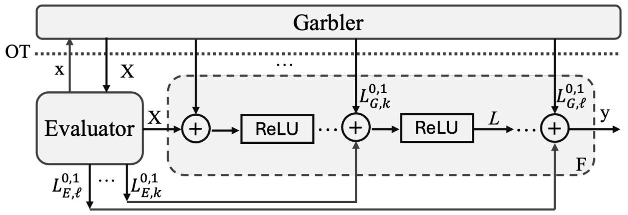

Figure 3: A high-level flow of an iterative GC-based NN inference. $L_{G,k}^{0,1}$ and $L_{E,k}^{0,1}$: garbler's and client's labels for k^{th} layer ($1 \le k \le \ell$). x and X: client's raw and garbled inputs received via OT; y: client's raw outputs; L: the intermediate layer garbled output.

last layer, as $M_\ell(\texttt{ReLU}(M_{\ell-1}(\cdots\texttt{ReLU}(M_1(x)))))$, where $M_k := (\texttt{AND}, \texttt{XOR})$ represents the neural network linear layer operations i.e., an AND followed by ADD operation. $M_\ell \in \mathbb{R}^{m \times t}$ (its image is $\mathbb{R}^m$) denotes the last layer with t connections and m classes to be determined by the NN. Figure 3 illustrates the linear layer execution of the last layer, M_ℓ, for one neuron. To calculate the j^{th} last layer neuron output, Y_j, the garbler label, $L_{G,\ell}^{0,1}$ which is the last layer weights, are multiplied by the client label, $L_{E,\ell}^{0,1}$ which is the output of the previous intermediate layer, as follows

$$Y_j = L_{G,\ell}^{0,1} L_{E,\ell}^{0,1}, \quad 1 \le j \le m,$$

where m is the number of neurons in the last layer. After this step, the client holds the garbled output, $Y = [Y_1, Y_2, \cdots, Y_m]$. Using the received decryption label from the garbler, d, the client decrypts Y and finds its raw output value as $(y_1, y_2, \cdots, y_m) = De(d, Y)$.

Weight disclosure through fault injection. The decrypted output y_j ($1 \le j \le m$) gives no information about the last layer weights w_ℓ (the decrypted version of the $L_{G,\ell}^{0,1}$) as the input to the NN and the biases are "0." Now, if the client injects a fault to change the AND in the last layer to XOR, the decrypted output $y_j = w_\ell \oplus 0 = w_\ell$ (the clients knows the input $x = \{0\}^n$); hence, observing the decrypted output reveals the weights in the last layers.

2) Recovering intermediate layers' weights: At this stage, the client has already extracted the weights in the last layers, e.g., for one neuron in that layer w_ℓ. The extraction of the intermediate layers is similar to the last layer but requires an additional step: forcing all the ReLU AFs after the targeted intermediate layer to behave linearly or like a buffer. After the extraction of the last layer weights, w_ℓ, the client extracts the intermediate layer weights layer by layer, $M_{\ell-1} \cdots M_1$.

Weight disclosure through fault injection. As for the last layer, the client sets its input to $x = \{0\}^n$ and evaluates the NN model honestly. Suppose the target intermediate layer is the k^{th} layer: $M_k(\texttt{ReLU}(M_{k-1}(\cdots\texttt{ReLU}(M_1(x)))))$, where $1 \le k \le \ell - 1$. This process includes a linear operation, M_k, and a non-linear operation, $\texttt{ReLU} = (1 - \texttt{MSB}(x)).x$;

see Figure 3. At this layer, the client injects the fault in the AND operation of the linear layer, M_k, to change it to XOR, similar to the last layer. We denote this faulty linear layer by M'_k. The output of the intermediate layer, L, is $L = \texttt{ReLU}(M'_k(L^{0,1}_{G,k}, L^{0,1}_{E,k})) = \texttt{ReLU}(L^{0,1}_{G,k} \oplus L^{0,1}_{E,k})$, which can be written as follows after integrating the bit-wise operation of ReLU:

$$L = (1 - \texttt{MSB}(L^{0,1}_{G,k} \oplus L^{0,1}_{E,k}))\, \texttt{AND}\, (L^{0,1}_{G,k} \oplus L^{0,1}_{E,k})$$

The ReLU function outputs "0" on negative values; therefore, to propagate the negative weights to L, the client injects another fault into ReLU to force it to behave linearly, i.e., as a buffer. In doing so, instead of $(1 - \texttt{MSB}(x))\, \texttt{AND}\, x$, ReLU operates as $(1\, \texttt{OR}\, \texttt{MSB}(x))\, \texttt{AND}\, x$ (see Section III-C for details). This way, the output of $(1\, \texttt{OR}\, \texttt{MSB}(x))\, \texttt{AND}\, x$ is altered to $(1\, \texttt{AND}\, x) = x$, which means the functionality of the ReLU becomes a buffer. As a result, we have

$$\begin{aligned} L &= (1\, \texttt{OR}\, \texttt{MSB}(L^{0,1}_{G,k} \oplus L^{0,1}_{E,k}))\, \texttt{AND}\, (L^{0,1}_{G,k} \oplus L^{0,1}_{E,k}) \\ &= (L^{0,1}_{G,k} \oplus L^{0,1}_{E,k}). \end{aligned}$$

L is given to the following intermediate layers, where the ReLU functions of the neurons receiving L are faulty:

$$Y_j = M_\ell(M_{\ell-1}(\cdots M_{k+1}(L^{0,1}_{G,k} \oplus L^{0,1}_{E,k}))),$$

where in $M_\ell \cdots M_{k+1}$, $w_\ell \cdots w_{k+1}$ are known values extracted in previous steps. Hence, after decoding, the output of the NN model can be derived by:

$$y_j = w_\ell w_{\ell-1} \cdots w_{k+1}(w_k \oplus 0) = \underbrace{w_\ell w_{\ell-1} \cdots w_{k+1}}_{Known} w_k.$$

This process can be repeated until all layers weights are extracted, refer to V-A for the queries and fault injection requirements for this.

