Hardware-Software Co-design for Side-Channel Protected Neural Network Inference

Anuj Dubey*, Rosario Cammarota†, Avinash Varna‡, Raghavan Kumar†, Aydin Aysu*
*Department of Electrical and Computer Engineering, North Carolina State University
{aanujdu, aaysu}@ncsu.edu
†Intel Labs
‡Intel Corporation

***Abstract*—Physical side-channel attacks are a major threat to stealing confidential data from devices. There has been a recent surge in such attacks on edge machine learning (ML) hardware to extract the model parameters. Consequently, there has also been some work, although limited, on building corresponding side-channel defenses against such attacks. All the current solutions either take the fully software or fully hardware-centric approaches, which are limited either in performance or flexibility.**

In this paper, we propose the first hardware-software co-design solution for building side-channel-protected ML hardware. Our solution targets edge devices and addresses both performance and flexibility needs. To that end, we develop a secure RISC-V-based coprocessor design that can execute a neural network implemented in C/C++. The coprocessor uses masking to execute various neural network operations like weighted summations, activation functions, and output layer computation in a side-channel secure fashion. We extend the original RV32I instruction set with custom instructions to control the masking gadgets inside the secure coprocessor. We further use the custom instructions to implement easy-to-use APIs that are exposed to the end-user as a shared library. Finally, we demonstrate the empirical side-channel security of the design with 1M traces.

***Index Terms*—machine learning inference, side-channel analysis, masking, flexibility.**

I. INTRODUCTION

Model development in machine learning (ML) frameworks has become an elaborate process with associated costs including research to develop efficient and effective algorithms, and training them over large proprietary datasets with expensive hardware. Therefore, trained models are considered Intellectual Property (IP) today. The confidentiality of these IPs has become a big concern due to the growing number of model-stealing attacks, including cryptanalytic [1], [2], and side-channel-based [3]–[9] attacks. Physical side-channel attacks can exploit the data-dependent power/electromagnetic (EM) signatures to that end. These attacks are a major threat because they are practical and hard to mitigate.

Although model stealing by exploiting power/EM side-channels has been repeatedly shown, building corresponding side-channel countermeasures is still largely unexplored. Existing works have focused on custom built hardware for protection [4], [5], [10]–[12] or integrated defenses into software in a microprocessor-based implementation [13]. These solutions lack either flexibility or efficiency. Indeed, 5a dedicated hardware implementation requires the user to know the details of the hardware design and may require the need to interface it with another controller. custom hardware will be rigid in terms of the neural network architectures it can support. Software-based solutions, by contrast, offer flexibility but do not provide custom hardware's efficiency. Hardware-software co-design methodology can address both concerns and has been used in the context of side-channel protection for cryptography [14] but is non-trivial to extend towards ML and requires making the right trade-offs and design decisions.

In this work, we address both the flexibility and efficiency limitations of earlier works by proposing the *first* RISC-V-based coprocessor design for secure neural network processing. There are two main advantages of having a RISC-V core in the design: 1) *usability*: the solution is easier to use without side-channel expertise for software developers who know developing ML models in C/C++, 2) *flexibility*: the hardware is capable of running different binaries that can program neural networks with different hyperparameters. A software-based approach may also provide these advantages but will lack the performance benefits of coprocessor hardware implementation. Thus, we believe that a hardware-software co-design approach provides the best of both worlds in terms of performance and flexibility. But achieving such a design is non-trivial and requires a cross-cutting approach across hardware, architecture, compiler, and software abstraction levels with the right trade-offs.

We make the following contributions in this work.

1) **Hardware:** We design a runtime reconfigurable coprocessor core that can selectively mask or unmask a certain layer operation to improve the performance and power of overall inference. We further optimize the design by reusing the unused datapaths when a certain layer is unmasked to improve the resource utilization on hardware.
2) **Architecture:** We add custom instruction-set extensions (ISE) in the RISC-V instruction set architecture (ISA) that can trigger the masking primitives in the secure neural network coprocessor. We also extend the original RISC-V toolchain to identify the ISEs and accordingly generate the binaries with these custom ISE embeddings.
3) **Software:** We develop a user-friendly software library to run secure inference to abstract the details of the custom ISEs from the user. We implement the software APIs internally using the custom ISEs that we add to the ISA.
4) **System Demonstration and Security Validation:** We

