sciendo

Proceedings on Privacy Enhancing Technologies ; 2022 (1):501–521

Konstantinos Athanasiou*, Thomas Wahl, A. Adam Ding, and Yunsi Fei

Masking Feedforward Neural Networks Against Power Analysis Attacks

Abstract: Recent advances in machine learning have enabled Neural Network (NN) inference directly on constrained embedded devices. This local approach enhances the privacy of user data, as the inputs to the NN inference are not shared with third-party cloud providers over a communication network. At the same time, however, performing local NN inference on embedded devices opens up the possibility of Power Analysis attacks, which have recently been shown to be effective in recovering NN parameters, as well as their activations and structure. Knowledge of these NN characteristics constitutes a privacy threat, as it enables highly effective Membership Inference and Model Inversion attacks, which can recover information about the sensitive data that the NN model was trained on. In this paper we address the problem of securing sensitive NN inference parameters against Power Analysis attacks. Our approach employs *masking*, a countermeasure well-studied in the context of cryptographic algorithms. We design a set of *gadgets*, i.e., masked operations, tailored to NN inference. We prove our proposed gadgets secure against power attacks and show, both formally and experimentally, that they are composable, resulting in secure NN inference. We further propose optimizations that exploit intrinsic characteristics of NN inference to reduce the masking's runtime and randomness requirements. We empirically evaluate the performance of our constructions, showing them to incur a slowdown by a factor of about 2–5.

Keywords: side-channels; neural networks; masking

DOI 10.2478/popets-2022-0025
Received 2021-05-31; revised 2021-09-15; accepted 2021-09-16.

***Corresponding Author: Konstantinos Athanasiou:** Northeastern University, Boston, MA, Email: athanasiou.k@northeastern.edu
Thomas Wahl: Northeastern University, Boston, MA, Email: t.wahl@northeastern.edu
A. Adam Ding: Northeastern University, Boston, MA, Email: a.ding@northeastern.edu
Yunsi Fei: Northeastern University, Boston, MA, Email: yfei@ece.neu.edu

1 Introduction

Following ground-breaking advances in the field of machine learning, an increasing number of real-world applications, such as image and speech recognition [17, 19, 26, 48], perform classification and prediction tasks using Neural Network (NN) inference. NN inference has also found numerous applications in domains that operate on real-time data. Wearable sensors for health monitoring [36, 38], smart devices [34, 54] and autonomous vehicles [32, 56] are typical classes of embedded systems that collect data and perform a classification task in real-time.

In this paper we consider the scenario of NN inference algorithms running locally on an embedded device (rather than on a cloud computer), as employed by many popular smart applications and end-user health monitoring devices [34, 54]. This scenario is preferred in cases where data privacy is paramount (which may be compromised in the cloud either during transmission or by the cloud service itself), or where network connectivity is unreliable, e.g., in autonomous driving [32, 56].

Even in the scenario of local inference, however, the privacy of the directly or indirectly involved data is not fully guaranteed. By pushing trained NN models to end-user devices, the NN model providers may be inadvertently making the privacy-sensitive training data partially recoverable to skilled attackers, which is particularly concerning in situations that deal with medical records, health data and biometrics. *Membership Inference* [37, 44, 47] and *Model Inversion* [11, 12] attacks target the sensitive data that a NN model was trained on. When executed in a *white-box setting*, i.e. with detailed knowledge of the NN model (including specific values of weight parameters), both types of attacks have been shown to outperform attacks without such knowledge [11, 31], by reducing the false-positive rate during training data recovery.

Knowledge of some NN model's parameters is a natural requirement of white-box attacks. *Differential power analysis* (DPA) of the embedded device [23] can provide this knowledge, paving the way for stealing privacy-relevant training data. DPA is a form of side-channel attack in which an adversary recovers sensitive data involved in the computation of the device, by observing its power consumption. Recent work has demonstrated that DPA can be successfully launched against NN inference models to reverse-engineer the number of layers and neurons in each layer, the activations as well as the values of its various parameters [3, 10, 55].

Motivated by the above privacy concerns, we ask whether model providers can be better equipped to limit the leakage of private information from the NN models they deploy. Specifically, we address the problem of protecting the internal NN parameters—which, in the above spirit, we consider to be secrets—against DPA. We assume the attacker has physical access to the device executing NN inference (e.g., by purchasing a health monitoring device) and can therefore control its inputs, and observe its outputs and power consumption. Our proposed DPA protection mechanism is *masking* [5, 16], a form of secret sharing. We present a suite of masked operations fundamental in NN inference that are resistant against DPA. Masked computations are carried out in a finite ring and performed in a *shared* fashion, i.e., they may only operate on randomized shares of a secret, without reconstructing it.

Two characteristics of NN inference algorithms make their masking a challenging task. The first is related to data representation, as NN inference operates on numerical values which are usually represented by floating-point arithmetic. Masking of floating-point values is problematic due to significant precision losses during secret sharing and complex floating-point semantics. In this work we use fixed-point arithmetic for masking NN inference, as it represents numerical values using (signed or unsigned) bounded integers, which in turn are amenable to masking, with marginal losses in prediction accuracy [14, 27].

The second is related to masking of the required operations themselves, which include primitive operations on fixed-point numbers as well as operations unique to NN inference. Multiplication in fixed-point arithmetic requires a truncation operation, which must be masked as well. We perform masked truncation by adapting an approach originally proposed in the context of multiparty computation [35]. Among the non-linear operations, masking of the activations requires a masked sign operation, which we accomplish using the bitwise representation of signed integers and masked multiplication.

After obtaining a complete set of masked operations for NN inference, we turn our focus to computational and randomness requirements. To improve the efficiency of masked NN inference along both axes, we take into account its intrinsic characteristics, such as the chaining of layer operations, and propose efficient masked algorithms that are resistant against DPA. Consider the dot product operation (which takes up a bulk of the NN inference): instead of a straightforward substitution of primitive addition, multiplication and truncation by their masked counterparts, we fuse the masked primitives into a masked dot product that is more efficient and minimizes precision loss due to truncation. As for randomness requirements, given the large number of secret weight parameters (compared to the number of secret key values of cryptographic algorithms), we devise effective ways to reduce the demands for randomness that is required for the weight parameters to be secret-shared, as well as to perform masked operations on them.

In summary our contribution are:

1. We devise a library of masked operations fundamental in NN inference; we call each of them a *gadget*. We prove the security of our gadgets under the notion of 1-*strong-non-interference* (1-SNI) [2]. This notion guarantees that an attacker observing single points in the shared computation cannot learn any information on the secrets, *and* that 1-SNI gadgets can be securely composed into larger constructions.
2. We show how to compose our proposed gadgets into larger constructions, e.g., Multi-Layer Perceptron (MLP) inference models. We reason about their security both formally—using again the notion of 1-SNI—and experimentally, using Test Vector Leakage Assessment (TVLA), a standard methodology for the security evaluation of DPA targets.
3. We describe a tightening of the randomness requirements for our proposed inference models. Our approach vastly reduces the required random numbers, a scarce resource in embedded devices, from linear in the number of NN parameters to linear in the depth of the NN, without impacting security guarantees.
4. Finally, we provide secure implementations of MLP and Convolutional NN (CNN) inference in a publicly available C library, and experimentally evaluate the runtime penalty incurred by our proposed constructions, as well as the improvements of the tightening. We provide results for both x86 and ARM, across different NN architectures. We show that our masked inference models do not hinder the accuracy performance and incur about a 5-fold slowdown for MLP and 2–3-fold slowdown for CNN (including the cost of generating the required amount of random numbers), comparable to or lower than the slowdowns suffered by commonly encountered masked cryptographic software [42] (3- to 20-fold).

Masking of NN in the context of DPA has only been explored very recently [9, 10]. The prior work targets a special class of NN, namely Binarized NN (BNN) [39], which operate mostly on Boolean values, and implements a masked version of the latter on an FPGA. To our knowledge, our work is the first to support implementing arbitrary masked Feedforward NN.

2 Motivating Example

Side-channel attacks exploit physical characteristics of computational systems, such as timing [24], power consumption [23] and electromagnetic emanations [13], to recover confidential data of the computation. In their seminal work on Differential Power Analysis (DPA) [23], Kocher et al. show that an attacker able to observe power traces of a cryptographic implementation, i.e., variations of power consumption during the steps of the computation, can exploit the correlation between the values produced along the computation and the amount of power required to process them, in order to recover cryptographic secrets.

Threat Model The main objective of this work is to devise algorithms for NN inference that are resistant against Power Analysis. The pre-trained NNs are executed as bare-metal applications on a microprocessor. Given a classification input, the microprocessor performs an inference round using the NN model. The parameters constitute the secrets of the NN. Our threat model assumes a non-invasive, passive attacker [49] that is able to probe the power consumption of the processing unit. The attacker knows the algorithm executed on the microprocessor, but doesn't have access to the exact implementation. The attacker is non-invasive since they cannot depackage the processing unit in order to read secrets directly from the wires of the device. The attacker is passive as they observe the device's behavior during the processing without tampering with its functionality, e.g., they don't inject faults to the computation or provide malformed inputs. Finally, the attacker is bounded, i.e., there is a limit on the number of probes that monitor the device's power consumption. Attackers that violate these assumptions, e.g., by using an unlimited number of probes [25] fall outside the scope of this work.

Example Setting Human Activity Recognition (HAR) uses mobile sensors of real-time and embedded systems to enable a wide array of applications such as life-logging, healthcare, senior care etc. Deep Learning is a beneficial technology for HAR, as it improves the inference accuracy without burdening the underlying hardware [18, 29, 57]. A malicious user with access to a sensor-based device can launch DPA to reverse engineer the parameters of the trained network performing HAR.

We demonstrate DPA on a MLP with a single hidden layer of 512 neurons (common in HAR tasks [29]) that expects as input motion readings and classifies them into one of 14 possible activities. The attacker provides multiple inference inputs to the device, recording the input and the device's power consumption for the inference round. The power consumption of an inference round for input i is the *power trace* t_i; each power trace consists of the same number of samples. The power consumption of the device for the trace t_i at sample s is denoted by $t_i[s]$. As a result, the attacker has collected sets of inputs I and power traces T ($|I| = |T|$).

Having recorded each inference input, the attacker targets the multiplication operation between inputs and weights of the fully connected layer, and makes a guess w_g for the value of a specific weight. Using the guessed value and the set of inference inputs the attacker computes the set $V = \{v_i := HW(i \cdot w_g) | i \in I\}$, where HW returns the *Hamming weight* of a value. Then, the attacker computes Pearson's correlation ρ_{V,T_s}, where $T_s = \{t_i[s] | i \in I\}$ is the set of power consumption values at sample s. Figure 1 shows the values of ρ for each sample of the power traces, for a total of 10K power traces, for an MLP performing HAR. Around sample 11K we observe a correlation peak, i.e., a high linear relationship between v_i and the power consumption at sample 11K, which indicates that at this point of the computation the MLP performs a multiplication between an input and the value w_g. By performing multiple such correlation tests for different weight guesses, the attacker can eventually recover the weights of the MLP [3, 49]. In our work we aim to provide NN inference algorithms that eliminate any correlations between the values processed by the device and its power consumption.

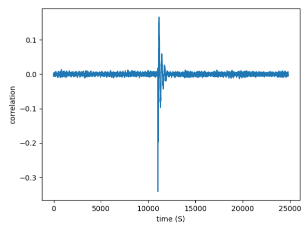

Fig. 1. Correlation between the power consumption of MLP inference and the result of multiplication of an input value with a secret weight.

3 Background

3.1 (Deep) Feedforward Neural Networks

NN are computing systems that, given some input, ultimately render a *decision*, e.g., assign to the input a particular classification label. In the context of classification, Neural networks (NN) can be thought of as mathematical functions that map an input vector to a vector of probabilities, each representing the likelihood for one of the classification outcomes.

Feedforward neural networks (FNN) are the quintessential deep learning model and form the basis of many NN applications. They are called *feedforward* because information flows through the computation without any feedback. They are called networks because they can be thought of as composition of multiple functions. Each function represents a *layer* of computation, and the composition of multiple functions forms a chain of layers. The length of this chain is the *depth* of the network. The last layer of the chain is the *output* layer. All other layers are referred to as the *hidden* layers.

The number of units in a layer, named *neurons*, defines the layer's *width*. Each neuron's output is defined by an expression of the form $Activation(\sum(weight \cdot input) + bias)$: the results of the preceding layer form the inputs of the neuron, which undergo a linear transformation—i.e., a weighted summation followed by a bias addition—, and a non-linear activation step.

An entire FNN architecture can be specified by its parameters along with the type of each layer. An architecture such that all the neurons in one layer are connected to the neurons in the next layer, i.e., it is Fully Connected (FC), and undergo a linear transformation followed by an activation function is known as a MLP. A MLP whose weight parameters are either -1 or 1 is a BNN. A FNN that uses convolutional layer and potentially pooling and FC layers is a CNN.

3.2 Fixed-Point Arithmetic

Fixed-point arithmetic is an approximation of real arithmetic that lends itself well to representation on computational platforms. Fixed-point numbers allocate a *fixed* amount of bits to represent their fractional part. An immediate caveat is that for a given word size ℓ, a fixed-point number can represent a smaller range of decimals compared to a floating-point number. The fixed number of bits of the fractional part brings, however, the advantage that operations over fixed-point numbers can be carried out by means of simpler circuits. As an example, addition in fixed-point arithmetic amounts to addition over integers that implement wrap-around semantics when overflows occur. This trade-off between precision/range and simplicity of data representation and operations has made fixed-point arithmetic a popular choice on embedded systems and more generally environments with constrained resources.