C. Fault Injection into Garbled NN Inference Engines

Implementing GC inference engines on general-purpose processors like MIPS [44] or ARM [55] offers significant benefits, including practicality and efficiency. Moreover, advanced development tools and software co-design enhance accessibility and reduce development time [43]. Here, to explain our attack in a straightforward manner, we focus on one of the most commonly used general-purpose implementations using MIPS I architectures [35], [37]. However, we should note that our attack is not limited to this type of general-purpose processor and can be mounted against other implementations on architectures with a similar instruction set. We first elaborate on the possible locations of fault injection to achieve our attack's desired functionality then choose one of these candidate locations to launch our attack as an example. For this, we use the definitions provided in Section II-B, and the architecture illustrated in Figure 4; see [56] for more information on MIPS I instruction set.

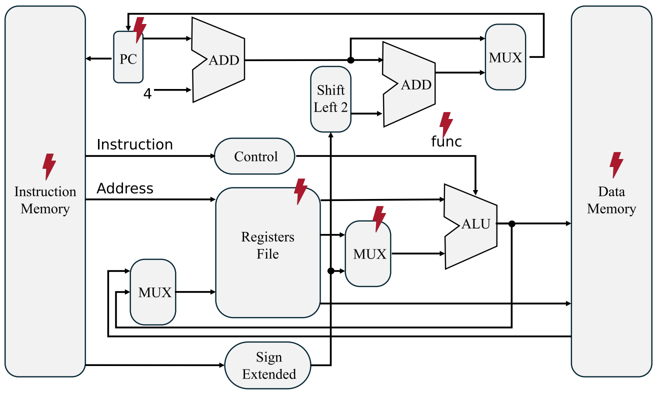

Figure 4: An abstract illustration of general-purpose processor architectures, inspired by [35], and possible fault injection locations.

Program counter (PC). During the processor's instruction execution, the first step is to run Instruction Fetch (IF) to fetch the instruction from the instruction memory. The PC, which holds the address of the current instruction, provides this address to the instruction memory. The instruction at that address is then returned, and the PC is incremented by 4 to point to the next instruction. The client can inject a fault to change the PC value and force the core to fetch an alternate instruction, such as XOR instead of AND operation in the linear layer execution.

Decoded instruction. In the second step, instruction decode (ID), the fetched instruction is decoded to understand what operation it is supposed to perform. Changing the funct register is the second possible location for fault injection. In this case, the client launches the fault attack to change the function of the instruction within funct register, to force the core to execute an arbitrary function.

Read memory registers. In the third step, register read (RR), the values in the source registers (rs and rt) are read from the register file, which stores the 32 general-purpose registers. In the case of an unprotected NN model evaluation protocol, the client can change the value of the source registers to a desired value. However, in the context of GC protocol, the client cannot modify the value to a known specific value due to the GC intrinsic protection of data through garbling [21].

ALU. In the fourth step, execution (EX), the ALU performs the operation specified by the instruction. The ALU Control determines the specific operation based on the funct field and the opcode. For instance, if the funct field indicates an ADD operation, the ALU adds the values read from the source registers and stores the result. This step is the third possible location of fault injection in the context of GC, as the client can alter the ALU logic or control flow to force the core to execute an alternate function, such as forcing the ReLU to behave linearly as a buffer.

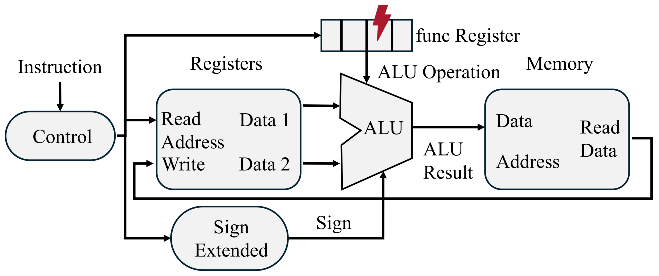

Figure 5: A high-level illustration of the control signals, ALU procedure, and the location of our fault attack.

The destination register. In the fifth step, Memory Access (MEM), memory is accessed if needed, which is generally not required for R-type instructions as they do not involve memory access. In the final step, write back (WB), the result of the ALU operation is written back to the destination register. Similar to the fault injection in read memory registers, the client can change the value of the destination register in an unprotected NN model evaluation protocol; nevertheless, since the data is garbled in the context of GC, the client cannot change it to a desired known value.

Among these possible fault injection locations, we elaborate on the fault injection against the *decoded instruction* as an example. We emphasize that this does not rule out the possibility that the client can exploit other possible fault locations.

D. Fault Injection in NN model's Decoded Instruction

We divide our attack into two parts: (1) change the operation of the linear layer from AND to XOR to extract the last layer weights and to propagate the intermediate layer weights to the next layers, and (2) force ReLU to behave linearly so that it acts as a buffer instead of ReLU. The latter prevents ReLU from changing the negative intermediate weights values to zero in order to extract the intermediates layer's weights (see Section III-B). Figure 5 depicts a high-level presentation of control signals, ALU procedure, and our attack fault injection location. To launch the first part of our attack, the manipulation of the ALU function (AND $\rightarrow$ XOR), we inject the fault in the func register, illustrated in Figure 5, during the execution of the last linear layer. This fault injection results in changing the ALU functionality from AND to XOR, where the reason is explained in Section III-B. This change ensures the propagation of the target weight to the last layer. Table I contains the list of the ALU functions in the R-Type instructions.

Recovering the last layer's weights: As shown in Table I, the register value corresponding to AND, $6'b000110$, has only one-bit difference with the register value corresponding to XOR, $6'b000111$ (the least significant bit (LSB) is different). In practice, to launch the attack, the client

Table I: ALU function register value in MIPS I architecture

Function	Binary Code	Function	Binary Code
NOTHING	6'b000000	OR	6'b000101
ADD	6'b000001	AND	6'b000110
SUBTRACT	6'b000010	XOR	6'b000111
LESS_THAN	6'b000011	NOR	6'b001000
LESS_SIGNED	6'b000100		

tracks the instruction decode path and determines the func register location on the die. At the beginning of the time window corresponding to the execution of the last layer, the client targets the first bit of the func register. Afterward, the operation of XOR takes place at the core, and the last layer weights are propagated to the core output.