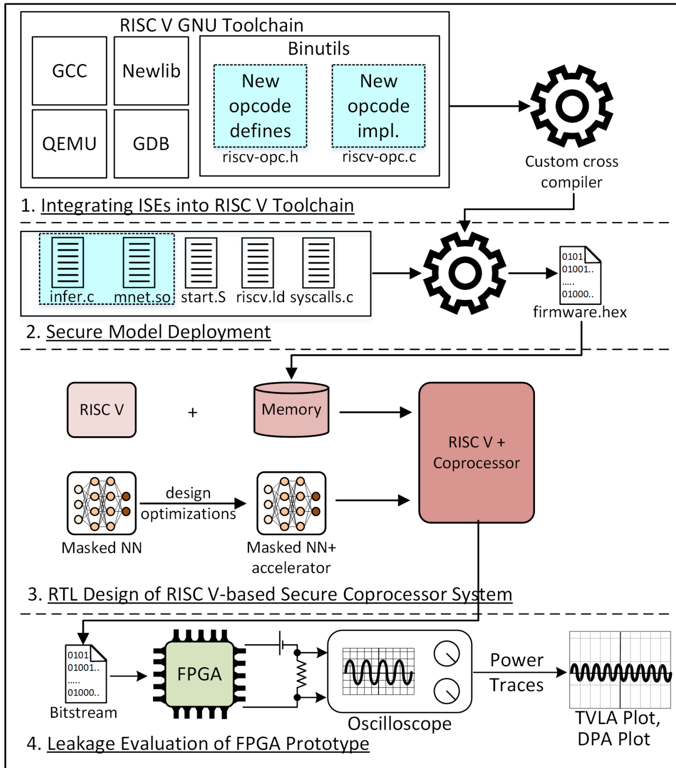

Fig. 1. We add new instructions in the RISC-V toolchain by adding the definitions and semantics of the new instructions in the opcode library of GNU binutils. We expose the APIs via the mnet shared library that has assembly implementations for the neural network operations. We load the cross-compiled binary on the RISC-V core coupled with a coprocessor to execute the neural network functions securely.

implement the complete system on an FPGA and validate the first-order side-channel security of the design with 1M power traces using the test vector leakage assessment (TVLA) and with differential attacks [15], [16].

5) **Extensions:** We propose extensions of our work for higher precision convolutional neural networks. We discuss the masking of new types of functions like integer convolutions, rectified linear unit (ReLU), and maxpool.

Fig. 1 outlines the flow of our solution and its security validation. After we decide the ISEs and their encodings, we modify the RISC-V compiler and the header files accordingly to map the corresponding C code to our defined custom instructions. The host would then load the memory of the RISC-V core with the generated firmware containing instructions and data. The coprocessor and the RISC-V read the instructions and data from the memory and process them accordingly. The final bitstream is mapped onto an FPGA for functional and side-channel security validation with standard t-tests and differential attacks.

II. Background

In this section, we briefly present relevant information on the commonly assumed threat model for side-channel attacks on ML hardware, the RISC-V framework, and hardware masking to help understand the rest of the paper.

A. Threat Models in Power Side Channel Analysis

Threat model defines the capabilities of the assumed adversary. The typical power side-channel model assumes that the adversary can measure the power while the platform runs secret computations. However, the leakage evaluation is time-consuming because this model requires actual power measurements. Thus, numerous theoretical models have been proposed in the literature to model the leakage [17]–[19], which can then be used to evaluate the security of a countermeasure without running real experiments and find leakages early on in the design phase. Eventually, the proofs are supplemented with empirical validations on actual power traces as well to ensure a complete security sign-off.

The *glitch-extended probing model* is a state-of-the-art theoretical model for a side-channel adversary on hardware [19], [20]. In this model, the adversary can directly probe 't' wires in the circuit, and also get information on all the wires in its input cone until the last registered point. Informally, security in this model implies that by using the information from probing 't' wires, the adversary cannot get any information about the secrets even when glitches occur. The security of the primitives in this model does not depend on the final hardware target (ASIC or FPGA) because the proofs do not make assumptions on the target hardware. Proving the security of large designs quickly becomes infeasible, and thus, the typical approach is to theoretically prove the security of smaller composable circuits, called 'gadgets', and build a larger circuit from them.