Fixed-Point numbers can be represented by signed or unsigned machine integers parameterized by two positive numbers ℓ and ℓ_d, where ℓ is the bit-length of the integer word, and $\ell_d < \ell$. A fixed-point number x can be now interpreted as a fraction whose numerator is the two's complement representation of the ℓ-bit integer x and whose denominator is 2^{ℓ_d}. That is, in fixed-point representation there is an implicit binary point between bits $\ell_d + 1$ and ℓ_d. A fixed-point number x can be decomposed into $x_1 \cdot 2^{\ell_d} + x_2$, where x_1 is a (signed) integer and $x_2 \in [0, 2^{\ell_d})$.

Fixed-point arithmetic comes in signed or unsigned forms, depending on whether the bits of the numerator are interpreted as signed or unsigned numbers. The difference in sign representation results in different representation ranges of a (ℓ, ℓ_d) fixed-point number, namely $\left[-\frac{2^{\ell-1}}{2^{\ell_d}}, \frac{2^{\ell-1}-1}{2^{\ell_d}}\right]$ for signed and $\left[0, \frac{2^{\ell}-1}{2^{\ell_d}}\right]$ for unsigned.

Addition over fixed-point numbers is straightforward: it amounts to addition over (signed) integers. Multiplication is more complicated as it results in a fixed-point number with $2 \cdot \ell_d$ bits representing the result's fractional part. The solution is an extra operation that *truncates* ℓ_d bits from the computed product. For the fixed-point number $x = x_1 \cdot 2^{\ell_d} + x_2$, we define the truncation operation as $\lfloor x \rfloor := x_1$, which is equivalent to a right-shift by ℓ_d bits of x.

3.3 Masking

Masking [5, 16], the most widely used countermeasure against DPA, is a form of secret sharing [46] in which a secret is split into *shares* that can be combined using a suitable function to recover the secret. Let $\mathbb{Z}_L$ be a finite ring of L elements. A $(t+1)$-*sharing* of an element $x \in \mathbb{Z}_L$ is a $(t+1)$-tuple $\langle x \rangle = \langle x^0, \ldots, x^t \rangle \in \mathbb{Z}_L^{t+1}$ such that $x^0 \star \ldots \star x^t = x$ and $x^i \in_\$ \mathbb{Z}_L$ for $i \in [0, t)$, where $\star \in \{+, \oplus\}$. Symbol $+$ denotes ring addition, $\oplus$ is bitwise XOR, and $\in_\$$ denotes uniform random selection of an element from a set. Intuitively, t of the shares are drawn at random, and the $(t+1)$-th share is computed from x and the rest of the shares. If $\star = +$, we speak of

$(t+1)$-*arithmetic sharing* and denote the $(t+1)$-tuple by $\langle x\rangle^L$. If $\star = \oplus$, we speak of $(t+1)$-*Boolean sharing* and denote the $(t+1)$-tuple by $\langle x\rangle^B$. Unless stated otherwise, we assume $(t+1)$-arithmetic sharings, or simply *sharings*, over $\mathbb{Z}_L$ and write $\langle x\rangle$. In rest of the paper we use masking and sharing interchangeably.

We call each element of a sharing $\langle x\rangle$ a *share* and denote the i-th share by $\langle x\rangle_i$. Given a sharing $\langle x\rangle$, we denote its set of shares by S_x. We use uppercase letters for matrices and vectors, and lowercase for scalars. Sharings of matrices and vectors are defined as pointwise sharings of their elements. For an $n\times m$ matrix $X \in \mathbb{Z}_L^{n\times m}$, $\langle X\rangle \in (\mathbb{Z}_L^{t+1})^{n\times m}$ denotes its $(t+1)$-shared counterpart; its (i,j)-th element is a $t+1$-tuple $\langle X_{ij}\rangle := \langle X^0_{i,j},\ldots,X^t_{i,j}\rangle$ such that $X^0_{i,j}+\ldots+X^t_{i,j} = X_{i,j}$.

4 Security Model

Our proposed algorithms use masking as the DPA countermeasure [5, 16]. Ishai et al. initiated the theoretical study of masking and introduced the first notion of security against side-channel attacks, namely t-probing security [20], stating that an adversary that can probe at most t intermediate results (i.e., values produced along a computation) of an algorithm cannot recover any information about the algorithm's secrets. Proofs of t-probing security rely on the notion of *perfect simulation* and amount to showing that any set of t intermediate results can be simulated by a proper subset of the input shares. From the definition of masking, an adversary can simulate any t of the $t+1$ shares of an input simply by generating random values. Showing that t intermediate results can be perfectly simulated by a proper subset of the input shares implies that the adversary cannot recover any information on the secrets (beyond the input shares that they generated themselves). A t-probing secure algorithm is also called t-order secure.

NN inference can be organized as a sequence of operations, each applied layer after layer, giving it a compositional structure. Motivated by this characteristic of NN inference, we provide a set of secure basic NN operations which can in turn be securely composed into full NN inference algorithms. The standard notion of t-probing security, however, does not guarantee that the composition of t-probing secure functions is also secure [8]. In this paper, we prove all our algorithms secure under the notion of *strong non-interference* (SNI) [2] that enables reasoning about security in a compositional fashion. Gadgets, i.e., fundamental units of a larger construction, whose inputs are $(t+1)$-sharings and that are shown to satisfy SNI, can be composed into algorithms that satisfy probing security.

Broadly, strong non-interference differs from the weaker notion of *non-interference* (NI) (which is a rephrased notion of t-probing security proposed after the latter) by distinguishing output shares from intermediate shares (for NI the output shares are also considered to be intermediate). When output shares are considered in isolation, their joint distribution must be *uniform*, i.e., they must be independent of the input shares. When considered in conjunction with intermediate shares, their joint distribution may only depend on as many input shares as the number of intermediate shares considered. These two properties allow SNI gadgets to be composed safely with other NI gadgets in a construction, as their output shares reveal no information on their input shares. We restate (and slightly reformulate) here the notions of NI and SNI, originally introduced by Barthe et al. [2].

Definition 1. *Let G be a gadget over n input sharings $\langle x_0\rangle,\ldots,\langle x_{n-1}\rangle$. G is t-**non-interferent**, t-**NI** for short, if, for every set V of at most t intermediate shares, there exist n sets $A_0,\ldots,A_{n-1}$ such that, for all i, $0\le i<n$: $A_i \subseteq S_{x_i}$, $|A_i|\le t$, and the shares in V can be perfectly simulated from $\langle A_0,\ldots,A_{n-1}\rangle$.*

Definition 2. *Let G be a gadget over n input sharings $\langle x_0\rangle,\ldots,\langle x_{n-1}\rangle$ and output sharing $\langle o\rangle$. G is t-**strong-non-interferent**, t-**SNI** for short, if, for every set $S\subseteq S_o$ of at most t output shares and every set V of $t' := t-|S|$ intermediate shares, there exist n sets $A_0,\ldots,A_{n-1}$ such that, for all i, $0\le i<n$: $A_i\subseteq S_{x_i}$, $|A_i|\le t'$, and the shares in V and S can be perfectly simulated from $\langle A_0,\ldots,A_{n-1}\rangle$.*

Example 3. *We illustrate the differences between the notions of NI and SNI. Algorithm 1 shows two gadgets that add two numbers $x,y\in\mathbb{Z}_L$ each given as a 2-sharing into the result sharing $\langle z\rangle$. It is easy to see that both gadgets are correct: recombining the sharing $\langle z\rangle$ by computing $\langle z\rangle_0+\langle z\rangle_1$ results in $x+y$ for both gadgets. The gadget on the left is 1-NI as each of the intermediate results depends on at most one of the shares of each input sharing $\langle x\rangle,\langle y\rangle$. More precisely, using Definition 1 with $n=2$ and $t=1$, for the intermediate share $\langle z\rangle_0$ we have that it can be simulated by the 2-tuple of sets $\langle A_0,A_1\rangle = \langle\{\langle x\rangle_0\},\{\langle y\rangle_0\}\rangle$. The same can be shown for $\langle z\rangle_1$ and therefore the gadget is 1-NI. The same gadget is not, however, 1-SNI: using Definition 2 with $n=2$*

and $t = 1$, for the output share set $S = \{\langle z \rangle_0\}$ we have $t' = 0$, hence $V = A_0 = A_1 = \emptyset$, but the share $\langle z \rangle_0 \in S$ cannot be simulated from $\langle \emptyset, \emptyset \rangle$.

In order to turn the shared addition algorithm into a 1-SNI gadget, an additional source of fresh randomness is required, denoted r in the gadget on the right. The intermediate shares $\langle w \rangle_0$ and $\langle w \rangle_1$ are defined using this random value and can thus be simulated from the empty set, which settles the case $S = \emptyset$ in Definition 2. For the case $S = \{\langle z \rangle_0\}$, we get as above: $t' = 0$, hence $V = A_0 = A_1 = \emptyset$. However, unlike with the gadget on the left, $\langle z \rangle_0$ can be simulated from the empty set: we have $\langle z \rangle_0 = (\langle x \rangle_0 - r) + \langle y \rangle_0$. Since r is uniformly sampled from $\mathbb{Z}_L$, this output share is uniform, too. The same can be shown for the case $S = \{\langle z \rangle_1\}$.

Algorithm 1 Two gadgets implementing shared addition: given 2-sharings $\langle x \rangle$ and $\langle y \rangle$, return 2-sharing $\langle z \rangle$ s.t. $z = x + y$. The gadget on the right receives an additional input, $r \in_\$ \mathbb{Z}_L$.

```
1: $\langle z \rangle_0 := \langle x \rangle_0 + \langle y \rangle_0$
2: $\langle z \rangle_1 := \langle x \rangle_1 + \langle y \rangle_1$
3: return $\langle z \rangle$
```

```
1: $\langle w \rangle_0 := \langle x \rangle_0 - r$
2: $\langle w \rangle_1 := \langle x \rangle_1 + r$
3: $\langle z \rangle_0 := \langle w \rangle_0 + \langle y \rangle_0$
4: $\langle z \rangle_1 := \langle w \rangle_1 + \langle y \rangle_1$
5: return $\langle z \rangle$
```

To see why NI is not composable, consider a call to the addition gadget on the left above, followed by a call to another gadget, name it G, with input sharings $\langle z \rangle$, $\langle x \rangle$. Under NI, an intermediate result in G that can be perfectly simulated from shares $\langle z \rangle_0$ and $\langle x \rangle_1$ does not reveal any information on neither x nor z. In this case however, due to the preceding addition gadget we have that $\langle z \rangle_0 = \langle x \rangle_0 + \langle y \rangle_0$, which implies that the ability to simulate $\langle z \rangle_0$ and $\langle x \rangle_1$ results in leakage of the value of x. In the rest of this paper we focus on devising gadgets that are 1-SNI and therefore lend themselves to implementing secure NN inference.

5 Masking Neural Networks

In this section we present our masking scheme for secure Neural Network (NN) inference. The main ideas behind our approach are to use a finite ring of integers to represent numeric values as fixed-point numbers, to arithmetically mask this representation, and finally to design operations that take masked representations of numeric values as input, and return them as output.

The masked NN hold their masked parameters. At each inference round, the masked NN must mask the unshared input and refresh the parameters' masks (but never recover them). In order to design operations over masked representations of fixed-point values, we begin with gadgets for basic operations (primitives) common in NN inference. We show masked implementations of these primitives and prove them to be 1-SNI. An intermediate result is the value stemming from a primitive operation in an algorithm's statement. In the case of multiple operations in a single statement, evaluation order is dictated first by parentheses, in their absence by the "multiplication-before-addition" rule, and is otherwise left-to-right. We provide some security proofs and lemma statements in the main paper and include the rest of the proofs in the Appendix, along with auxiliary shared algorithms. The correctness of our algorithms follows directly from recombining their output shares; the correctness of our constructions follows by composing correct gadgets. We then give examples of masking the computation of network *layers* by composing suitable masked gadgets for the basic operations. Crucially, we tap into the strength of SNI, which guarantees that such compositions preserve the security properties afforded by masking. Finally, we give an example full construction by implementing a MLP.

5.1 Representation

For our secure NN inference, we use a finite ring $(\mathbb{Z}_L, +, \cdot)$ of integers to represent signed fixed-point arithmetic. An integer $x \in \mathbb{Z}_L$ with ℓ_d bits allocated for its fractional part, where $\ell_d < \ell := \log_2 L$, represents the numeric value $\frac{tc(x)}{2^{\ell_d}}$ where $tc(x)$ interprets x in two's complement and returns its signed value. At the same time, x—being an element of $\mathbb{Z}_L$—readily lends itself to arithmetic masking, as described in Section 3.3, i.e., it can be split into shares $\langle x \rangle = \langle x^0, x^1 \rangle$, where $x^1 \in_\$ \mathbb{Z}_L$ and $x^0 = x - x^1$. The sharings of the gadgets that follow in the rest of this section represent signed fixed-point values.