Recovering intermediate layers' weights: To extract the intermediate layers' weights, the attack must first change the operation of the target intermediate layer from AND to XOR, similar to the attack against the last layer; then, she forces all the ReLU functions from the layer that she targets on to behave linearly, like a buffer. As explained in Section III-C, the ReLU function in the context of GC is implemented as $(1-\texttt{MSB}(x))\texttt{AND}\,x$. Hence, the ReLU function is compiled within the MIPS I instruction set as the following two operations: `SUB $result, $Constant_1, $MSB;` or `AND $ReLUOutput, $result, $x;` The client aims to change the first operation from $1-\texttt{MSB}(x)$ to $1\texttt{OR}\texttt{MSB}(x)$, the ReLU function turns from $(1-\texttt{MSB}(x))\,\texttt{AND}\,x$ to $1\,\texttt{AND}\,x$ as the OR of any value with 1 equals 1 and the AND with 1 buffers the input to the output.

To obtain the faulty ReLU, the client changes the func register from SUB ($6'b000010$) to OR ($6'b000101$). To do this, the client injects the fault in the 3 LSB bits of the func register to flip them, $010 \rightarrow 101$. After that, first, the core calculates the OR logic of the $\texttt{MSB}(x)$ register and `Constant_1`, which always results in `Constant_1`, then executes the AND operation on the `Constant_1` and x. As the AND operation between x and $Constant_1$ value always gives x, the value of x is then propagated to the output, unchanged. This process is repeated for each weight, which requires multiple faults and queries as discussed in Section V-A.

IV. Experimental Setup

A. Device Under Test

A Genesys 2 development kit was employed for the laser fault injection experiment. This kit contains an AMD/Xilinx Kintex 7 (XC7K325T-2FFG900C) FPGA fabricated with 28 nm technology. The FPGA die is in a flip-chip package. By removing the fan and heat spreader, access was gained to the backside silicon of the FPGA (see Figure 6). No additional modifications were made to the package or board, e.g., silicon polishing. The FPGA core was operated at 1.0 V, and the clock frequency was 200 MHz for all experiments.

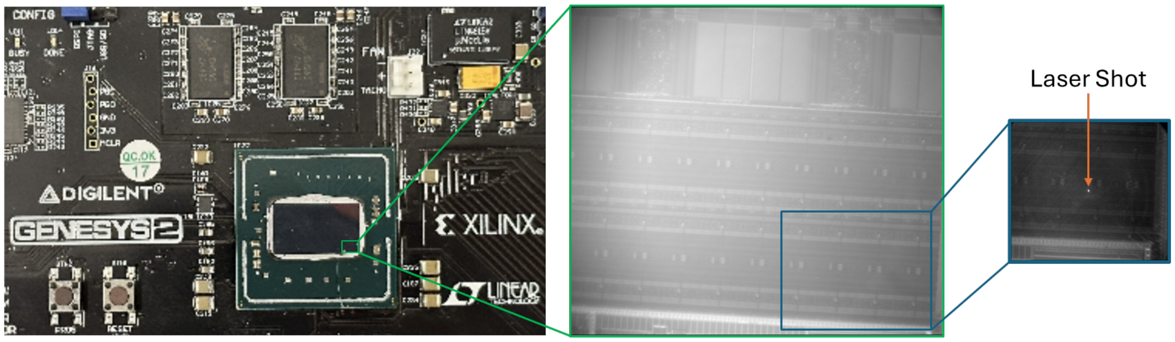

Figure 6: Iterative magnification of the device under the AlphaNOV setup (from left to right): the Genesys2 board and the die shown is the Kintex 7 FPGA with the heatsink removed; the middle image depicts the die using the 20X lens to show the corner where the FF for fault is placed; Lastly, the right-most image is captured using the 50X lens, illustrating the fault injection at the point of interest (the white dot corresponds to the laser shot).

B. *Laser Setup*

We used an ALPhANOV S-LMS [57] setup for near-infrared (NIR) microscopy and laser fault injection. The microscope consists of a camera system for capturing images and a lens on an XYZ stage to focus on a region. We used 20X and 50X magnification lenses for the experiment. The 20X lens is a standard Resolution (NA=0.6) with a typical field of 480 $\times$ 380 μm, while the 50X lens is an Ultra High Resolution (NA=0.7) with a typical field of 190 $\times$ 150 μm. The combined setup is controlled using the software and hardware switches to control the XYZ stage and camera options. The software provides an IR view of the die, which can be used for navigation. To capture the live feed of the laser shots, the integration time for the image was kept at 0.1 ms with a frame rate of 60 Hz (the refresh rate of the display).

The laser source used for the experiment is the High Pulse Performance PDM laser source. The wavelength of the laser is 1064 nm. The peak current used for the fault is 1 A, while the pulse width is 250 ns with a frequency of 100 kHz. The laser is controlled by the AplhaNOV control software in combination with viewing the die using the camera software. The combination results in a live feed to laser shots on the desired fault region (a screenshot shown in 6).

C. *Hardware Implementation*

The FPGA and the microscope with controlling devices form our experimental setup. Before programming the FPGA, the laser setup is prepared by focusing the lens on the desired location for the laser shot. The FPGA bitstream is programmed then using Vivado 2021 [58] from a computer. Once programmed, the design will generate a trigger signal connected to the AplhaNOV laser module. This trigger signal is used to control the laser pulse timing. The laser is shot continuously using the settings mentioned in the laser setup. The Genesys 2 development board has user-programmable outputs used as flags to show a successful fault. For more details, please see Section V-C.

V. Results

A. *Complexity of the Attack: Number of Faults and Queries*

Here, we elaborate on the number of queries and faults required to extract the weights of some common NN models as discussed in the most relevant literature [11], [54]. To extract the last layer weights, the client only needs 1 fault per weight as $L^{0,1}_{G,\ell}$ is the corresponding label of 0 and 0 multiplied by any value results in 0; therefore, only the weight of the faulty neuron with `AND` $\rightarrow$ `XOR` is propagated to the output. Hence, for extracting the weights in the last layer, $\#fault = p_\ell$, where p_ℓ is the number of parameters in the last layer.