Empirical security validation of full circuits rely on statistical evaluation of real power traces. Statistical evaluations can be either model-based such as the differential power analysis (DPA) [16] that assumes a well-defined power model[1], or model-less such as the test vector leakage assessment (TVLA) [15], which do make any assumptions on the power model and just detect information leakage. In a DPA attack[2], the adversary 1) chooses an intermediate computation involving the secret, 2) collects multiple power measurements for different inputs and also computes the power model values for secret hypotheses, and 3) correlates[3] the power models with the power measurements, which usually yields a high value for the correct hypothesis. TVLA is more generic. It does not assume any power model and just detects information leakage by checking if the distribution of power traces for a fixed input is statistically indistinguishable from the distribution of the power traces for random inputs. The test quantifies the leakage using Welch's t-score given by the following equation.

$$t = \frac{\mu_{fixed} - \mu_{random}}{\sqrt{\frac{\sigma^2_{fixed}}{N_{fixed}} + \frac{\sigma^2_{random}}{N_{random}}}}$$

The leakage is considered statistically significant if the score crosses the threshold of ± 4.5. No information leakage in a TVLA test implies no leakage in the DPA test too because

[1]A function that maps data to its approximate power profile; Hamming weight and Hamming distance are two commonly used models

[2]We use correlation power analysis (CPA) but use "DPA" interchangeably.

[3]Pearson correlation coefficient is commonly used as the metric.

the information is statistically insignificant for the DPA test to correlate. However, for the sake of completeness, we provide the results for both tests in Section V.

B. Threat of Physical Side-Channels for AI/ML

Traditionally, physical side-channel attacks have mostly targeted cryptographic implementations to extract secrets, such as the key in AES [21]. However, given the growing need for confidentiality in machine learning models, numerous works have shown successful attacks to also extract the model parameters from a device running machine learning inference [3], [4], [6]. We follow the same attack setup for this work and build defenses against it. The training happens securely and the computed parameters are programmed into a secure memory inside the edge device. The programmed device then operates in an uncontrolled medium where an untrusted end user (adversary) can have physical access. *The adversary's goal is to learn the parameters of the neural network by conducting a DPA on the power traces captured during the inference computation on the device*. These parameters include weights and biases in fully connected layers and kernels in convolution layers. Adversary aims to steal the exact values of these parameters—known as the high-fidelity extraction of the model parameters [1]. Additionally, we also follow the assumption that adversary knows the hyperparameters of the model either because it is public, or by using the techniques mentioned in prior works [6].

To mitigate side-channel leakage, we use the masking primitives from prior works that are proven first-order secure in the glitch-extended probing model in cryptography [19], [20]. We then validate the empirical security of our complete design using both DPA and TVLA tests. We exclude invasive attacks such as clock glitching, or laser fault injection on the hardware. We also assume that only authenticated binaries can be loaded into the core. Thus, either the model owner programs the binary, or a third party first verifies their source code with the model owner, then generates a valid token, and then programs the binary on the device. If fault attacks or secure boot are concerns, they can be handled by a different layer of defense.

C. RISC-V ISA and Toolchain

RISC-V is an open-source instruction-set architecture with 47 base instructions that need to be present in any variant of the ISA [22]. Additionally, it is developed in a modular fashion to enable easy extensions over the base ISA. The encoding space of instructions is split into standard, reserved, and custom categories. The category is decided by bits [6:2] of the instruction and the exact mapping is listed in the latest ISA manual [22] of RISC-V. Any custom instructions should preferably use the encoding space allotted to the custom category because the standard space is already in use and the reserved space is kept for possible future standardizations. See Fig. 3 for more details on the encoding for various categories in RISC-V ISA. The RISC-V cross-compiler (or toolchain) is publicly available with its source code. It consists of many