5.2 Basic Operations

Dot Product These operations are fundamental in the linear components of FNN. In MLP, each neuron computes a dot product between inputs and weight parameters. In CNN, each feature is extracted by convolving part of the input with a kernel, i.e., by performing a dot product between the two. Algorithm 2 describes a masked dot product gadget. It uses sharing $\langle c \rangle$ as an accumulator and performs point-wise secure multiplication between the elements of its two input vectors.

Algorithm 2 *DotProd*$(\langle A \rangle, \langle B \rangle, r)$: shared dot product

```
Input: 2-sharing vectors $\langle A \rangle, \langle B \rangle \in (\mathbb{Z}_L^2)^n$, $r \in_\$ \mathbb{Z}_L$.
Output: 2-sharing $\langle c \rangle$ s.t. $c = A \cdot B$.
1: $\langle c \rangle_0 := -r$
2: $\langle c \rangle_1 := r$
3: for $k$ from 0 to $n-1$ do
4:     $\langle c \rangle_0 := \langle c \rangle_0 + \langle A_k \rangle_0 \langle B_k \rangle_1 + \langle A_k \rangle_1 \langle B_k \rangle_0$
5:     $\langle c \rangle_1 := \langle c \rangle_1 + \langle A_k \rangle_0 \langle B_k \rangle_0 + \langle A_k \rangle_1 \langle B_k \rangle_1$
6: return $\langle c \rangle$
```

Algorithm 2 is similar to the secure multiplication gadget between two masked secrets [42]. It uses a single fresh random number to mask the accumulator variable. It is crucial that the accumulator is masked first, before proceeding with the point-wise masked multiplications. Alternatively, each masked multiplication can use a fresh random number, but this would require as many random numbers as the size of the vectors. The correctness of the algorithm follows by expanding $\langle c \rangle_0 + \langle c \rangle_1$ and arriving to c. We show below that the optimization of using a single random number does not compromise security:

Theorem 4. *The DotProd gadget (Alg. 2) is 1-SNI.*

Proof. We use Definition 2 with $t = 1$ and distinguish two cases. In the first, we consider the output shares, i.e., $S = \{\langle c \rangle_0\}$ or $S = \{\langle c \rangle_1\}$, hence $t' = 0$ and $V = A_0 = A_1 = \emptyset$. We have:

$$\begin{aligned} \langle c \rangle_0 = \quad & -r \quad + \langle A_0 \rangle_0 \langle B_0 \rangle_1 + \langle A_0 \rangle_1 \langle B_0 \rangle_0 + \dots \\ & \qquad + \langle A_k \rangle_0 \langle B_k \rangle_1 + \langle A_k \rangle_1 \langle B_k \rangle_0 \\ \langle c \rangle_1 = \quad & r \quad + \langle A_0 \rangle_0 \langle B_0 \rangle_0 + \langle A_0 \rangle_1 \langle B_0 \rangle_1 + \dots \\ & \qquad + \langle A_k \rangle_0 \langle B_k \rangle_0 + \langle A_k \rangle_1 \langle B_k \rangle_1 \end{aligned}$$

Due to the random value r, both $\langle c \rangle_0$ and $\langle c \rangle_1$ are uniform and thus can be perfectly simulated by the empty sets of shares $\langle A_0, A_1 \rangle$.

In the second case we consider the intermediate shares, i.e., $S = \emptyset$, hence $t' = 1$. We show in Table 1 all single intermediate shares forming set V (left), and the tuples of sets of input shares with cardinality at most 1 that can perfectly simulate them (right). □

Intermediate shares	Sets of input shares
$-r$	$\emptyset$
r	$\emptyset$
$\langle A_k \rangle_0 \langle B_k \rangle_0$	$\{\langle A_k \rangle_0\}, \{\langle B_k \rangle_0\}$
$\langle A_k \rangle_0 \langle B_k \rangle_1$	$\{\langle A_k \rangle_0\}, \{\langle B_k \rangle_1\}$
$\langle A_k \rangle_1 \langle B_k \rangle_0$	$\{\langle A_k \rangle_1\}, \{\langle B_k \rangle_0\}$
$\langle A_k \rangle_1 \langle B_k \rangle_1$	$\{\langle A_k \rangle_1\}, \{\langle B_k \rangle_1\}$
$r + \ldots + \langle A_k \rangle_0 \langle B_k \rangle_0$	$\emptyset$
$r + \ldots + \langle A_k \rangle_0 \langle B_k \rangle_0 + \langle A_k \rangle_1 \langle B_k \rangle_1$	$\emptyset$
$r + \ldots + \langle A_k \rangle_0 \langle B_k \rangle_1$	$\emptyset$
$r + \ldots + \langle A_k \rangle_0 \langle B_k \rangle_1 + \langle A_k \rangle_1 \langle B_k \rangle_0$	$\emptyset$

Table 1. Tuples of sets of input shares that can perfectly simulate the intermediate shares. Instead of presenting $2 * k$-tuples of sets of shares, we show the sets that carry information. If all $2 * k$ sets are empty we write $\emptyset$.

Truncation The arithmetic 2-sharings we use for masked NN inference represent signed integers, which in turn when interpreted as fixed-point integers represent decimal numbers. As explained in Section 3.2, one of the benefits of using the fixed-point representation of decimals is that decimal addition and multiplication can be immediately performed in fixed-point by means of signed integer addition and multiplication. In the case of multiplication, an extra *truncation* step is, however, required in order to keep the number of bits that represent the fraction part constant. More precisely, the result of each signed integer multiplication must be followed by the truncation of the result's ℓ_d least significant bits. In the context of multi-party computation of machine learning models, Mohassel et al. [35] show a probabilistically approximately correct way to truncate an arithmetic 2-sharing. Their approach is easy and efficient to implement, and it incurs a small error on the least significant bit of the reconstructed result, with high probability. We recall here a theorem from the prior work:

Theorem 5 ([35]). *Let $\langle x \rangle$ be the 2-sharing of $x \in [0, 2^{\ell_t}] \cup [2^\ell - 2^{\ell_t}, 2^\ell)$ where $\ell_t < \ell - 1$. If $\langle \lfloor x \rfloor \rangle_0 = \lfloor \langle x \rangle_0 \rfloor$ and $\langle \lfloor x \rfloor \rangle_1 = 2^\ell - \lfloor 2^\ell - \langle x \rangle_1 \rfloor$ then $\langle \lfloor x \rfloor \rangle_0 + \langle \lfloor x \rfloor \rangle_1 \in \{\lfloor x \rfloor - 1, \lfloor x \rfloor, \lfloor x \rfloor + 1\}$ with probability $1 - 2^{\ell_t + 1 - \ell}$.*

Broadly, Theorem 5 states that, under some assumptions and with high probability, the truncated individual shares denoted by $\langle \lfloor x \rfloor \rangle_0$ and $\langle \lfloor x \rfloor \rangle_1$ may have an off-by-one error from the un-shared truncated value $\lfloor x \rfloor$ when recombined. That is, in contrast to all other shared algorithms in this paper, the truncation operation may incur a precision loss. Moreover, if the assumptions of Theorem 5 are not satisfied, the truncation operation may result in a faulty value altogether. Specifically, for the shared truncation to return a value off-by-one from its unshared counterpart, the unshared value x must fall into a specific region of $\mathbb{Z}_L$, parameterized by the value ℓ_t. Additionally the random number r used in the sharing $\langle x \rangle$ must be in the range $[2^{\ell_t}, 2^{\ell-\ell_t})$, therefore the probability of a correct result increases exponentially with the value of ℓ_t. We discuss the choice of appropriate values for ℓ_t further in Section 8.

Algorithm 3 shows our rendering of Theorem 5 as a 1-SNI algorithm. The main difference between the two is the additional fresh randomness at the end of Algorithm 3, which guarantees 1-SNI.

Algorithm 3 *Trunc*$(\langle x \rangle, r)$: shared truncation

```
Input: 2-sharing $\langle x \rangle$, $r \in_\$ \mathbb{Z}_L$
Output: 2-sharing $\langle z \rangle$ s.t. $z \in \{\lfloor x \rfloor -1, \lfloor x \rfloor, \lfloor x \rfloor +1\}$ with probability $(1 - 2^{l_x+1-l})$
1: $\langle y \rangle_0 := \langle x \rangle_0 \gg \ell_d$          ▷ ($\gg$) denotes right-shift
2: $\langle y \rangle_1 := 2^\ell - \left((2^\ell - \langle x \rangle_1) \gg \ell_d\right)$
3: $\langle z \rangle_0 := \langle y \rangle_0 + r$
4: $\langle z \rangle_1 := \langle y \rangle_1 - r$
5: return $\langle z \rangle$
```

Activation These functions constitute the non-linear component of NN inference. In the context of this work we consider the rectifier function defined as $ReLU(x) = max(0, x)$. A straightforward way to compute $ReLU(x)$ in fixed-point arithmetic is via its derivative $ReLU' \colon \mathbb{Z} \to \{0,1\}$: $ReLU(x) = ReLU'(x) \cdot x$ where $ReLU'(x) = 1$ iff $x > 0$.

Computing $ReLU'(x)$ essentially amounts to checking the sign of x. The fixed-point representation that we consider in this work uses signed integers, which are in turn represented in two's complement. The sign of the number therefore equals its most significant bit (MSB). However, the *shared* representation of the MSB is not simply given by the MSB of the two shares, as $\langle x \rangle_0[\ell-1] + \langle x \rangle_1[\ell-1] \neq x[\ell-1]$. In our approach we use known transformation functions between arithmetic and Boolean sharing. We transform the arithmetic 2-sharing $\langle x \rangle$ to its corresponding Boolean 2-sharing $\langle x \rangle^B$, which satisfies $\langle x \rangle_0^B \oplus \langle x \rangle_1^B = \langle x \rangle^B$. We then compute the MSB of x by bit extraction.

Algorithm 4 shows how to compute $ReLU'$. First it calls the *ArithToBoolean* gadget, to transform the sharing from arithmetic to Boolean. It then extracts the MSB of the Boolean shares. If their XOR is zero then—according to two's complement representation—x is positive, so $ReLU'$ must return 1, otherwise it must return 0. Hence, we negate one of the extracted bits (Line 3, via $\oplus 1$). Finally, to return a value that is arithmetically shared, we transform the Boolean 2-sharing of the extracted bit to an arithmetic sharing.

Algorithm 4 $ReLU'(\langle x \rangle, R)$: shared derivative of $ReLU$

```
Input: 2-sharing $\langle x \rangle$, $R \in_\$ \mathbb{Z}_L^4$.
Output: 2-sharing $\langle y \rangle$ such that $y = ReLU'(x)$
1: $\langle z \rangle^B := ArithToBoolean(\langle x \rangle, R_0, R_1)$
2: $\langle w \rangle_0^B := \langle z \rangle_0^B[\ell - 1]$
3: $\langle w \rangle_1^B := \langle z \rangle_1^B[\ell - 1] \oplus 1$
4: $\langle y \rangle := BooleanToArith(\langle w \rangle^B, R_2, R_3)$
5: return $\langle y \rangle$
```

ArithToBoolean and *BooleanToArith* were originally introduced by Goubin [15] and shown to be 1-NI. They have since been optimized and extended to higher-orders of protection [7] as well as shown to be 1-SNI [6]. The conversion gadgets require 2 fresh random numbers each, in order to guarantee 1-SNI, so Algorithm 4 expects an array R consisting of 4 fresh random numbers.

$ReLU'$ is the first non-primitive gadget so far, constructed by composition of other 1-SNI gadgets. We provide a lemma useful in proving non-primitive gadgets.

Lemma 6. *Let G be a **1-SNI** gadget over input sharings $\langle x_0 \rangle, \ldots, \langle x_{n-1} \rangle$, output sharing $\langle o \rangle$. Each output share of G can be perfectly simulated from the empty set.*

Theorem 7. *The $ReLU'$ gadget (Alg. 5) is 1-SNI.*

The final step toward computing the $ReLU$ activation is to multiply the result of $ReLU'(x)$ with x. Algorithm 5 implements the shared $ReLU$ activation, using the *Mul* gadget to multiply two sharings. *Mul* is well studied in the side-channel literature [20, 42] and has been shown to be SNI. While Algorithm 5 performs a multiplication, notice that it is not followed by a truncation operation. This is due to the first operand of the multiplication being the result of $ReLU'(x)$ which is either 0 or 1. This implies that the final multiplication result will be either 0 or x, neither of which requires truncating excess fractional bits. Algorithm 5 expects a vector of 5 fresh random numbers as parts of its inputs: 4 dedicated to the shared $ReLU'$ computation, and 1 to *Mul*.

Algorithm 5 *ReLU*$(\langle x\rangle, R)$: shared rectifier function

```
Input: 2-sharing $\langle x\rangle$, $R \in_\$ \mathbb{Z}_L^5$.
Output: 2-sharing $\langle y\rangle$ such that $y = ReLU(x)$
1: $\langle z\rangle := ReLU'(\langle x\rangle, R_{0:4})$
2: $\langle y\rangle := Mul(\langle z\rangle, \langle x\rangle, R_4)$
3: return $\langle y\rangle$
```

Max Pool Pooling components appear in CNN after their linear and non-linear components and are meant to provide a summary statistic of nearby elements of a layer. A commonly used statistic is that of the maximum element, i.e., *MaxPool*. To compute *MaxPool* over n shared elements, we iteratively compute the maximum of two values: the i-th element and the maximum of the $i-1$ previous elements and return the result of the $(n-1)$st maximum operation.