To extract the intermediate weights, the same fault as in the last layer must be injected to propagate the target weight to the `ReLU` input. Another fault must be injected in the target layer's `ReLU` to make it a buffer. Additionally, one fault in all the neurons' linear calculation from the target layer to the last layer, $M_k, M_{k+1}, \cdots, M_\ell$, must be injected to propagate the target weight value. Furthermore, the `ReLU` of these layers has to be forced to act linearly like a buffer, which results in an additional fault. Hence, we calculate the number of faults to extract the NN model as follows:

$$\#\text{faults} = (\underbrace{\ell(\ell-1)/2}_{\texttt{ReLU}} + \underbrace{\ell(\ell-1)/2}_{\texttt{AND}\rightarrow\texttt{XOR}}) \cdot (p - p_\ell) + p_\ell = O(\ell^2 p)$$

where p is the total number of the NN model parameters, p_ℓ is the number of parameters in the last layer, and ℓ is the number of the NN model layers. Moreover, the number of queries is the same as the number of parameters, as per parameter, input is given to the NN; therefore, the query complexity of the attack is similar to that of a state-of-the-art model extraction attack, although we target a much harder scheme protected against the malicious adversary.

Table II shows the comparison between our attack against GC NN inference protected against the malicious adversary,

Table II: Query and fault complexity of our attack vs. [11] and [54] (the number of faults is only applicable to our attack). While our attack targets maliciously-secure NN inference engine, [11] and [54] consider HbC-secure and unprotected designs, respectively, which are indeed simpler to be attacked.

Network Dimensions	# Parameters	#Queries			#Fault
		[11]	[54]	Ours	
784-128-1	100,480	100,480	$2^{21.5}$	100,480	200,832
784-32-1	25,120	25,120	$2^{19.2}$	25,120	50,208
10-10-10-1	210	210	2^{16}	210	610
10-20-20-1	620	620	$2^{17.1}$	620	1820

the cryptanalysis in [11] that has been mounted against HbC-secure inference engine, and the cryptanalysis in [54], which targets unprotected NN models. [11] and [54] did *not* launch any fault attack against NN models; therefore, the number of faults is only applicable to our attack. In Table II, our attack requires up to 30× fewer queries compared to [54]. It matches the queries requirement in [11] to extract the entire NN model weights, even though their attack has launched on an *HbC-secure* NN model while our attack is launched against a *maliciously-secure* NN model. This shows that the client can extract a maliciously secure NN model without extra query requirements. Our attack, however, requires the fault injection, which the client can do.

B. Simulation Results

To evaluate the success of our attack against garbled NN inference engines, we first simulate the impact of the fault. We implemented a proof-of-concept multi-layer perception (MLP), hereafter called *target model*, with two hidden layers, each with 5 neurons, a last layer with 10 neurons, and an input layer with 5 inputs, similar to benchmark MLPs [18], [12], [59] (for results on scalability of the attack, see Section V-A). We used the GC Lite_MIPS implementation proposed in [60] to evaluate the target model. In doing so, given an input, we can observe the labels and the output in order to assess the impact of fault injection. To compile the MLP to MIPS I instructions, we use GNU cc [61] as advised in [35]. To simulate the fault injection, we utilize SystemVerilog assertion (SVA) in Vivado Suite 2023 [62]. We set the clock period to 50ns. In the simulation results, the value of the `alu_func` register follows the order of ALU functions in Table I, starting from 0 to 8.

Figure 7 shows the simulation results for the `alu_func` register during the execution of the target model using MIPS I. The blue rectangle is the `alu_func` register value during the execution window of one neuron in the first hidden layer, including the neuron multiplication and the ReLU . The purple rectangle shows the execution of one multiplication ($\texttt{AND} = 6'b000110$) followed by one summation ($\texttt{ADD} = 6'b000010$). The yellow rectangle shows the execution of the ReLU at the end of the neuron multi-

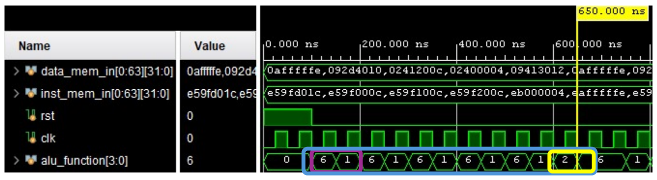

Figure 7: Simulation of the `alu_func` register during the computation of one neuron in the *first* layer (blue: the execution window of one neuron, purple: the execution window of one multiplication and summation corresponding to each connected input; yellow: the execution of the ReLU).

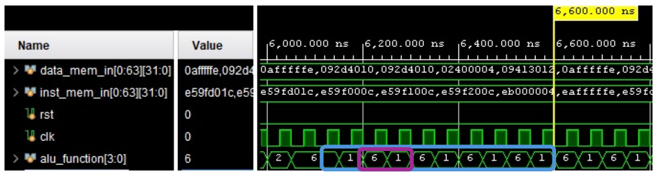

Figure 8: Simulation of the `alu_func` register during the computation of one neuron in the last layer (blue: the execution window of one neuron; purple: the execution window of one multiplication and summation corresponding to each connected input).

plication, which includes a subtraction ($\texttt{SUB} = 6'b000011$) and a multiplication ($\texttt{ADD} = 6'b000010$). The blue execution window is repeated for each connected input value. As it can be observed in Figure 7, each neuron execution in the first hidden layer of the target model takes 12 clock cycles: two clock cycles for each input connected to a neuron (10 clock cycles for 5 inputs), and 2 clock cycles for the ReLU . Hence, the execution of each neuron in the hidden layer takes $12 \times 50\text{ns}= 600\text{ns}$, where 50ns is our defined clock period. The target model includes two hidden layers and 5 neurons in each layer; therefore, the same execution process shown in Figure 7 takes 6000ns in total. Afterward, the linear layer and the last layer are executed.

Figure 8 shows the simulation of the `alu_func` register in the execution window of one neuron placed in the target model's last layer. The blue rectangle is the `alu_func` register value during that period, whereas the purple rectangle shows the execution of one multiplication ($\texttt{AND} = 6'b000110$) followed by one summation ($\texttt{ADD} = 6'b000010$). According to Figure 8, the execution time of each neuron in the last layer takes 2 clock cycles per connection. With 5 connections per neuron, 10 clock cycles passed to run the computation of a neuron, equal to 500ns. The target model's last layer includes 10 neurons; therefore, the execution process shown in Figure 7 takes $500 \times 10 = 5000\text{ns}$ in total.