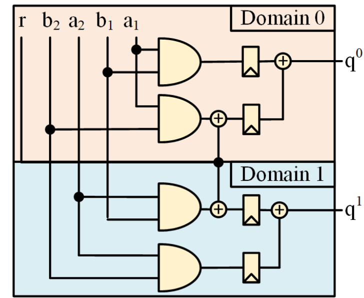

Fig. 2. Circuit diagram of the first-order secure DOM AND gate. The two domains are depicted by the pink and blue regions. The cross-domain partial products are refreshed, registered, and eventually compressed to produce two Boolean shares q^0 and q^1.

tools but the relevant ones for this work are the GCC, the Binutils, and the Newlib as Fig. 1 shows. Binutils contains the GNU assembler (`as`) and linker (`ld`). GCC is the GNU compiler for C, and Newlib provides the required low-level libraries for basic C routines like malloc, free, etc. Adding a new custom instruction requires modifying the source code of the toolchain and rebuilding it. The rebuilt toolchain can then be used to generate the binary for a source code containing the newly added custom instructions.

D. Hardware Masking

Masking is a common side-channel countermeasure. It splits the secret variable into multiple statistically independent and uniformly random shares and modifies the original algorithm to process these shares instead of the original secret and still maintain correct functionality. The power consumption is, therefore, decorrelated from the secret since the computations only happen on random shares. Based on whether the shares are split using exclusive-OR operation or modular addition, the scheme is respectively called Boolean or arithmetic masking. Multiple masking schemes have been proposed in the literature [23], [24]. Fig. 2 shows the circuit of the domain-oriented masking (DOM) based AND gate. We use it to mask Boolean functions because it is secure in the glitch-extended probing model and has a low randomness and area overhead [24]. It can be composed securely with an additional register at the output. Notably, this style of masking is also adopted in real-world products such as Google's OpenTitan [25].

E. Neural Networks Basics

Neural networks are a class of ML classifiers frequently used for classification problems in computer vision, language processing, etc. They consist of units called neurons that perform a weighted summation followed by a bias addition and a non-linear transformation. Multiple neurons are stacked together in layers that feed their results to the next layers. The layer is called fully connected (FC) if its neurons are connected to all the neurons of the previous layer. Another flavor of neural networks uses convolutional layers. The idea is to process smaller regions of the image and extract meaningful information before using the FC layers using kernels [26], [27]. The connection weights or kernel values are tuned during

inst[4:2] / inst[6:5]	000	001	010	011	100	101	111
00	LOAD	LOADFP	**custom-0**	MISCMEM	OP-IMM	AUIPC	OPIMM32
01	STORE	STOREFP	custom-1	AMO	OP	LUI	OP32
10	MADD	MSUB	NMSUB	NMADD	OP-FP	OP-V	custom-2
11	BRANCH	JALR	reserved	JAL	SYSTEM	reserved	custom-3

(a)

funct7	rs2	rs1	minor	rd	major		
31:25	24:20	19:15	14:12	11:7	6:5	4:2	1:0

(b)

Fig. 3. Figures (a) and (b) respectively depict the overall opcode map for the RISC-V ISA and the encoding of an R-type instruction. Our design uses the major opcode of custom-0 (highlighted) to distinguish our proposed custom instructions from the base instructions, and the minor opcode to distinguish between each custom instruction.

the training process and typically in floating point representation. However, to reduce the power and memory footprint for hardware implementations, quantized neural networks have been proposed that limit the precision of the parameters to fewer bits [28], with the extreme case being binary [29]. Interested readers can refer to the survey by Deng et al [30].

III. The Programming Model and Custom ISEs

This section describes the programming interface to run ML applications on our secure platform, and the custom ISEs that we add to the ISA.

Instead of exposing the assembly-level custom ISEs directly to the user, we expose C-based[4] APIs, which are easier to understand and debug. We provide the APIs as part of a shared library. The rationale behind the syntax of the APIs is to keep it close to existing ML frameworks. We develop an interface similar to the Sequential model of TensorFlow Keras [31]. Our proposed programming interface configures and executes each layer of the neural network sequentially similar to the Keras model that builds the model by stacking layers.