The above scheme allows us to focus on the problem of computing the maximum of two values, $\max(x, y)$. We observe that *ReLU* is a special case of max with $y = 0$ and use the same idea of extracting the sign of a value, only this time of the *difference* $x - y$. We do this via the comparison function $Cmp(x, y) := ReLU'(x-y)$. The result of *Cmp*, i.e., the sign bit of $x - y$, along with its negation can now be used to compute the maximum as $\max(x, y) = Cmp(x, y) \cdot x + (Cmp(x, y) \oplus 1) \cdot y$.

Algorithm 6 implements shared *MaxPool*. At iteration i, the shared $Cmp(\langle X_{i+1}\rangle, \langle y_i\rangle)$ gadget returns a tuple of sign sharings $\langle s_i\rangle, \langle s_i'\rangle$ where $s_i = ReLU'(X_{i+1} - y_i)$ and $s_i' = ReLU'(X_{i+1} - y_i) \oplus 1$. The sign sharings are multiplied with the current shared element $\langle X_{i+1}\rangle$ and the previous maximum $\langle y_i\rangle$, resp., and their results are added to compute the maximum of the two.

Algorithm 6 *MaxPool*$(\langle X\rangle, R)$: shared maxpool

```
Input: 2-sharing vector $\langle X\rangle \in (\mathbb{Z}_L^2)^n$, $R \in \mathbb{Z}_L^{9n}$
Output: 2-sharing $\langle y\rangle$ such that $y = \max_{0\le i<n} X_i$
1: $\langle y_0\rangle := \langle X_0\rangle$
2: for $i$ from 0 to $n-1$ do
3:     $\langle s_i\rangle, \langle s_i'\rangle := Cmp(\langle X_{i+1}\rangle, \langle y_i\rangle, R_{9i:9i+6})$
4:     $\langle w_i\rangle := Mul(\langle s_i\rangle, \langle X_{i+1}\rangle, R_{9i+6})$
5:     $\langle z_i\rangle := Mul(\langle s_i'\rangle, \langle y_i\rangle, R_{9i+7})$
6:     $\langle y_i\rangle := Add(\langle w_i\rangle, \langle z_i\rangle, R_{9i+8})$
7: return $\langle y_{n-1}\rangle$
```

As with *ReLU*, the two shared multiplications do not need to be followed by a shared truncation, as one of their operands is always a sign sharing, representing either 0 or 1. The *Cmp* gadget (shown in the Appendix) closely follows the shared *ReLU'* gadget: it prepends Algorithm 4 with a shared subtraction operation and appends it with an additional call to *BooleanToArith* for the complementary sign sharing. Due to these two additional operations, shared *Cmp* requires 6 fresh random numbers. Shared *MaxPool* requires 9 random numbers per binary max computation, for a total of $9(n-1)$ for a *MaxPool* of n elements.

Theorem 8 summarizes the security proofs for the remaining gadgets of basic operations.

Theorem 8. *The Trunc, ReLU, Add, Mul, Cmp, MaxPool gadgets are 1-SNI.*

5.3 Layer Operations and Full Constructions

We demonstrate how the gadgets for basic operations defined in the previous section can be composed into *layer operations*, and provide a full construction of a shared MLP. We compose the *DotProd*, *Trunc*, *Add* gadgets into the *Linear* gadget of Algorithm 7, which computes the linear component of a FC connected layer. To avoid unnecessary loss of precision we perform truncation once per neuron, after the dot product and before bias addition. The *Linear* gadget requires a number of fresh randoms linear in the layer's width m. A convolutional gadget can be constructed (and shown to be 1-SNI) using the same gadgets and invoking them in the same order (see Appendix). The difference lies in the selection of inputs that are multiplied with the convolution kernel in the *DotProd* gadget.

Algorithm 7 *Linear*$(\langle I\rangle, \langle W\rangle, \langle B\rangle, R)$: shared FC linear component

```
Input: Input vector $\langle I\rangle \in (\mathbb{Z}_L^2)^n$, weight matrix $\langle W\rangle \in (\mathbb{Z}_L^2)^{m\times n}$, bias vector $\langle B\rangle \in (\mathbb{Z}_L^2)^m$, $R \in_\$ \mathbb{Z}_L^{3m}$.
Output: $\langle Z\rangle$ s.t. $Z = I \times W + B$
1: for $i$ from 0 to $m-1$ do
2:     $\langle X_i\rangle := DotProd(\langle I\rangle, \langle W_{i,:}\rangle, R_{3i})$
3:     $\langle Y_i\rangle := Trunc(\langle X_i\rangle, R_{3i+1})$
4:     $\langle Z_i\rangle := Add(\langle Y_i\rangle, \langle B_i\rangle, R_{3i+2})$
5: return $\langle Z\rangle$
```

The *Activation* gadget, shown in Algorithm 8, is straightforward and computes the non-linear component of a layer by applying the *ReLU* gadget to all neurons of a layer. As each call to *ReLU* requires 5 fresh random numbers the total cost of the activation layer in terms of fresh randoms is $5m$ where m the layer's width.

Algorithm 8 *Activation*$(\langle X\rangle, R)$: shared Activation Component

```
Input: 2-sharing vector $\langle X\rangle \in (\mathbb{Z}_L^2)^m$, $R \in \mathbb{Z}_L^{5m}$
Output: 2-sharing vector $\langle Z\rangle \in (\mathbb{Z}_L^2)^m$ s.t. $Z_i = ReLU(X_i)$, $0 \le i < m$
1: for $i$ from 0 to $m-1$ do
2:     $\langle Z_i\rangle := ReLU(\langle X_i\rangle, R_{5i:5(i+1)})$
3: return $\langle Z\rangle$
```

The layer operations are composed exclusively of 1-SNI gadgets. Thus, by Lemma 6, all their intermediate and output shares can be simulated by tuples containing empty sets of shares. The proof of security for the layer operations follows directly from this observation.

Theorem 9. *The Linear, Activation gadgets are 1-SNI.*

Having access to 1-SNI layer operations, we can compose multiple layers in a larger chain construction and form a shared MLP. In Algorithm 9 we show a shared MLP with n shared inputs, a single hidden layer of width m and an output layer of width k. It requires $(3+5)m + 3k$ fresh random numbers in total, i.e., linear in the size of the network's width. The precision loss that can be incurred in the network, when compared to its unmasked fixed-point counterpart, grows with the depth of the network, as each linear layer operation requires a truncation step. Similarly to the layer operations, the security of Algorithm 9 follows directly from the 1-SNI layer gadgets.

Algorithm 9 FNN: 2-shared MLP. (1 hidden layer)

```
Input: $\langle I\rangle \in (\mathbb{Z}_L^2)^n, \langle W1\rangle \in (\mathbb{Z}_L^2)^{m\times n}, \langle B1\rangle \in (\mathbb{Z}_L^2)^m, \langle W2\rangle \in (\mathbb{Z}_L^2)^{k\times m}, \langle B2\rangle \in (\mathbb{Z}_L^2)^k$: 2-shared inputs and parameters.
Output: 2-shared classification output vector $\langle Z\rangle \in (\mathbb{Z}_L^2)^k$.
1: $R1 \in_R \mathbb{Z}_L^{3m}$, $R2 \in_R \mathbb{Z}_L^{5m}$, $R3 \in_R \mathbb{Z}_L^{3k}$
2: $\langle X\rangle := Linear(\langle I\rangle, \langle W1\rangle, \langle B1\rangle, R1)$
3: $\langle Y\rangle := Activation(\langle X\rangle, R2)$
4: $\langle Z\rangle := Linear(\langle Y\rangle, \langle W2\rangle, \langle B2\rangle, R3)$
5: return $\langle Z\rangle$
```

6 Security Evaluation

In Section 5 we provided masked gadgets whose composition leads to masked FNN inference algorithms. We reasoned about their security in the probing model by showing the inference algorithms to be 1-SNI. This implies that an attacker probing 1 intermediate result of the masked computation cannot recover any information on the secret, i.e., the network's parameters. The probing model however, relies on some assumptions, e.g., that each intermediate result leaks independently [40]. In this section we investigate the leakage behavior of our proposed algorithm in practice, and experimentally validate its security.

Test Vector Leakage Assessment (TVLA) This is a leakage evaluation methodology [45] that uses Welch's t-test to asses the security for implementations of masked algorithms. TVLA requires the collection of two power measurement datasets: one where the inputs to the masked implementation are *fixed* to a specific value, and one where the input of each measurement is *random*. We then compute the so-called *fixed-vs-random* t-test values for the two datasets on all points (i.e., samples) of the measurement traces and test the hypothesis that the traces in the two datasets have the same *means* on all points. High t-test values indicate violation of the null hypothesis, i.e., that the traces have the same means on all points. By increasing the *degree of freedom* of the t-test, i.e., the number of measurement traces in the datasets, the confidence on accepting or rejecting the null hypothesis also increases. Usually a predefined threshold is set to reject the null hypothesis based on the t-test value. In common industry standards for TVLA, the t-value threshold is set to $|th| = 4.5$. When the t-values on all points of the traces are lower than the threshold, it confirms that there are no leakages on the traces. Attacking an implementation whose t-test values lie within $|th|$ is expected to have the same success rate as a random guess [43].

Experimental Setup We use a STM32F407IG 32-bit ARM Cortex-M4 CPU, clocked at 168Mhz and with 1MB of internal flash memory, to execute our masked implementations as bare-metal applications. We collect power measurements using a LeCroy oscilloscope with a sampling rate of $1GS/s$ (GS =giga-samples) in order to collect approximately 6 samples per clock cycle. The CPU is part of the Pinata Development board [41], which bypasses the on-board voltage regulator and provides less-noisy power signals.

Figure 2 shows a diagram of our collection setup. A control unit (a PC) generates the input sharings as well as any random sources required by the gadgets, and communicates them to the board. The board reads the parameters and sets a trigger signal, prior to and after executing the masked algorithm, and communicates the algorithm's result to the control. The oscilloscope captures the power measurement within the trigger window and communicates it to the control unit: we call each power measurement collected in this way a *trace*. For reproducibility purposes, we use an AES-based seeded pseudo-random number generator (PRNG) to generate the random numbers required for the input and parameter sharings and for the fresh random numbers of the gadget computation.

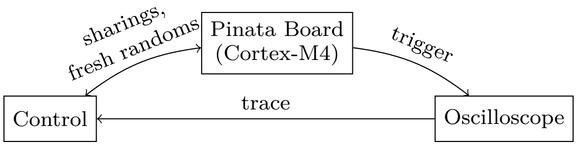

Fig. 2. Trace collection setup.

Evaluation We evaluate a masked C implementation of a MLP similar to the one presented in Algorithm 9, with an architecture of 2 inputs, a single hidden layer of 2 neurons, and an output layer of width 2. We chose this minimal architecture to keep the number of samples collected for each trace small, while still being able to exercise all the basic and layer operations described in Section 5. Following Algorithm 9, our implementation expects as input the input and parameter sharings ($\langle I \rangle, \langle W1 \rangle, \langle B1 \rangle$ etc.) as well as the fresh random numbers (R), and outputs a classification probability vector. The sharings and fresh randoms are generated offline (in the control unit). In our TVLA we use the *non-specific* t-test, which makes no assumptions on the leakage model of the board and targets all intermediate shares. During trace collection we interleave (at random) the traces of the fixed and random datasets to avoid any external influences or dependencies on the internal state of the board.

Figure 3 shows fixed vs random t-test results for the whole inference computation of the MLP described in the previous paragraph. We have performed three experiments. In the first, called *PRNG-Off*, instead of using the PRNG to supply the random values used in sharings and the fresh random numbers, we set these values to a constant. This is the baseline, as the lack of randomness in the sharings should result in high leakage. This conjecture is confirmed in Figure 3 (a): at 100K traces, we see clear deviations from set threshold $|th| = 4.5$, for samples throughout the trace, rejecting the null hypothesis and indicating the existence of power leakage.

In the second experiment, called *PRNG-On*, we use the seeded PRNG to supply the random values. The result is shown in Figure 3 (b): at 1 million traces, there is no indication of exploitable power leakage, as the t-test values fall within the desired threshold, for all sample points along the whole trace, validating that our masked gadgets do eliminate first-order power leakage. In the third experiment, we increase the order of the t-test in the TVLA. A second-order TVLA checks the hypothesis that two datasets have the same second statistical moments (i.e., no second-order leakage). Figure 3 (c) invalidates this hypothesis by showing values outside of th, which indicate that there are still second-order leakages in the traces. Since our implementations are 1-SNI, they do not have first-order leakage but do not protect against second-order leakage. With this experiment we validate the correctness of our measurement setup and the security protection of our gadgets.

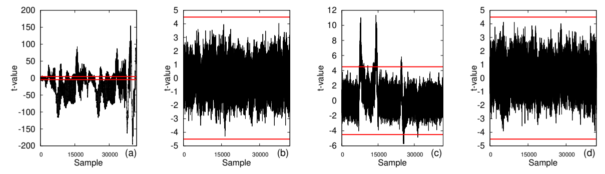

Fig. 3. T-test of full shared MLP inference. (a): PRNG-Off, (b): PRNG-On, (c): 2nd-order PRNG-On, (d): PRNG-On Tightened. (a): 100K Traces, (b-d): 1 mil. Traces.