1) Fault injection in the last layer: We target the second neuron of the last layer as an example to demonstrate the successful change in `alu_func` due to the fault injection;

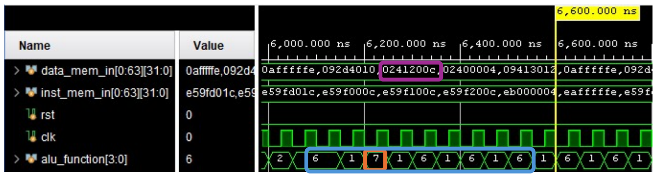

Figure 9: Simulation of the `alu_func` register during the computation of one neuron in the last layer after fault injection (blue: the execution window of one neuron, purple: the changed data register value due to the fault injection, orange: the value of the `alu_func` after fault injection).

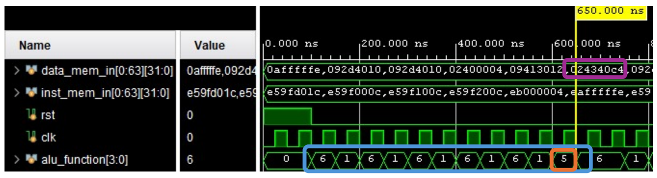

Figure 10: Simulation of the `alu_func` register during the computation of one neuron in the first *intermediate layer* (blue: the execution window of one neuron; purple: the changed data register value due to the fault injection; orange: the value of the `alu_func` after fault injection).

see Figure 9 for the result. The orange rectangle depicts the effect of fault injection in the `alu_func` register. The fault is injected on the LSB of `alu_func` register, and as a result, `alu_func` register value changed from 6 to 7, which means the core executes the $\texttt{XOR} = 6'b0000111$ instead of the $\texttt{AND} = 6'b0000110$.

The purple rectangle shows the changed output data after the fault injection. To assess the impact of our attack, we obtain the garbled output label for the target neuron (the second neuron in the last layer). The weight of this neuron is intentionally set to 1 to observe the output when changing AND and XOR on the label 0 and weight 1. Before the attack, according to Figure 8, the core generates "$0x092d4010$," which is the corresponding label of 0 calculated as $0 \texttt{AND} 1 = 0$ based on the output given by the tool [60]. After injecting the fault, as illustrated in Figure 9, the core generates "$0x0241200c$." This value corresponds to the label of 1 according to the results we get from the tool [60]. This value is generated as the result of the operation XOR on input 0 and weight value 1, $0 \oplus 1 = 1$, which confirms the success of our attack to extract the second neuron on the last layer. To extract the weights of the rest of the neurons in the last layer, the same steps with the same fault location can be launched but in a different time frame.

2) Fault injection in the intermediate layers: Launching our attack against intermediate layers includes repeating steps of injecting fault against the last layer on the target intermediate layer and extra fault injection to force the ReLU in the rest of the path to the last layer to behave linearly, as a buffer.

To force the ReLU to behave linearly, we target the first neuron in the first hidden layer, where Figure 10 shows the simulation results for that. The orange rectangle illustrates where the fault is injected into the `alu_func` register during the execution of the ReLU. As it is observable in Figure 10, the operation of $\texttt{SUB} = 6'b000010$ is changed to $\texttt{OR} = 6'b000101$ after the fault injection. This means the ReLU operation is changed from $(1 - \texttt{MSB}(x)) \, \texttt{AND} \, x$ to $(1 \, \texttt{OR} \, \texttt{MSB}(x)) \, \texttt{AND} x = 1 \, \texttt{AND} \, x = x$. This changes the ReLU function to a buffer and propagates the output of the target neuron to the output of the ReLU. To check its effect, we intentionally set the intermediate weights to -1 and all its input to 1; hence, the neuron outputs -5, as it is connected to five inputs. The core passes the neuron output to the ReLU, and the ReLU's output becomes 0, as its input is a negative value. After the fault injection, we observe the changes in the LSB of the ReLU output. If the ReLU's output is 0, the LSB is the label corresponding to that value. If the output of the ReLU is a negative value, the LSB becomes 1, as the core uses the extended sign format, which is the two's complement of the positive value: $5 = 0b101$ and $-5 = 0b011$.

As observable in Figure 10, the value of the data register LSB becomes "$0x024340c4$," which is the label corresponding to value 1 according to the labels generated by the tool [63]. However, the data register LSB in the absence of any fault register is "$0x0afffffa$", according to the results illustrated in Figure 7, the corresponding label of value 0 generated by the tool [60]. This confirms that our fault injection changes the functionality of the ReLU to a buffer. Using this step and the fault injection on the linear layer, a client can extract the entire NN model weights by injecting fault (see Section III-B for the steps in the cryptanalysis).

C. Laser Fault Injection Results

The laser fault injection is used to validate the practicality of the results presented in section V-B. The design used for the laser fault is the same MIPS implementation with the neural network program stored in the program memory. The flip-flop for the opcode is located in slice X1Y138 of the Kintex 7 FPGA This slice is located at one of the corners of the FPGA; see Figure 6. First, the 20X lens is used to navigate to the corner, and then the 50X lens to localize the target slice and irradiate the laser. Our target is the opcode FF for the last layer to demonstrate that we can flip the bit to change the instruction from `AND` to `XOR`. Once flipped, we expect the same behavior as shown in the simulated fault. To ensure that the fault only affects the target bit and not other portions of the circuit, we deployed flag outputs to check for multiple stages of the fault and the correct output. These flags also help us to filter out transient faults. First, the most simple flag is the fault bit, i.e., the bit of the opcode that we are faulting. Next, we used an output to compare the ALU output with the desired known faulty value after the fault injection. Lastly, we checked the next instruction output to see if the program continued working correctly. From our experiment, we got all 3 outputs to trigger, demonstrating successful weight extraction from the last layer. Following the same procedure, the laser fault injection could also be applied to other layers.

VI. Discussion

A. Why Cut-and-choose Fails to Prevent Our Attack

Countermeasures devised to stop malicious parties leverage various techniques, with cut-and-choose constructions being one of the most prominent ones [23], which we focus on due to its widespread application in schemes. The principle behind the cut-and-choose technique is simple but effective: a set of circuits presumably computing the same function are generated and sent to the client. After selecting some of the circuits to be evaluated, the client inspects to verify that all the check circuits have been generated correctly. Despite its efficacy in preventing malicious circuit generation, it does not account for the client's malicious behavior.