A key advantage of our solution is exploiting layer-wise configurability of this software interface to activate/deactivate the masking of each layer. This helps trading off security with performance. Selective masking has also been explored in cryptography—prior works propose masking only the first and last rounds of AES because those are the most vulnerable rounds in a DPA attack [32]. Neural networks are no different in this regard. Additionally, this also enables only protecting the parameters of the layers retrained using transfer learning and saves the power and clock cycles spent on unnecessary masking of layers with public parameters [33]. The library provides the four following APIs.

```
void fetch_input(int* image, int num_images);
void input_layer(int* image, int num_images,
    int* weight0, int num_weight0, int* bias,
    int num_bias, int masking_enable);
void hidden_layer(int* weight, int
    num_weight, int masking_enable);
void output_layer(int* weight, int
    num_weight, int masking_enable);
```

[4]Our approach is not fundamentally limited to a particular software language but we chose C given its suitability for embedded platforms.

We provide one API each for the input[5] and the output layer computations. We use the stacking approach for hidden layers: each call to the `hidden_layer(.)` API triggers a hidden layer computation where the hardware processes the previous layer results to compute the results of the next layer. We implement the APIs using inline assembly and custom ISEs. We describe the encoding of the custom ISEs next.

Instructions in the RISC-V ISA are classified as either an R, S, I, or U-type, which is based on how the 32 bits of the instruction are encoded. We choose the R-type format shown in Fig. 3(a) because some instructions require two source operands and a destination operand. Fig. 3(b) shows how the bits [6:2] are encoded for various categories of instructions. The base opcode encoding space has the two least significant bits [1:0] set to one. Since we only need 5 custom instructions, we choose the standard 30-bit base encoding space.

We use the major opcode (bits [6:2]) to distinguish our custom instructions from the base instructions in RISC-V ISA and the minor opcode (bits [14:12]) to distinguish between each custom instruction. We choose the *custom-0* space without loss of generality. The major opcode for this space is *00010*.

We add five custom instructions to control the coprocessor from RISC-V. The instruction names and operations are explained below.

1) `mnn.cfgwr rs1 rs2`. This instruction stores the pointer to a data structure (`rs1`) and its size (`rs2`) in a configuration register inside the coprocessor. It is used to store the pointers to the input pixels, the weights, and the biases of the neural network. It is also used to store the hyperparameters such as the sizes of the input, hidden, and output layers.
2) mnn.<i/h/o>layer rs1. These instructions have just one source operand (`rs1`) that controls the enabling or disabling of masking for the layers. Based on the opcode, the instructions `mnn.ilayer`, `mnn.hlayer`, and `mnn.olayer` trigger the input, hidden and output layer computations inside the coprocessor, respectively.
3) `mnn.ifetch rs1 rs2`. This instruction fetches the required number of input pixels (`rs2`) from the host PC. The user allocates the desired memory space for the pixels and sends the associated pointer in this instruction ((`rs1`).). The hardware then uses this information to store the fetched pixels from the host to that location.

IV. Hardware Design of the Complete System

Fig. 4 shows the block diagram of our complete design. The important blocks are the RISC-V core (referred to as just *core* from hereon), the dual-ported memory accessible through a shared bus, and the coprocessor. The coprocessor further consists of the command decoder (CMD) and the secure neural network unit (SNNU). In the following sections, we describe the design details in a top-down fashion—how the design sequentially processes the high-level commands from the host PC to eventually perform the secure neural network inference.

[5]By input layer computation, we mean processing the input pixels to compute the first hidden layer.

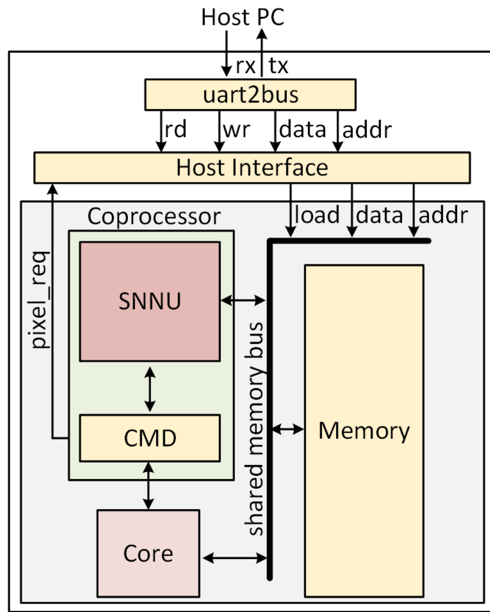

Fig. 4. The figure depicts major components of the RISC-V based SoC that we design. Both the RISC-V and coprocessor share the memory interface. The command decoder decodes the custom instructions sent on the PCPI interface and relays appropriate commands to the coprocessor to execute the required computations.