7 Tightening Random Numbers

In Section 5.3 we gave an example of a shared MLP with n inputs, a single hidden layer of width m and an output layer of width k and found that, for an inference round, masking of the shared layer operations requires a number of randoms linear in the layer's width. In addition to masking operations, at each inference round the inputs as well as the weights of the MLP must be masked with fresh random numbers, increasing the randomness requirements to linear in the number of parameters of the MLP. Random numbers are an expensive resource, as their generation requires either hardware support or frequent invocation of PRNG routines.

In this section we describe our approach to tightening the number of required fresh randoms for masking of both the NN parameters and operations involved in a layer. The key observation that enables this tightening is that within a layer, each neuron's (in the case of MLP) or feature's (in the case of CNN) computation can be done in parallel. The main ideas behind our tightening approach are i) to re-use the same random numbers in the sharings of the parameters of a layer, and ii) to re-use the fresh random numbers in the gadgets across different neurons/features of the same layer. We show that, despite aggressively re-using random numbers, the gadgets of Section 5 can be made to satisfy 1-SNI.

We begin with tightening the number of randoms in a layer's parameter sharings and propose to use a single random number for all parameters of a layer. We focus on the *DotProd* gadget as both MLP and CNN compute their linear components first. Theorem 10 formalizes the idea behind using a single random number in sharings of parameters of a layer and a single random number in sharings of inputs. Its proof is similar to that of Theorem 4 after swapping the parameter (and input) shares for a single random number.

Theorem 10. *The DotProd gadget with $\langle A_0\rangle_1 = \ldots = \langle A_{n-1}\rangle_1$ and $\langle B_0\rangle_1 = \ldots = \langle B_{n-1}\rangle_1$ is 1-SNI.*

Proof. Let $a_r := \langle A_0\rangle_1 = l \cdots = \langle A_{n-1}\rangle_1$ and $b_r := \langle B_0\rangle_1 = \ldots = \langle B_{n-1}\rangle_1$. We use Definition 2 with $t = 1$ and distinguish two cases. In the first, we consider the output shares, i.e., $S = \{\langle c\rangle_0\}$ or $S = \{\langle c\rangle_1\}$, hence $t' = 0$ and $V = A_0 = A_1 = \emptyset$. We have:

$$\begin{aligned}\langle c\rangle_0 &= -r + \langle A_0\rangle_0 b_r + a_r\langle B_0\rangle_0 + \ldots \\ &\quad +\langle A_{n-1}\rangle_0 b_r + a_r\langle B_{n-1}\rangle_0 \\ \langle c\rangle_1 &= r + \langle A_0\rangle_0\langle B_0\rangle_0 + a_r b_r + \ldots \\ &\quad +\langle A_k\rangle_0\langle B_k\rangle_0 + a_r b_r\end{aligned}$$

Due to the random value r, both $\langle c\rangle_0$ and $\langle c\rangle_1$ are uniform and can be perfectly simulated by the empty sets of shares $\langle A_0, A_1\rangle$.

In the second case we consider the intermediate shares, i.e., $S = \emptyset$, hence $t' = 1$. We show in Table 1 all single intermediate shares forming set V (left), and the tuples of sets of input shares with cardinality at most 1 that can perfectly simulate them (right). ☐

Intermediate shares	Sets of input shares
$-r$	$\emptyset$
r	$\emptyset$
$\langle A_k\rangle_0\langle B_k\rangle_0$	$\{\langle A_k\rangle_0\}, \{\langle B_k\rangle_0\}$
$\langle A_k\rangle_0 b_r$	$\{\langle A_k\rangle_0\}, \{b_r\}$
$a_r\langle B_k\rangle_0$	$\{a_r\}, \{\langle B_k\rangle_0\}$
$a_r b_r$	$\{a_r\}, \{b_r\}$
$r + \ldots + \langle A_k\rangle_0\langle B_k\rangle_0$	$\emptyset$
$r + \ldots + \langle A_k\rangle_0\langle B_k\rangle_0 + a_r b_r$	$\emptyset$
$r + \ldots + \langle A_k\rangle_0 b_r$	$\emptyset$
$r + \ldots + \langle A_k\rangle_0 b_r + a_r\langle B_k\rangle_0$	$\emptyset$

Table 2. Sets of input shares that can perfectly simulate the intermediate shares.

We now turn to tightening the fresh random numbers required for sharing layer operations. We discuss the *Linear* gadget of MLP; the goal is to use the same fresh random values across its neurons. More precisely, the *DotProd*, *Add* and *Trunc* gadgets, which *Linear* is composed of, can use the same fresh random values at each neuron. The reason is that their computations are completely parallel. We can therefore rewrite Algorithm 7 so that it requires $R \in_\$ \mathbb{Z}_L^3$ instead of $R \in_\$ \mathbb{Z}_L^{3m}$ and re-uses the random values R_0, R_1 and R_2 at each invocation of *DotProd*, *Add* and *Trunc* respectively, across the m neurons. A similar type of reasoning can be applied to the *Activation* gadget. All invocations of *ReLU* at different neurons can use the same random sources. Therefore we can rewrite Algorithm 8 so that it requires $R \in_\$ \mathbb{Z}_L^5$ instead of $R \in_\$ \mathbb{Z}_L^{5m}$. Following the above observations, a theorem similar to Theorem 9 can be easily shown for the tightened version of the *Linear* and *Activation* gadgets.

We make two remarks. First, the bias parameter tightening does not compromise 1-SNI of the *Add* gadget as all the calls to *Add* are invoked in parallel, and the results of *DotProd* are masked by R_0. Second, the results of each neuron of a layer depend on the same fresh random numbers. This satisfies the condition of Theorem 10 that both its inputs (i.e., the results of the previous layer) and its parameters can use a single fresh random value each. We can therefore apply the tightening strategy in a chain of layers.

Table 3 compares the randomness requirements of the *original* gadgets described in Section 5 and their *tightened* versions described in this section, which re-use random numbers. The original layer gadgets (*Linear*, *Activation*) require a number of randoms linear in their parameter count. The tightened versions require a constant number of randoms, without compromising the 1-SNI security property of the gadgets. We have experimentally confirmed that the tightening has no discernible impact on the leakage behavior of our implementations. We have performed the same fixed-vs-random t-test experiment as in Section 6 with PRNG-on for the tightened implementation and for 1 million traces. Figure 3 (d) shows that, as in the original version, there is no indication of power leakage.

The tightened *Linear* and *Activation* gadgets can be readily used in CNN inference. The *MaxPool* gadget, however, is not amenable to our tightening approach. The reason is that the output of *MaxPool* depends on the intermediate shares it introduces (which hold the max value up to the i-th iteration). Using the same random values for these intermediate shares would cancel

	Parameter sharings	Neuron computation	Total
original	$m \cdot n + m$	$3m + 5m$	$m \cdot n + 9m$
tightened	1	$3 + 5$	9

Table 3. Randomness requirements for a layer of n inputs and m neurons, invoking the *Linear* and *Activation* gadgets.

	ADD	MUL	BIN	CMP
Dot Product	$m \cdot n$	$m \cdot n$	0	0
Bias Addition	m	0	0	0
Truncation	0	0	m	0
ReLU	0	m	0	m
Total	$m \cdot n + m$	$m \cdot n + m$	m	m
MaxPool	$n - 1$	$2(n - 1)$	0	$n - 1$

Table 4. Primitive operation counts of unshared FC layer with n inputs, m neurons (top), of Maxpool over n elements (bottom).

the masks and result in leakage. In practice, this is not problematic as *MaxPool* is commonly performed over 4 elements, so its randomness requirements do not depend on the number of CNN parameters. Multiple invocation of the *MaxPool* gadget can reuse the same random numbers, as their results are independent of each other.

8 Performance Evaluation

In this section we investigate the performance characteristics of our proposed gadgets and pursue the following questions: 1) *How much overhead does the shared NN inference incur, compared to its unshared counterpart?*, 2) *How much does the shared NN inference impact the model's accuracy*?

8.1 Operation Complexity

We describe first a simple computational cost model and use it to provide a complexity analysis for shared and unshared inference. The complexity model counts *primitive* operations of the same type; it ignores loads and stores from memory. It consist of 4 types, namely addition, multiplication, bitwise, and comparison operations. We apply the model at the layer level, and compare the operation counts per type, for the unshared and shared NN inference. We focus our analysis on FC layers and the *MaxPool* operation. The gadgets used in FC layers also appear in convolution layers, so their results transfer directly. Since our shared NN gadgets rely on fixed-point representation of the numeric values of the network, we use the fixed-point representation for unshared NN inference to perform a fair comparison.

Each cell of Table 4 shows how often each type of operation (shown in columns) occurs, for every unshared subcomponent (shown in rows) of. We have implemented the unshared ReLU as $(x > 0) * x$, to allow a fair comparison with the shared *ReLU* gadget. The truncation operation is implemented as a right-shift. Asymptotically, for a FC layer, the cost of inference is linear in the product of its number of inputs and its width.

	ADD	MUL	BIN
DotProd	$4m \cdot n$	$4m \cdot n$	0
Add	$4m$	0	0
Trunc	$4m$	0	$2m$
ReLU	$8m$	$36m$	$130m$
Total	$4m \cdot n + 16m$	$4m \cdot n + 36m$	$132m$
MaxPool	$4(n - 1)$	$42(n - 1)$	$252(n - 1)$

Table 5. Primitive operation counts of shared FC layer with n inputs, m neurons (top), of Maxpool over n elements (bottom).

The rows in Table 5 are now the shared gadgets we presented in Section 5. The operation counts per type can be inferred from the corresponding gadget. For example, the *Add* gadget is invoked m times and performs 4 primitive additions; therefore, its entry in column ADD has a count of $4 \cdot m$. Notice that there is no CMP operation, as we perform comparison by 0 by means of the *ReLU'* or *Cmp* gadgets. *ReLU* and *MaxPool* are the most complicated gadgets as they make use of the *Mul*, *BooleanToArith* and *ArithToBoolean* (the implementations of the conversion and comparison gadgets along with a detailed operation breakdown of all basic gadgets are shown in Appendices A, C. Asymptotically, the cost of shared NN inference is still linear in the layer size. However, the dominating factor $m \cdot n$ is multiplied by a constant factor of 4.

8.2 Experimental Evaluation

We evaluate the performance of our unshared, shared and tightened implementations of NN inference in terms of runtime and accuracy. We investigate whether the operation overhead incurred by the sharing matches the slowdown predicted by our simple complexity model in the previous section, and whether sharing influences the accuracy of NN models.

NN Models We have trained NN of different classes (MLP, CNN, BNN) in PyTorch on the MNIST dataset, extracted their floating-point parameters, and converted them to fixed-point parameters for shared inference. We choose a collection of NN ranging from standard networks for MNIST [30], to models that were used in prior work related to secure inference [10, 35, 53]. We scale up these networks by increasing the depth, width, and number of channels to further evaluate how our algorithms respond. The architectures we work with are common in the domain of HAR [18].

Inference Library We have implemented the shared algorithms for secure inference in a C library [1]. The library includes the unshared, fixed-point counterparts of the algorithms, which we use as a baseline. For each NN model we provide four inference implementations. The UNSHARED implementation is straightforward and implements $ReLU(x)$ as $(x > 0) \cdot x$ to match the behavior of the shared implementation. The shared implementations follow the algorithms of Section 5 and use a software PRNG (KECCAK-based [4]) to generate the random numbers that they require for sharing the inputs and parameters, as well as those for layer operations. We have implemented three shared versions. SHARED-PRNG-OFF doesn't generate random values but uses constants in their place. The intention of this version is twofold: it is meant to separate the cost of sharing from the cost of randomness generation, and it serves as a baseline for comparing implementations with different randomness requirements. SHARED-ORIGINAL uses a number of randoms linear in the size of each layer and in the number of the model's parameters. Finally, SHARED-TIGHTENED uses a constant number of random numbers for each layer as well as for the parameters of each layer, as described in Table 3 of Section 7.

We represent fixed-point numbers using signed 32-bit integers. Our shared implementations substitute each fixed-point value by a tuple of shared values, effectively doubling the memory requirements of their unshared counterparts. We defer a space optimized version of our library for future work.

We recall from Section 5.2 that the *Trunc* gadget's probability of correctness is parametric to the value $\ell_t < \ell = 32$, and discuss how to choose values for ℓ_t, ℓ_d. ℓ_t constrains the range of integers in $\mathbb{Z}_L$ that can be shared. In detail, only the integers in $[0, 2^{\ell_t}) \cup [2^{\ell-\ell_t}, 2^{\ell})$ can be used to represent plaintext data and random numbers must come from the range $[2^{\ell_t}, 2^{\ell-\ell_t})$. We perform a profiling step in which we vary the values of ℓ_t and $\ell_d < \ell_t$ and observe the accuracy performance of an UNSHARED MNIST MLP. In our experiments, the pair $(\ell_t, \ell_d) = (16, 6)$ maximized the accuracy. $\ell_t = 16$ gives a 99.99% chance for the *Trunc* gadget to return a wrong result and allows us to share fixed-point values in the range $[-\frac{2^{\ell_t}}{2^{\ell_d}}, \frac{2^{\ell_t}}{2^{\ell_d}}) = [-1025, 1025)$.