In fact, the cut-and-choose approach [64] forces the party generating the garbled circuit, i.e., garbler, to stick with the correct circuit. From the theory point of view, this indicates that cut-and-choose would not thwart our attack. From a practical perspective, one should consider when to launch an attack if a scheme's security is boosted through cut-and-choose. Under this scenario, first, the garbler constructs a large number of garbled circuits sent to the client, who chooses a subset of the circuits to open and check. If all of these circuits are correct, the second step is as follows: the client runs the protocol against all the remaining circuits and takes the majority output value as the output. Our malicious adversary follows these steps accurately, but inject faults into only one circuit in the second step and observe the output to extract the weights. This process clearly does not violate the principle of the maliciously secure GCs.

B. Vulnerabilities of Other General-purpose Processors

General-purpose processors, including MIPS [44], ARM [55], x86 [65], PowerPC [66], SPARC [67], and RISC [45], all utilize the fundamental fetch-decode-execute cycle to process instructions. During the fetch stage, the processor retrieves the instruction from memory based on the PC. In the decode stage, the instruction is interpreted to determine the operation to be performed by the core. The results of the decode stage are stored in a set of registers as control signals, similar to the func register explained in Section II-B, which direct the ALU to execute the specified arithmetic or logical operation using the provided operands. This systematic approach ensures efficient processing of a wide range of tasks.

These control signal registers, containing crucial operational directives for the ALU, can become targets of fault injection attacks if an adversary identifies their locations on the hardware platform. Since the architectural information for these processors is publicly available [44], [55], [66], [65], [67], [45], adversaries can potentially manipulate the ALU function operands register, func registers, leading to the manipulation of the ALU operations. Despite the differences in the design of the architectures mentioned above, the fetch-decode-execute cycle remains a core operational process across them. Hence, our attack could potentially be conducted against these architectures by adjusting the value of the func register to the desired ALU operation value. This should be done according to the target architecture instruction sets information, as described in Section III-C.

C. Possible Countermeasures

Our attack's success is based on our knowledge of the NN model function and the general-purpose processor architectures. Knowing these two key factors, the client can find the correct location to inject a fault and extract the NN model assets, although it might be protected through GC. A possible way of combating our attack is to evaluate it in the context of private function evaluation (PFE). In the PFE setting, the function is also garbled along with the data to prevent the attacker from knowing how the function is computed. An implementation of PFE can be found in the GarbledCPU [37], [46], [18]. Although effective against our attack, it suffers from tremendous resource overhead, a burden for cost-efficient implementation.

Another possible countermeasure is the interactive fault-tolerant maliciously secure GC frameworks. Two examples of such frameworks are MiniLEGO [68] and TinyLEGO [31]. Utilizing the XOR-homomorphic commitments [68], they can mitigate fault injection attacks against implementations. In such settings, the circuit's XOR-homomorphic commitments ensure that each gate's output is a function of the entire path reaching it in the circuit instead of solely the gate's inputs. Therefore, if any fault occurs in any part of the path, the outputs of the gates on the path to the target gate become invalid as they no longer follow the XOR-homomorphic properties of the commitment scheme. Needless to say, these frameworks impose the computation overhead and require massive communication between parties.

VII. Conclusion

In this paper, we present the first fault attack against secure NN inference implementations relying on GC, a prime example of MPC schemes. Our work demonstrates that laser fault injection, along with a model-extraction attack, can effectively break the security of existing solutions. This implies that the private inputs of the NN owner, i.e., the NN's parameters, are disclosed. Remarkably, the number of queries required for our attack is equivalent to that of the state-of-the-art model-extraction attacks, although garbled NN inference is much harder to target than what has been considered in the literature. This highlights a critical vulnerability of current secure inference protocols that depend on MPC. Our findings emphasize the urgent need for robust countermeasures to address fault injection threats. We have further discussed possible countermeasures to ensure the privacy and security of deep learning models and user data in practical applications. Given the shortcomings of possible countermeasures, future research should focus on developing advanced fault-tolerant techniques to fortify secure MPC frameworks against such attacks.

References

[1] Q. Zhang, C. Xin, and H. Wu, "Privacy-preserving deep learning based on multiparty secure computation: A survey," *IEEE Internet of Things Journal*, vol. 8, no. 13, pp. 10412–10429, 2021.

[2] Z. Á. Mann, C. Weinert, D. Chabal, and J. W. Bos, "Towards practical secure neural network inference: the journey so far and the road ahead," *ACM Computing Surveys*, vol. 56, no. 5, pp. 1–37, 2023.

[3] W. G. Hatcher and W. Yu, "A survey of deep learning: Platforms, applications and emerging research trends," *IEEE access*, vol. 6, pp. 24411–24432, 2018.

[4] J. Breier, X. Hou, D. Jap, L. Ma, S. Bhasin, and Y. Liu, "Practical fault attack on deep neural networks," in *Proc. of the 2018 ACM SIGSAC Conf. on Comp. and Communications Security*, pp. 2204–2206, ACM, 2018.

[5] L. Batina, S. Bhasin, D. Jap, and S. Picek, "CSI NN: Reverse engineering of neural network architectures through electromagnetic side channel," in *28th USENIX Security Symp. (USENIX Security 19)*, pp. 515–532, 2019.

[6] M. M. Alam, S. Tajik, F. Ganji, M. Tehranipoor, and D. Forte, "Ram-jam: Remote temperature and voltage fault attack on fpgas using memory collisions," in *2019 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, pp. 48–55, IEEE, 2019.

[7] X. Hou, J. Breier, D. Jap, L. Ma, S. Bhasin, and Y. Liu, "Security evaluation of deep neural network resistance against laser fault injection," in *2020 IEEE Intrl. Symposium on the Physical and Failure Analysis of Integrated Circuits (IPFA)*, pp. 1–6, IEEE, 2020.

[8] J. Breier, D. Jap, X. Hou, S. Bhasin, and Y. Liu, "Sniff: reverse engineering of neural networks with fault attacks," *IEEE Transactions on Reliability*, vol. 71, no. 4, pp. 1527–1539, 2021.

[9] S. Tajik and F. Ganji, "Artificial neural networks and fault injection attacks," in *Security and Artificial Intelligence: A Crossdisciplinary Approach*, pp. 72–84, Springer, 2022.