A. The RISC-V Integration

We select the open-source PicoRV32 RISC-V core[6] for this work because it has a low area footprint and provides the basic features we need for our solution. The design communicates to the host PC via the UART interface and processes the UART signals in two steps. First, the uart2bus IP from OpenCores[7] converts the UART signals received from the host PC to address-based read and write commands. Then, the design processes these commands using the host interface to generate specific commands using address mapping. The UART can be replaced with any other interface that can generate the required bus signals for the host interface. This keeps our design modular and is the reason why we did not merge the two IPs into one that directly translates the UART signals to design commands.

We depict the entire sequence of operations in Fig. 5. The host first sends a software reset for a clean start and then loads the cross-compiled RISC-V binary to the memory using the `romload` command. The binary contains the model parameters, hyperparameters, and custom ISEs to perform the neural network inference. The `start` command activates the core to fetch instructions. The core has an in-built peripheral coprocessor interface (PCPI) that follows a valid-ready[8] protocol. The CMD unit inside the coprocessor decodes the custom instructions discussed in Section III and issues commands to the SNNU to perform the respective computations based on the decoded instruction. The logical sequence of instructions is: `mnn.ifetch`, `mnn.ilayer`, a bunch of `mnn.hlayer`

[6]https://github.com/YosysHQ/picorv32

[7]https://opencores.org/projects/uart2bus

[8]Master sends a request to the slave with an asserted valid signal, and the slave responds back with the expected output with an asserted ready signal.

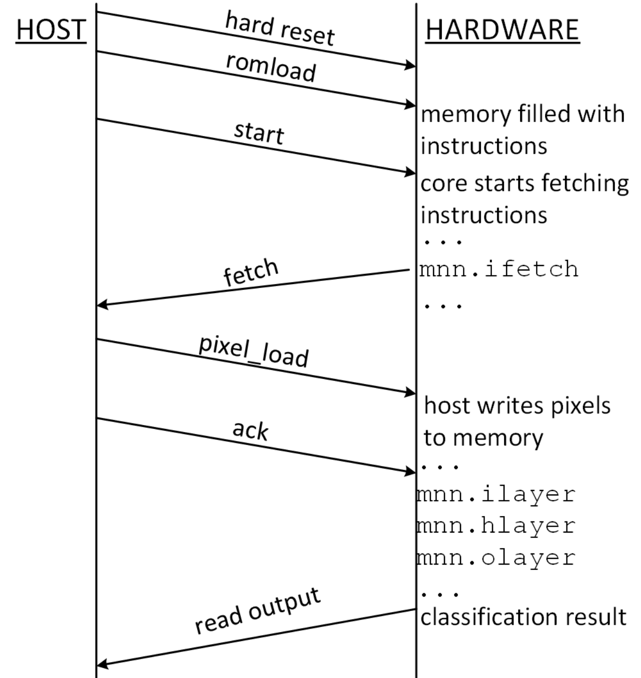

Fig. 5. The figure depicts the entire start-up sequence of our proposed design. The host first resets the design, then flashes the instructions to the memory, then writes the inputs when requested by the hardware, and eventually reads back the final classification result when it is available.

instructions, and finally the `mnn.olayer`. `mnn.ifetch` sends a fetch request to the host. The host writes the pixels to the desired memory location and sends an acknowledgment, which unblocks the core to execute the remaining layer execution instructions.

The core writes all the data such as input pixels, weights and biases directly to the memory while executing the instructions from the binary. The coprocessor can directly read from the respective addresses during its computations because it shares the memory access with the core. We choose to share the memory between the core and coprocessor instead of having distinct memories to avoid wasting cycles copying data from one memory to another.