Accuracy Our goal is to evaluate the impact of sharing, and specifically the additional precision losses of shared truncation, in the accuracy performance of fixed-point NN inference. We use the accuracy % of UNSHARED as our baseline and report the differences in accuracy % with the baseline for the SHARED-ORIGINAL and SHARED-TIGHTENED implementations (columns Acc. δ in Table 6). We observe small differences in the range $[-0.33, 0.19]$ % (negative numbers indicate losses).

Neither the class nor the architecture of NN suggest trends for the accuracy δ. Tightening the randomness requirements has no significant impact on the accuracy δ as the tightened algorithms perform the same number of truncation operations, but on different random values. This is reflected in the small differences in the accuracy δ between SHARED-ORIGINAL and SHARED-TIGHTENED. Optimizing the accuracy of the unshared DNN by means of training, and translating the models from floating-point to fixed-point arithmetic constitute orthogonal concerns. We omit results on the accuracy of SHARED-PRNG-OFF since its main goal is to provide runtime insights.

Runtime We measure the runtime of the four implementations of each DNN in a desktop Intel-i7-4770@3.40GHz with 16GB RAM. Working with x86 allows us to evaluate larger networks so we use it to investigate how our algorithms scale. We measure runtime in cycles using CPU time-stamp counters and report them in Table 6. Figure 4 shows the slowdown of the shared implementations relative to the unshared version.

We note that in MLP models SHARED-PRNG-OFF is about 5.5 times slower than UNSHARED across all networks, which matches the complexity analysis of the previous section. SHARED-ORIGINAL incurs slowdowns larger than an order of magnitude in all MLP. SHARED-TIGHTENED reverses the "trend": compared to UNSHARED its slowdown is negligibly larger than that of SHARED-PRNG-OFF compared to UNSHARED. We also report in Figure 4 the slowdown of a MLP executed in ARM Cortex-M4, as embedded devices are the nominal target of power analysis attacks. Due to memory

1 https://gitlab.com/athanasiou.k/masked-nn-lib

DNN	Architecture	unshared		shared-prng-off	shared-original		shared-tightened	
		Acc. %	Cycles	Cycles	Acc. δ	Cycles	Acc. δ	Cycles
MLP	15	92.99	105,952	583,601	-0.01	3,551,167	0.0	596,193
MLP [30]	1000	96.11	7,041,711	37,949,619	0.03	223,276,433	0.01	38,605,268
MLP [35, 53]	128-128	96.6	1,059,416	5,727,786	0.17	33,646,512	0.19	5,833,809
MLP [30]	300-100	96.86	2,379,748	12,708,031	-0.02	75,081,421	0.01	12,934,383
MLP	128-128-128	96.34	1,206,417	6,635,506	0.0	38,644,214	0.0	6,742,266
MLP	256-256-256	97.11	3,000,600	16,270,602	-0.03	95,133,584	-0.06	16,628,439
BNN [10]	512	57.61	3,613,095	19,355,779	-0.11	114,252,312	-0.06	19,681,165
CNN [30, 53]	(6,5)-(16,5)-256-120-84	97.92	6,244,028	20,964,586	-0.03	50,164,490	-0.07	21,157,075
CNN	(16,5)-(32,5)-1568	98.15	58,757,262	137,914,596	-0.33	238,620,513	-0.28	138,337,964

Table 6. Accuracy and runtime evaluation of DNN models on MNIST. Architecture separates layers by dashes, denotes FC layers by their width, and convolutional layers by the tuple (c, k) where c the output channels, k the kernel dimension of the convolutional layer. All architectures have a FC output layer of width 10. Cycles are the average CPU cycles required per inference round over the 10K test images of MNIST. δ is the result of subtracting the accuracy % of the unshared from the shared implementation.

limitations, the MLP executed in ARM expects 250 inputs. The depth and width of hidden layers as well as the underlying execution platform have no impact on the MLP implementations' slowdown as it remains the same across MLP models and platforms. The BNN implementation's slowdown follow that of the MLP models, which is expected as our library handles BNN by setting their weight parameters as either -1 or 1, and otherwise treats them as regular MLP.

CNN models demonstrate different slowdown characteristics when compared to MLP, but also between different CNN architectures. Broadly in CNN, SHARED-PRNG-OFF shows smaller slowdown that in MLP. We attribute this to the better data locality between subsequent calls to the dot product operation of a convolutional layer. The slowdown of SHARED-ORIGINAL in CNN is again smaller when compared to its equivalent of MLP (an order of magnitude). CNN models have a vastly smaller number of parameters compared to MLP and require less random values to mask them in SHARED-ORIGINAL. SHARED-TIGHTENED in CNN has the same impact as in MLP, i.e., it makes the slowdown minimally larger than that of SHARED-PRNG-OFF. Finally, we see that between the two CNN models, the one with fewer FC layers (and total number of neurons) performs much faster, despite the fact that it's convolution layers have a much larger number of channels.

The above trends demonstrate that shared convolution layers perform invariably better than FC layers. We consider the reported slowdowns to be encouraging toward the general direction of application of masking in NN inference. The 2.3–5.6× slowdown of shared NN inference is lower than the slowdowns suffered by commonly encountered masked cryptographic algorithms, e.g., masked AES [42], in software implementations.

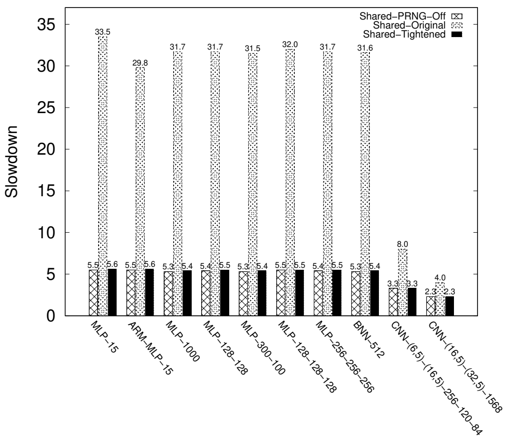

Fig. 4. Slowdown barcharts

9 Related Work and Discussion

DPA and Masking in NN Inference Batina et al. [3] were the first to successfully launch DPA against NN inference algorithms. They reverse engineer all the parameters as well as the NN structure for MLP and CNN inference, executed on AVR and ARM microcontrollers. Dubey et al. [10] performed DPA on an FPGA executing BNN inference and Yu et al. extended attacks against the same type of networks in the domain of electromagnetic emanations [55]. Dubey et al. [10] were the first to propose countermeasures for NN against DPA and used a combination of Boolean masking and hiding [50] to protect BNN inference in hardware against DPA.

The work closest to ours is BoMaNet [9] which describes a Boolean masked hardware implementation of BNN. Dubey et al. use masked lookup tables (LUT) to implement masked multiplication, and a masked implementation of ripple-carry adders for summations. ReLU activations are implemented by MSB extractions. The carry computation in the adder circuit requires masking of non linear AND gates, which they perform using the Trichina AND gate [51]. BoMaNet's masked circuits have been used to implement only MLP architectures and no formal guarantees of its masked circuits is provided. The constructions they provide are shown to experimentally satisfy 1-NI security, however in principle, we conjecture they could be refined to also satisfy 1-SNI.

BNN MLP inference utilizes the fact that parameters are binary values to perform space and time optimizations. As an example, a dot product computation over n elements can be carried out as an XNOR operation between two n-bit registers. The constructions in BoMaNet are specific to such BNN characteristics and do not extend to non-binarized NN that have parameters of arbitrary values. While BoMaNet's masked adder and MSB extraction for the ReLU computation could be extended to non-binarized NN, implementing masked multiplication by means of LUT is prohibited when dealing with arbitrary weight values. On the flipside, while our library can implement NN inference for any value representable in fixed-point arithmetic, including BNN (by simply representing weights as fixed-point 1 or -1), our gadgets are not optimized to exploit the binary weights (e.g., by performing dot product via a single XNOR). In terms of randomness requirements, BoMaNet does not exploit the layered structure of NN inference to reduce its randomness requirements. Each secret parameter requires a fresh random value to be masked and so does each non-linear operation. Finally, BoMaNet's secure circuits have been used to implement a 2-layer MLP and have not been applied to CNN.

BoMaNet uses additional flip-flops to partially account for known issues related to hardware glitches in CMOS circuitry [33]. Our work considers platform specific leakage characteristics, e.g., glitches or transition based leakage [1], an orthogonal question.

Secure Multi-Party Computation (MPC) A large body of work uses secret sharing techniques for NN and machine learning models in the context of MPC, both for training [28, 35, 53] and inference [21, 22, 52]. Masking and MPC target orthogonal security properties, e.g., a party in MPC might be the exclusive owner of a secret that must be protected against DPA. In masking there is a single party that holds all $t+1$ shares at all times, and an attacker observing at most t shares must learn no information about the secret. In MPC, there are at least two parties and a number of shares. An attacker, being either one of the parties or an external observer of the parties' communication, must at no point of hold all shares of the secret. Compared to masking, MPC incurs additional computational overhead due to communication and cryptographic assumptions which are not present in masking. Without including the communication overhead, prior work on secure inference [35, 53] reported total runtimes of 4.88 and 0.043 seconds per inference for a MLP with two hidden layers of 128 neurons. The cycles of our shared-tightened implementation (MLP 128-128 of Table 6) translate to runtime of 0.001753 seconds, i.e., at least an order of magnitude faster than the MPC based approaches.

10 Conclusion and Future Work

We have proposed a library of masked gadgets, resistant against DPA, that are the first to our knowledge that can be safely composed into full MLP and CNN constructions. We demonstrated the security of our construction formally and experimentally. We described how to reduce the random number requirements of our NN constructions, and showcased that they incur about a 2–5× slowdown to their unmasked counterparts, with minimal, if any, accuracy impairment. We set as future work to apply our library to more complex DNN targeting advanced datasets, as well as expanding to different architectures such as recurrent NN.

11 Acknowledgments

We would like to thank Alan Mislove for his advice on presenting our story, and Katerina Sotiraki for inspirational discussions on the broader security context of this work. This work was supported in part by the US National Science Foundation under grants SaTC-1929300, SaTC-1563697, and SHF-1718235.

References

[1] J. Balasch, B. Gierlichs, V. Grosso, O. Reparaz, and F.-X. Standaert. On the Cost of Lazy Engineering for Masked Software Implementations. In *International Conference on Smart Card Research and Advanced Applications*, pages 64–81. Springer, 2014.

[2] G. Barthe, S. Belaïd, F. Dupressoir, P. Fouque, B. Grégoire, P. Strub, and R. Zucchini. Strong Non-Interference and Type-Directed Higher-Order Masking. In E. R. Weippl, S. Katzenbeisser, C. Kruegel, A. C. Myers, and S. Halevi, editors, *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, pages 116–129. ACM, 2016.

[3] 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 Symposium (USENIX) Security 19)*, pages 515–532, 2019.

[4] G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche. KECCAK specifications. *Submission to nist (round 2)*, pages 320–337, 2009.

[5] S. Chari, C. S. Jutla, J. R. Rao, and P. Rohatgi. Towards Sound Approaches to Counteract Power-Analysis Attacks. In *Annual International Cryptology Conference*, pages 398–412. Springer, 1999.

[6] J.-S. Coron. High-Order Conversion from Boolean to Arithmetic Masking. In *International Conference on Cryptographic Hardware and Embedded Systems*, pages 93–114. Springer, 2017.

[7] J.-S. Coron, J. Großschädl, M. Tibouchi, and P. K. Vadnala. Conversion from Arithmetic to Boolean Masking with Logarithmic Complexity. In *International Workshop on Fast Software Encryption*, pages 130–149. Springer, 2015.

[8] J.-S. Coron, E. Prouff, M. Rivain, and T. Roche. Higher-Order Side Channel Security and Mask Refreshing. In *International Workshop on Fast Software Encryption*, pages 410–424. Springer, 2013.

[9] A. Dubey, R. Cammarota, and A. Aysu. BoMaNet: Boolean Masking of an Entire Neural Network. In *IEEE/ACM International Conference On Computer Aided Design, ICCAD 2020, San Diego, CA, USA, November 2-5, 2020*, pages 51:1–51:9. IEEE, 2020.

[10] A. Dubey, R. Cammarota, and A. Aysu. MaskedNet: The First Hardware Inference Engine Aiming Power Side-Channel Protection. In *2020 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 197–208, 2020.

[11] M. Fredrikson, S. Jha, and T. Ristenpart. Model Inversion Attacks that Exploit Confidence Information and Basic Countermeasures. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 1322–1333, 2015.

[12] M. Fredrikson, E. Lantz, S. Jha, S. M. Lin, D. Page, and T. Ristenpart. Privacy in Pharmacogenetics: An End-to-End Case Study of Personalized Warfarin Dosing. In K. Fu and J. Jung, editors, *Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014*, pages 17–32. USENIX Association, 2014.

[13] K. Gandolfi, C. Mourtel, and F. Olivier. Electromagnetic Analysis: Concrete Results. In *International workshop on cryptographic hardware and embedded systems*, pages 251–261. Springer, 2001.

[14] S. Gopinath, N. Ghanathe, V. Seshadri, and R. Sharma. Compiling KB-sized Machine Learning Models to Tiny IoT Devices. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 79–95, 2019.