[10] D. M. Mehta, M. Hashemi, D. S. Koblah, D. Forte, and F. Ganji, "Bake it till you make it: Heat-induced power leakage from masked neural networks." Cryptology ePrint Archive, Paper 2023/076, 2023.

[11] R. Lehmkuhl, P. Mishra, A. Srinivasan, and R. A. Popa, "Muse: Secure inference resilient to malicious clients.," in *USENIX Security Symposium*, pp. 2201–2218, 2021.

[12] M. S. Riazi, M. Samragh, H. Chen, K. Laine, K. Lauter, and F. Koushanfar, "{XONN}:{XNOR-based} oblivious deep neural network inference," in *28th USENIX Security Symp. (USENIX Security 19)*, pp. 1501–1518, 2019.

[13] S. Hussain, B. Li, F. Koushanfar, and R. Cammarota, "Tinygarble2: Smart, efficient, and scalable yao's garble circuit," in *Proc. of the 2020 WKSP on Privacy-Preserving Machine Learning in Practice*, pp. 65–67, 2020.

[14] B. D. Rouhani, S. U. Hussain, K. Lauter, and F. Koushanfar, "Redcrypt: Real-time privacy-preserving deep learning inference in clouds using fpgas," *ACM Trans. on Reconfigurable Technology and Systems (TRETS)*, vol. 11, no. 3, pp. 1–21, 2018.

[15] B. D. Rouhani, M. S. Riazi, and F. Koushanfar, "Deepsecure: Scalable provably-secure deep learning," in *Proc. of the 55th annual design automation Conf.*, pp. 1–6, 2018.

[16] M. Ball, B. Carmer, T. Malkin, M. Rosulek, and N. Schimanski, "Garbled neural networks are practical," *Cryptology ePrint Archive*, 2019.

[17] X. Chen, Z. Chen, B. Dong, S. Wei, L. Chen, and D. He, "Fobnn: Fast oblivious binarized neural network inference," *arXiv preprint arXiv:2405.03136*, 2024.

[18] M. Hashemi, S. Roy, D. Forte, and F. Ganji, "Hwgn 2: Side-channel protected nns through secure and private function evaluation," in *Security, Privacy, and Applied Cryptography Engineering: 12th Intrl. Conf., SPACE 2022, Jaipur, India, December 9–12, 2022, Proceedings*, pp. 225–248, 2022.

[19] J. Sander, S. Berndt, I. Bruhns, and T. Eisenbarth, "Dash: Accelerating distributed private machine learning inference with arithmetic garbled circuits," *arXiv preprint arXiv:2302.06361*, 2023.

[20] Z. Chen and X. Chen, "Secure computation framework for multiple data providers against malicious adversaries," *arXiv preprint arXiv:2007.14915*, 2020.

[21] M. Bellare, V. T. Hoang, and P. Rogaway, "Foundations of garbled circuits," in *Proc. of the 2012 ACM Conf. on Computer and Comm. security*, pp. 784–796, 2012.

[22] S. Goldwasser, Y. T. Kalai, and G. N. Rothblum, "One-time programs," in *Annual Intrl. Cryptology Conf.*, pp. 39–56, Springer, 2008.

[23] Y. Lindell and B. Pinkas, "Secure two-party computation via cut-and-choose oblivious transfer," *J. of cryptology*, vol. 25, pp. 680–722, 2012.

[24] I. Levi and C. Hazay, "Garbled circuits from an sca perspective: Free xor can be quite expensive. . .," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, vol. 2023, no. 2, p. 54–79, 2023.

[25] M. Hashemi, D. Forte, and F. Ganji, "Time is money, friend! timing side-channel attack against garbled circuit constructions," in *Intrl. Conf. on Applied Cryptography and Network Security*, pp. 325–354, Springer, 2024.

[26] D. Beaver and S. Goldwasser, "Multiparty computation with faulty majority," in *Conf. on the Theory and Application of Cryptology*, pp. 589–590, Springer, 1989.

[27] M. Ben-Or, S. Goldwasser, and A. Wigderson, "Completeness theorems for non-cryptographic fault-tolerant distributed computation," in *Providing sound foundations for cryptography: on the work of Shafi Goldwasser and Silvio Micali*, pp. 351–371, 2019.

[28] D. Beaver, "Secure multiparty protocols and zero-knowledge proof systems tolerating a faulty minority," *J. of Cryptology*, vol. 4, pp. 75–122, 1991.

[29] Y. Aumann and Y. Lindell, "Security against covert adversaries: Efficient protocols for realistic adversaries," *J. of Cryptology*, vol. 23, no. 2, pp. 281–343, 2010.

[30] Y. Lindell, "Secure multiparty computation," *Communications of the ACM*, vol. 64, no. 1, pp. 86–96, 2020.

[31] T. K. Frederiksen, T. P. Jakobsen, J. B. Nielsen, and R. Trifiletti, "Tinylego: An interactive garbling scheme for maliciously secure two-party computation," *Cryptology ePrint Archive*, 2015.

[32] S. Goldwasser, S. Micali, and C. Rackoff, "The knowledge complexity of interactive proof-systems," in *Providing sound foundations for cryptography: On the work of shafi goldwasser and silvio micali*, pp. 203–225, 2019.

[33] N. Koti, M. Pancholi, A. Patra, and A. Suresh, "{SWIFT}: Super-fast and robust {Privacy-Preserving} machine learning," in *30th USENIX Security Symposium (USENIX Security 21)*, pp. 2651–2668, 2021.

[34] L. K. Ng and S. S. Chow, "Sok: Cryptographic neural-network computation," in *2023 IEEE Symposium on Security and Privacy (SP)*, pp. 497–514, IEEE, 2023.

[35] E. M. Songhori, S. U. Hussain, A.-R. Sadeghi, T. Schneider, and F. Koushanfar, "Tinygarble: Highly compressed and scalable sequential garbled circuits," in *2015 IEEE Symp. on Security and Privacy*, pp. 411–428, IEEE, 2015.

[36] E. M. Songhori, M. S. Riazi, S. U. Hussain, A.-R. Sadeghi, and F. Koushanfar, "Arm2gc: Succinct garbled processor for secure computation," in *Proc. of the 56th Annual Design Automation Conf. 2019*, pp. 1–6, 2019.