[15] L. Goubin. A Sound Method for Switching between Boolean and Arithmetic Masking. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 3–15. Springer, 2001.

[16] L. Goubin and J. Patarin. DES and Differential Power Analysis—the "Duplication" Method. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 158–172. Springer, 1999.

[17] A. Graves, A.-r. Mohamed, and G. Hinton. Speech Recognition with Deep Recurrent Neural Networks. In *2013 IEEE international conference on acoustics, speech and signal processing*, pages 6645–6649. Ieee, 2013.

[18] N. Y. Hammerla, S. Halloran, and T. Plötz. Deep, Convolutional, and Recurrent Models for Human Activity Recognition using Wearables. In *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence*, IJCAI'16, page 1533–1540. AAAI Press, 2016.

[19] K. He, X. Zhang, S. Ren, and J. Sun. Deep Residual Learning for Image Recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[20] Y. Ishai, A. Sahai, and D. Wagner. Private Circuits: Securing Hardware against Probing Attacks. In *Annual International Cryptology Conference*, pages 463–481. Springer, 2003.

[21] C. Juvekar, V. Vaikuntanathan, and A. Chandrakasan. GAZELLE: A Low Latency Framework for Secure Neural Network Inference. In *27th USENIX Security Symposium USENIX Security 18)*, pages 1651–1669, 2018.

[22] Á. Kiss, M. Naderpour, J. Liu, N. Asokan, and T. Schneider. SoK: Modular and Efficient Private Decision Tree Evaluation. *Proceedings on Privacy Enhancing Technologies*, 2019(2):187–208, 2019.

[23] P. Kocher, J. Jaffe, and B. Jun. Differential Power Analysis. In *Annual International Cryptology Conference*, pages 388–397. Springer, 1999.

[24] P. C. Kocher. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In *Annual International Cryptology Conference*, pages 104–113. Springer, 1996.

[25] T. Krachenfels, F. Ganji, A. Moradi, S. Tajik, and J.-P. Seifert. Real-World Snapshots vs. Theory: Questioning the t-Probing Security Model. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 1955–1971. IEEE, 2021.

[26] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet Classification with Deep Convolutional Neural Networks. *Advances in neural information processing systems*, 25:1097–1105, 2012.

[27] A. Kumar, S. Goyal, and M. Varma. Resource-efficient Machine Learning in 2 KB RAM for the Internet of Things. In D. Precup and Y. W. Teh, editors, *Proceedings of the 34th*

International Conference on Machine Learning, volume 70 of *Proceedings of Machine Learning Research*, pages 1935–1944, International Convention Centre, Sydney, Australia, 06–11 Aug 2017. PMLR.

[28] N. Kumar, M. Rathee, N. Chandran, D. Gupta, A. Rastogi, and R. Sharma. CrypTFlow: Secure TensorFlow Inference. In *IEEE Symposium on Security and Privacy*. IEEE, May 2020.

[29] N. D. Lane and P. Georgiev. Can Deep Learning Revolutionize Mobile Sensing? In *Proceedings of the 16th International Workshop on Mobile Computing Systems and Applications*, pages 117–122, 2015.

[30] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-Based Learning Applied to Document Recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[31] K. Leino and M. Fredrikson. Stolen Memories: Leveraging Model Memorization for Calibrated White-Box Membership Inference. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 1605–1622, 2020.

[32] S. Liu, L. Liu, J. Tang, B. Yu, Y. Wang, and W. Shi. Edge Computing for Autonomous Driving: Opportunities and Challenges. *Proceedings of the IEEE*, 107(8):1697–1716, 2019.

[33] S. Mangard, N. Pramstaller, and E. Oswald. Successfully Attacking Masked AES Hardware Implementations. In J. R. Rao and B. Sunar, editors, *Cryptographic Hardware and Embedded Systems – CHES 2005*, pages 157–171, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[34] G. Manogaran, P. M. Shakeel, H. Fouad, Y. Nam, S. Baskar, N. Chilamkurti, and R. Sundarasekar. Wearable IoT Smart-Log Patch: An Edge Computing-Based Bayesian Deep Learning Network System for Multi Access Physical Monitoring System. *Sensors*, 19(13):3030, 2019.

[35] P. Mohassel and Y. Zhang. SecureML: A System for Scalable Privacy-Preserving Machine Learning. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 19–38. IEEE, 2017.

[36] S. C. Mukhopadhyay. Wearable Sensors for Human Activity Monitoring: A Review. *IEEE sensors journal*, 15(3):1321–1330, 2014.

[37] M. Nasr, R. Shokri, and A. Houmansadr. Comprehensive Privacy analysis of Deep Learning: Passive and Active White-box Inference Attacks against Centralized and Federated Learning. In *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*, pages 739–753. IEEE, 2019.

[38] A. Pantelopoulos and N. G. Bourbakis. A Survey on Wearable Sensor-Based Systems for Health Monitoring and Prognosis. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, 40(1):1–12, 2009.

[39] H. Qin, R. Gong, X. Liu, X. Bai, J. Song, and N. Sebe. Binary neural networks: A survey. *Pattern Recognit.*, 105:107281, 2020.

[40] M. Renauld, F. Standaert, N. Veyrat-Charvillon, D. Kamel, and D. Flandre. A Formal Study of Power Variability Issues and Side-Channel Attacks for Nanoscale Devices. In *Advances in Cryptology - EUROCRYPT 2011 - 30th Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 109–128, 2011.

[41] Riscure. Pinata Training Target. https://www.riscure.com/product/pinata-training-target/, 2020.

[42] M. Rivain and E. Prouff. Provably Secure Higher-Order Masking of AES. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 413–427. Springer, 2010.

[43] D. B. Roy, S. Bhasin, S. Guilley, A. Heuser, S. Patranabis, and D. Mukhopadhyay. Leak Me If You Can: Does TVLA Reveal Success Rate. Technical report, Cryptology ePrint Archive, Report 2016/1152, 2016., 2016.

[44] A. Salem, Y. Zhang, M. Humbert, P. Berrang, M. Fritz, and M. Backes. ML-Leaks: Model and Data Independent Membership Inference Attacks and defenses on machine learning models. In *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*. The Internet Society, 2019.

[45] T. Schneider and A. Moradi. Leakage Assessment Methodology. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 495–513. Springer, 2015.

[46] A. Shamir. How to Share a Secret. *Commun. ACM*, 22(11):612–613, 1979.

[47] R. Shokri, M. Stronati, C. Song, and V. Shmatikov. Membership Inference Attacks against Machine Learning Models. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 3–18. IEEE, 2017.

[48] K. Simonyan and A. Zisserman. Very Deep Convolutional Networks for Large-Scale Image Recognition. In Y. Bengio and Y. LeCun, editors, *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015.

[49] F.-X. Standaert. Introduction to Side-Channel Attacks. In *Secure integrated circuits and systems*, pages 27–42. Springer, 2010.

[50] K. Tiri and I. Verbauwhede. A Logic Level Design Methodology for a Secure DPA Resistant ASIC or FPGA Implementation. In *Proceedings Design, Automation and Test in Europe Conference and Exhibition*, volume 1, pages 246–251 Vol.1, 2004.

[51] E. Trichina, T. Korkishko, and K. H. Lee. Small Size, Low Power, Side Channel-Immune AES Coprocessor: Design and Synthesis Results. In H. Dobbertin, V. Rijmen, and A. Sowa, editors, *Advanced Encryption Standard – AES*, pages 113–127, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[52] A. Tueno, F. Kerschbaum, and S. Katzenbeisser. Private Evaluation of Decision Trees using Sublinear Cost. *Proceedings on Privacy Enhancing Technologies*, 2019(1):266–286, 2019.

[53] S. Wagh, D. Gupta, and N. Chandran. SecureNN: 3-Party Secure Computation for Neural Network Training. *Proceedings on Privacy Enhancing Technologies*, 2019(3):26–49, 2019.

[54] X. Wang, Y. Han, V. C. Leung, D. Niyato, X. Yan, and X. Chen. Convergence of Edge Computing and Deep Learning: A Comprehensive Survey. *IEEE Communications Surveys & Tutorials*, 22(2):869–904, 2020.

[55] H. Yu, H. Ma, K. Yang, Y. Zhao, and Y. Jin. DeepEM: Deep Neural Networks Model Recovery through EM Side-Channel Information Leakage. In *2020 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 209–218. IEEE, 2020.

[56] E. Yurtsever, J. Lambert, A. Carballo, and K. Takeda. A Survey of Autonomous Driving: Common Practices and Emerging Technologies. *IEEE Access*, 8:58443–58469, 2020.

[57] M. Zeng, L. T. Nguyen, B. Yu, O. J. Mengshoel, J. Zhu, P. Wu, and J. Zhang. Convolutional Neural Networks for Human Activity Recognition using Mobile Sensors. In *6th International Conference on Mobile Computing, Applications and Services*, pages 197–205. IEEE, 2014.

A Auxiliary Shared Algorithms

Algorithms 10 [6], 11 [6], 12 are used by the gadgets that appear in the main paper.

Algorithm 10 $\mathit{BooleanToArith}(\langle x\rangle, r_0, r_1)$

```
Input: 2-Boolean sharing $\langle x\rangle$, $r_0 \in_R \mathbb{Z}_L$, $r_1 \in_R \mathbb{Z}_L$
Output: 2-arithmetic sharing $\langle y\rangle$ s.t. $y = x$
 1: $\langle x\rangle_0 := \langle x\rangle_0 \oplus r_0$
 2: $\langle x\rangle_1 := \langle x\rangle_1 \oplus r_0$
 3: $t := \langle x\rangle_0 \oplus r_1$
 4: $t := t - r_1$
 5: $t := t \oplus \langle x\rangle_0$
 6: $r_1 := r_1 \oplus \langle x\rangle_1$
 7: $\langle y\rangle_0 := \langle x\rangle_0 \oplus r_1$
 8: $\langle y\rangle_0 := \langle y\rangle_0 - r_1$
 9: $\langle y\rangle_0 := \langle y\rangle_0 \oplus t$
10: $\langle y\rangle_1 := \langle x\rangle_1$
11: return $\langle y\rangle$
```

Algorithm 11 $\mathit{ArithToBoolean}(\langle x\rangle, r_0, r_1)$

```
Input: 2-arithmetic sharing $\langle x\rangle$, $r_0 \in_R \mathbb{Z}_L$, $r_1 \in_R \mathbb{Z}_L$
Output: 2-Boolean sharing $\langle y\rangle$ s.t. $y = x$
 1: $\langle x\rangle_0 := \langle x\rangle_0 \oplus r_0$
 2: $\langle x\rangle_1 := \langle x\rangle_1 \oplus r_0$
 3: $t := 2 \cdot r_1$
 4: $\langle y\rangle_0 := r_1 \oplus \langle x\rangle_1$
 5: $o := r_1 \wedge \langle y\rangle_0$
 6: $\langle y\rangle_0 := t \oplus \langle x\rangle_0$
 7: $r_1 := r_1 \oplus \langle y\rangle_0$
 8: $r_1 := r_1 \wedge \langle x\rangle_1$
 9: $o := o \oplus r_1$
10: $r_1 := t \wedge \langle x\rangle_0$
11: $o := o \oplus r_1$
12: for $k = 1$ to $\ell - 1$ do
13:     $r_1 := t \wedge \langle x\rangle_1$
14:     $r_1 := r_1 \oplus o$
15:     $t := t \wedge \langle x\rangle_0$
16:     $r_1 := r_1 \oplus t$
17:     $t := 2 \cdot r_1$
18: $\langle y\rangle_0 := \langle y\rangle_0 \oplus t$
19: $\langle y\rangle_1 := \langle x\rangle_1$
20: return $\langle y\rangle$
```

Algorithm 12 $\mathit{Cmp}(\langle x\rangle, \langle y\rangle, R)$: shared comparison

```
Input: 2-sharings $\langle x\rangle, \langle y\rangle$, $R \in \mathbb{Z}_L^6$.
Output: 2-sharings $\langle a\rangle, \langle b\rangle$ such that $a = \mathit{Cmp}(x, y)$, $b = \neg\mathit{Cmp}(x, y)$
 1: $\langle w\rangle_0 := \langle x\rangle_0 - \langle y\rangle_0$
 2: $\langle w\rangle_1 := \langle x\rangle_1 - \langle y\rangle_1$
 3: $\langle z\rangle^B := \mathit{ArithToBoolean}(\langle w\rangle, R_0, R_1)$
 4: $\langle v\rangle_0^B := \langle z\rangle_0[\ell - 1]$
 5: $\langle u\rangle_0^B := \langle z\rangle_0[\ell - 1]$
 6: $\langle v\rangle_1^B := \langle z\rangle_1[\ell - 1]$
 7: $\langle u\rangle_1^B := \langle z\rangle_1[\ell - 1] \oplus 1$
 8: $\langle a\rangle := \mathit{BooleanToArith}(\langle v\rangle^B, R_2, R_3)$
 9: $\langle b\rangle := \mathit{BooleanToArith}(\langle u\rangle^B, R_4, R_5)$
10: return $\langle a\rangle, \langle b\rangle$
```

Algorithm 13 shows how the *Linear* gadget (Alg. 7) can be adapted to compute shared convolution.

Algorithm 13 $\mathit{Conv2d}(\langle I\rangle, \langle W\rangle, \langle b\rangle, R)$: shared convolutional component

```
Input: Input $\langle I\rangle$, kernel $\langle K\rangle \in (\mathbb{Z}_L^2)^{2h+1\times 2w+1}$.
Output: $\langle Z\rangle \in (\mathbb{Z}_L^2)^{n\times m}$ s.t. $Z = I * K$
1: for $i$ from 0 to $n$ do
2:     for $j$ from 0 to $m$ do
3:         $\langle X_{i,j}\rangle := \mathit{DotProd}(\langle K\rangle, \langle I_{i-h:i+h,j-w:j+w}\rangle, R_{3(i+m\cdot j)})$
4:         $\langle Y_{i,j}\rangle := \mathit{Trunc}(\langle X_{i,j}\rangle, R_{3(i+m\cdot j)+1})$
5:         $\langle Z_{i,j}\rangle := \mathit{Add}(\langle Y_{i,j}\rangle, \langle b_i\rangle, R_{3(i+m\cdot j)+2})$
```

B Proofs of Lemmas and Theorems

We provide here the proofs for the lemmas and theorems stated in the main paper. We also provide proofs of security for the auxiliary algorithms of Appendix A. The numbers of the theorems and lemmas correspond to the numbers of theorems and lemmas in the main paper.

Lemma 6. *Let G be a **1-SNI** gadget over input sharings $\langle x_0\rangle, \ldots, \langle x_{n-1}\rangle$, output sharing $\langle o\rangle$. Each output share of G can be perfectly simulated from the empty set.*

Proof: From Definition 2 with $t = 1$, let $\langle o\rangle_i$ be the given output share ($i \in \{0, 1\}$) and $S_o = \{\langle o\rangle_i\}$. We know that the elements in S_o can be perfectly simulated from a tuple of sets $\langle A_0, \ldots, A_{n-1}\rangle$ with $|A_i| = t' = t - |S_o| = 0$, i.e., from the tuple of empty sets. □

Theorem 7. *The ReLU$'$ gadget (Alg. 5) is 1-SNI.*

Proof: We distinguish two cases. In the first, we consider the output shares, i.e., $S = \{\langle y\rangle_0\}$ or $S = \{\langle y\rangle_1\}$, hence $t' = 0$ and $V = A_0 = \emptyset$, where $\langle y\rangle_0 = \langle \mathit{BooleanToArith}(\langle w\rangle^B, R_2, R_3)\rangle_0$ and $\langle y\rangle_1 = \langle \mathit{BooleanToArith}(\langle w\rangle^B, R_2, R_3)\rangle_1$. From Lemma 6 we have that both $\langle y\rangle_0, \langle y\rangle_1$ can be simulated by $\langle A_0\rangle$. In the second case, we consider the intermediate shares, i.e., $S = \emptyset$, hence $t' = 1$. From Lemma 6 we have that both $\langle z\rangle_0, \langle z\rangle_1$ can be simulated by $\langle\emptyset\rangle$. Consequently, $\langle w\rangle_0, \langle w\rangle_1$ can also be simulated by $\langle\emptyset\rangle$ as they only depend on shares of $\langle z\rangle$. □

Lemma 11. *The Trunc gadget (Alg. 3) is 1-SNI.*

Proof:
Case 1: $|S_o| = 0$**:** Then $S_o = \emptyset$, and for some $i \in \{0,1\}$ $V = \{\langle y\rangle_i\}$ can be perfectly simulated by the tuple $\langle\{\langle x\rangle_i\}\rangle$ for some $i \in \{0,1\}$.
Case 2: $|S_o| > 0$ We have $S_o = \{\langle z\rangle_i\}$ for some $i \in \{0,1\}$ and hence $V = \emptyset$ and $\langle z\rangle_i$ can be perfectly simulated by the tuple containing the empty set $\langle\emptyset\rangle$.

□

Lemma 12. *The ReLU gadget (Alg. 5) is 1-SNI.*

Proof:From Lemma 6 and Theorem 7, we have that the intermediate shares $\langle z\rangle_0$ and $\langle z\rangle_1$ can be perfectly simulated by the empty set of shares, and from Lemma 6 and [20] , we have that the outputs shares $\langle y\rangle_0$ and $\langle y\rangle_1$ can be perfectly simulated by the empty set of shares. We distinguish two cases:
Case 1: $|S_o| = 0$**:** Then $S_o = \emptyset$, $V = \{z_i\}$ for some $i \in \{0,1\}$. We have shown above that $\langle z\rangle_i$ can be perfectly simulated from the empty set of shares.
Case 2: $|S_o| > 0$ We have $S_o = \{y_i\}$ for some $i \in \{0,1\}$ and hence $V = \emptyset$. We have shown above how $\langle y\rangle_i$ can be perfectly simulated from the empty set of shares.

□

Lemma 13. *The Add gadget (Alg. 1, right column) is 1-SNI.*

Proof:
Case 1: $|S_o| = 0$**:** Then $S_o = \emptyset$, and for some $i \in \{0,1\}$ $V = \{\langle w\rangle_i\}$ can be perfectly simulated by the tuple containing the empty set $\langle\emptyset\rangle$.
Case 2: $|S_o| > 0$ We have $S_o = \{\langle z\rangle_i\}$ for some $i \in \{0,1\}$ and hence $V = \emptyset$ and $\langle z\rangle_i$ can be perfectly simulated by the tuple containing the empty set $\langle\emptyset\rangle$.

□

Lemma 14 (Barthe et al. [2])**.** *The Mul gadget is 1-SNI.*

Lemma 15. *The Cmp gadget (Alg. 12) is 1-SNI.*

Proof: From Lemma 6 and Theorem 20, Theorem 19 we have that the intermediate shares $\langle z\rangle_i^B$, $\langle a\rangle_i$ and $\langle b\rangle_i$ ($i \in \{0,1\}$) can be perfectly simulated by the empty set of shares. The intermediate shares $\langle u\rangle_i^B$ and $\langle v\rangle_B$ depend only on $\langle z\rangle_i^B$ and can therefore also be perfectly simulated by the empty set of shares ($i \in \{0,1\}$). We distinguish two cases:
Case 1: $|S_o| = 0$**:** Then $S_o = \emptyset$. For $V = \{\langle u\rangle_i^B\}$, $V = \{\langle v\rangle_i\}$ for some $i \in \{0,1\}$, we have shown that $\langle v\rangle_i^B$ and $\langle u\rangle_i^B$ can be perfectly simulated from the empty set of shares. For $V = \{\langle w\rangle_i\}$ we have that the tuples of sets of shares $\langle\{\langle x\rangle_i\}, \{\langle y\rangle_i\}\rangle \subseteq S_x \times S_y$ for $i \in \{0,1\}$ can perfectly simulate $\langle w\rangle_i$.
Case 2: $|S_o| > 0$ For $i \in \{0,1\}$ we have $S_o = \{\langle a\rangle_i\}$ or $S_o = \{\langle b\rangle_i\}$ and hence $V = \emptyset$. We have shown above how $\langle a\rangle_i$, $\langle b\rangle_i$ can be perfectly simulated from the empty set of shares.

□

Lemma 16. *The MaxPool gadget (Alg. 6) is 1-SNI.*

Proof: From Lemma 6 and Lemma 14, Lemma 15 we have that the shares $\langle y_i\rangle$, $\langle s_i\rangle_i$, $\langle s_i'\rangle$, $\langle w_i\rangle$, ($i \in \{0,\ldots,n-1\}$) can be perfectly simulated by the empty set of shares. We distinguish two cases:
Case 1: $|S_o| = 0$**:** Then $S_o = \emptyset$. For $V = \{\langle y_i\rangle\}$ or $V = \{\langle y_i\rangle\}$ or $V = \{\langle s_i\rangle\}$ or $V = \{\langle s_i'\rangle\}$ or $V = \{\langle z_i\rangle\}$for some $i \in \{0,\ldots,n-2\}$, we have shown that they can be perfectly simulated from the empty set of shares.
Case 2: $|S_o| > 0$ For $i \in \{0,\ldots,m-1\}$ we have $S_o = \{\langle y_{n-1}\rangle\}$ and $S_o = \{\langle z_{n-1}\rangle\}$ and hence $V = \emptyset$. We have shown above how $\langle y_{n-1}\rangle$ and $\langle z_{n-1}\rangle$can be perfectly simulated from the empty set of shares.

□

Theorem 8. *The Trunc, ReLU, Add, Mul, Cmp, MaxPool gadgets are 1-SNI.*

Proof: Follows from Lemmas 11-16. □

Lemma 17. *The Linear gadget (Alg. 7) is 1-SNI.*

Proof: From Lemma 6 and Theorem 4, Example 3, Lemma 11 we have that the intermediate shares $\langle X_i \rangle$, $\langle Y_i \rangle$, $\langle Z_i \rangle$, $(i \in \{0,\dots,m-1\})$ can be perfectly simulated by the empty set of shares. We distinguish two cases:

Case 1: $|S_o| = 0$**:** Then $S_o = \emptyset$. For $V = \{\langle X_i \rangle\}$, $V = \{\langle Y_i \rangle\}$ for some $i \in \{0,\dots,m-1\}$, we have shown that they can be perfectly simulated from the empty set of shares.

Case 2: $|S_o| > 0$ For $i \in \{0,\dots,m-1\}$ we have $S_o = \{\langle Z_i \rangle\}$ and hence $V = \emptyset$. We have shown above how $\langle Z_i \rangle$ can be perfectly simulated from the empty set of shares.

□

Lemma 18. *The Activation gadget (Alg. 8) is 1-SNI.*

Proof: The algorithm only has output shares so we focus on them. From Lemma 6 and Lemma 12 we have that the output shares $\langle Z_i \rangle$ $(i \in \{0,\dots,m-1\})$ can be perfectly simulated by the empty set of shares. □

Theorem 9. *The Linear, Activation gadgets are 1-SNI.*

Proof: Follows from Lemmas 17, 18. □

Theorem 19 (Coron [6])**.** *The BooleanToArith gadget (Alg. 10) is* ***1-SNI***

Theorem 20 (Coron [6])**.** *The ArithToBoolean gadget (Alg. 11) is* ***1-SNI***

Theorem 21. *The Cmp gadget (Alg. 12) is* ***1-SNI***

Proof: We distinguish two cases. In the first, we consider (without loss of generality) the output sharing $\langle a \rangle$, i.e., $S = \{\langle a \rangle_0\}$ or $S = \{\langle a \rangle_1\}$, hence $t' = 0$ and $V = A_0 = \emptyset$, where $\langle a \rangle_0 = \langle \mathit{BooleanToArith}(\langle v \rangle^B, R_2, R_3) \rangle_0$ and $\langle a \rangle_1 = \langle \mathit{BooleanToArith}(\langle v \rangle^B, R_2, R_3) \rangle_1$. From Lemma 6 we have that both $\langle a \rangle_0, \langle a \rangle_1$ can be simulated by $\langle A_0 \rangle$. The same holds for the case of the output sharing $\langle b \rangle$.

In the second case, we consider the intermediate shares, i.e., $S = \emptyset$, hence $t' = 1$. For some $i \in \{0,1\}$ $V = \{\langle w \rangle_i\}$ can be perfectly simulated by the tuple $\langle \{\langle x \rangle_i, \langle y \rangle_i\} \rangle$ for some $i \in \{0,1\}$. From Lemma 6 we have that $\langle z \rangle_0, \langle z \rangle_1$ can be simulated by $\langle \emptyset \rangle$. Consequently, $\langle u \rangle_0, \langle u \rangle_1, \langle v \rangle_0, \langle v \rangle_1$ can also be simulated by $\langle \emptyset \rangle$ as they only depend on shares of $\langle z \rangle$. □

Theorem 22. *The Conv2d gadget (Alg. 13) is* ***1-SNI***

Proof: From Lemma 6 and Theorem 4, Example 3, Lemma 11 we have that the intermediate shares $\langle X_{i,j} \rangle$, $\langle Y_{i,j} \rangle$, $\langle Z_{i,j} \rangle$, $(i \in \{0,\dots,n-1\}, j \in \{0,\dots,m-1\})$ can be perfectly simulated by the empty set of shares. We distinguish two cases:

Case 1: $|S_o| = 0$**:** Then $S_o = \emptyset$. For $V = \{\langle X_{i,j} \rangle\}$, $V = \{\langle Y_{i,j} \rangle\}$ for some $i \in \{0,\dots,n-1\}$ and $j \in \{0,\dots,m-1\}$, we have shown that they can be perfectly simulated from the empty set of shares.

Case 2: $|S_o| > 0$ For $i \in \{0,\dots,n-1\}$ and $j \in \{0,\dots,m-1\}$ we have $S_o = \{\langle Z_{i,j} \rangle\}$ and hence $V = \emptyset$. We have shown above how $\langle Z_{i,j} \rangle$ can be perfectly simulated from the empty set of shares.

□

C Operation Breakdown for Basic Gadgets

	ADD	MUL	BIN
Add	4	0	0
Mul	4	4	0
DotProd(n)	4*n	4*n	0
Trunc	4	0	2
BooleanToArith	2	0	7
ArithToBoolean	2	32	122
ReLU	8	36	130

Table 7. Primitive Operation Breakdown of Basic Gadgets, ReLU and, Maxpool Gadget.