[37] E. M. Songhori, T. Schneider, S. Zeitouni, A.-R. Sadeghi, G. Dessouky, and F. Koushanfar, "Garbledcpu: A mips processor for secure computation in hardware," in *2016 53nd ACM/EDAC/IEEE Design Automation Conf. (DAC)*, pp. 1–6, IEEE, 2016.

[38] V. Kolesnikov, J. B. Nielsen, M. Rosulek, N. Trieu, and R. Trifiletti, "Duplo: unifying cut-and-choose for garbled circuits," in *Proceedings of the 2017 ACM SIGSAC Conf. on Computer and Communications Security*, pp. 3–20, 2017.

[39] J. Mo, J. Gopinath, and B. Reagen, "Haac: A hardware-software co-design to accelerate garbled circuits," in *Proceedings of the 50th Annual Intrl. Symposium on Computer Architecture*, pp. 1–13, 2023.

[40] V. Kolesnikov, "Free if: How to omit inactive branches and implement s-universal garbled circuit (almost) for free," in *Intrl. Conf. on the Theory and Application of Cryptology and Information Security*, pp. 34–58, Springer, 2018.

[41] Y. Lindell and B. Pinkas, "A proof of yao's protocol for secure two-party computation. eccc report tr04-063," in *Electronic Colloquium on Computational Complexity (ECCC)*, 2004.

[42] Y. Lindell and B. Pinkas, "A proof of security of yao's protocol for two-party computation," *J. of cryptology*, vol. 22, no. 2, pp. 161–188, 2009.

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

[44] G. Kane, *mips RISC Architecture*. Prentice-Hall, Inc., 1988.

[45] D. A. Patterson and C. H. Sequin, "Risc i: A reduced instruction set vlsi computer," in *25 years of the Intrl. symposia on Computer architecture (selected papers)*, pp. 216–230, 1998.

[46] M. Hashemi, S. Roy, F. Ganji, and D. Forte, "Garbled eda: Privacy preserving electronic design automation," in *Proceedings of the 41st IEEE/ACM Intrl. Conf. on Computer-Aided Design*, pp. 1–9, 2022.

[47] J. L. Hennessy and D. A. Patterson, *Computer architecture: a quantitative approach*. Elsevier, 2011.

[48] D. A. Patterson and J. L. Hennessy, *Computer organization and design ARM edition: the hardware software interface*. Morgan kaufmann, 2016.

[49] C. Juvekar, V. Vaikuntanathan, and A. Chandrakasan, "GAZELLE: A low latency framework for secure neural network inference," in *27th USENIX Security Symp. (USENIX Security 18)*, pp. 1651–1669, 2018.

[50] Y. Lindell, "Fast cut-and-choose-based protocols for malicious and covert adversaries," *J. of Cryptology*, vol. 29, no. 2, pp. 456–490, 2016.

[51] H. Carter, B. Mood, P. Traynor, and K. Butler, "Secure outsourced garbled circuit evaluation for mobile devices," *J. of Computer Security*, vol. 24, no. 2, pp. 137–180, 2016.

[52] Y. Lindell and B. Riva, "Blazing fast 2pc in the offline/online setting with security for malicious adversaries," in *Proceedings of the 22nd ACM SIGSAC Conf. on Computer and Communications Security*, pp. 579–590, 2015.

[53] F. Tramèr, F. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart, "Stealing machine learning models via prediction {APIs}," in *25th USENIX security symposium (USENIX Security 16)*, pp. 601–618, 2016.

[54] N. Carlini, M. Jagielski, and I. Mironov, "Cryptanalytic extraction of neural network models," in *Annual Intrl. cryptology Conf.*, pp. 189–218, Springer, 2020.

[55] ARM, "Architecture reference manual," *ARMv7-A and ARMv7-R edition*, 2012.

[56] S. Rhoads, "Plasma-most mips i (tm) opcodes: Overview," *http://www. opencores. org/projects. cgi/web/mips/*, 2006.

[57] AlphaNOV, "S-LMS." [Online]https://www.alphanov.com/en/products-services/single-laser-fault-injection [Accessed: Dec.1, 2023], 2023.

[58] Xilinx, Inc., "v2021.1." [Online]https://www.xilinx.com/products/design-tools/vivado.html [Accessed May.15, 2024], 2021.

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

[60] E. Songhori, H. Siam, and S. Riazi, "Tinygarble framework." [Online]https://github.com/esonghori/TinyGarble [Accessed May.15, 2024], 2019.

[61] R. Stallman *et al.*, *Using and porting GNU CC*, vol. 675. Free Software Foundation, 1998.

[62] AMD, "v2021.1." [Online]https://www.xilinx.com/products/design-tools/vivado.html [Accessed May.23, 2024], 2023.

[63] L. Braun and W. Zakarias, R, "Tinylego framework." [Online]https://github.com/AarhusCrypto/TinyLEGO [Accessed May.15, 2024], 2019.

[64] Y. Lindell and B. Riva, "Cut-and-choose yao-based secure computation in the online/offline and batch settings," in *Advances in Cryptology–CRYPTO 2014: 34th Annual Cryptology Conf., Santa Barbara, CA, USA, August 17-21, 2014, Proceedings, Part II 34*, pp. 476–494, Springer, 2014.

[65] Intel, "Intel®64 and ia-32 architectures software developer's manual," *Volume 3A: System Programming Guide, Part*, 2024.

[66] Apple Computer, Inc. and Intrl. Business Machines Corporation and Motorola, Inc., *PowerPC Microprocessor Common Hardware Reference Platform: A System Architecture*. Morgan Kaufmann, 1995.

[67] R. P. Paul, *SPARC architecture, assembly language programming, and C*. Prentice Hall PTR, 1999.

[68] T. K. Frederiksen, T. P. Jakobsen, J. B. Nielsen, P. S. Nordholt, and C. Orlandi, "Minilego: Efficient secure two-party computation from general assumptions," in *Advances in Cryptology–EUROCRYPT 2013: 32nd Annual Intrl. Conf. on the Theory and Applications of Cryptographic Techniques, Athens, Greece, May 26-30, 2013. Proceedings 32*, pp. 537–556, Springer, 2013.