B. The Coprocessor Interface Design and Operating Principle

The CMD breaks down each custom instruction it receives from the core into multiple commands for the SNNU. We explain this next by describing how a typical high-level instruction stream triggers multiple events in the NN unit. The following listing gives an example of an instruction sequence to perform inference using an MLP with one input, two hidden, and one output layers. The Listing abstracts the initialization of the pointers to data structures `image, w0, w1,w2, bias`, and the respective sizes `ni, nw0, nw1, nw2, nb` in the `init()` call. `m_en` variable is used to enable or disable masking for a certain layer and can be updated accordingly.

```
int main(){
  ... //declare and initialize pointers and
      variables
  init(image,w0,w1,w2,bias);
  fetch_input(image, ni);
```

```
  m_en=1;
  input_layer(image, ni, w0, nw0, bias, nb,
      m_en);
  hidden_layer(w1, nw1, m_en);
  output_layer(w2, nw2, m_en);
}
```

The `fetch_input(.)` API triggers the CMD to assert the `pixel_req` signal shown in Fig. 4. The host PC sends pixels with the associated counts ranging from 0 to $ni - 1$. The coprocessor adds these counts to the image pointer to compute the write address for each pixel, and generates the write request to the memory. The `input_layer(.)` API internally first calls the `mnn.cfgwr` instruction to configure the starting addresses and sizes of the bias, image, and weights between the input and first hidden layer. The CMD accordingly generates the write commands to the configuration registers inside coprocessor to store this information. Next, the API calls the `mnn.ilayer` that sends the masking configuration of input layer via one operand. The CMD configures the masking control register for the layer and then sends a trigger pulse to initiate the input layer computations. The SNNU computes the first hidden layer activations, stores them in a local activation memory, and signals the CMD once done.

Next, the code calls the $hidden_layer(.)$ API to compute the second hidden layer of the neural network. In a generic code, this API will be called as many times as the number of hidden layers in the network. Each call to the API first configures the details of the weights between the previous and next hidden layer using the `mnn.cfgwr`, and then configures the mask enable bit and triggers the hidden layer computations using the `mnn.hlayer` instruction. The SNNU computes the second hidden layer activations, stores them in the second local activation memory, and communicates to the CMD once done. Since the SNNU is layer-sequentialized, it only maintains two activation memories, in which the writes and reads keep ping-ponging. Finally, the code calls the $output_layer()$ API that computes the non-binary output layer activations to produce the inference result. The result is either Boolean-shared or not based on the masking configuration of the output layer.

C. The Secure Neural Network Unit

In this subsection, we first describe the masking gadgets that exist in prior literature for efficient masking of neural network functions, which we augment for our baseline design. Then, we describe our novel design changes to develop a masked neural network design that can be reconfigured in runtime to securely disable or enable the masking of individual layers. We also discuss how we implement our optimization of reusing the masking-related datapaths during the evaluation of unmasked layers to improve the overall inference latency. We finally discuss novel extensions of our hardware to quantized neural networks beyond binarized.

1) Baseline Masked Hardware Design

Prior works observed that a straightforward application of masking to neural network operations could be very expensive [5]. One way to alleviate this issue is by incorporating modular arithmetic in neural network functions [11]. The key idea is to find a large enough modulus to minimize the overflow and let the neural network perform a correct inference. Modular addition is compatible with arithmetic masking and thus completely avoids the use of expensive Boolean masking needed for masking regular additions. It significantly reduces the area, latency, and randomness costs. We adopt these techniques to build our baseline masked design.

(a) Weighted Summations. The design performs modular additions instead of regular additions. We restrict the modulus to a power of 2 for efficiency and the datapath width to that power. To mask the modular additions, the hardware can now utilize arithmetic masking. It splits the input pixels into two independent additive shares and individually performs weighted summations on each share. It sends the two shares to the masked activation function next.

(b) Activation Function. The activation function is a threshold function for binarized neural networks with modular arithmetic; the threshold is half of the modulus. The function checks the MSB to compare against the threshold. Prior works propose building a masked carry propagator that evaluates the Boolean shares of the MSB securely. Following the latest work, we also use a Kogge-Stone tree to propagate the carry and replace the regular AND gates in the logic with DOM AND gates.

(c) Output Layer. Modular arithmetic perturbs the distribution of the output summations and leads to incorrect classifications [11]. This is addressed by splitting the default straightforward computation of the maximum confidence score into two phases. The first phase finds the maximum below the threshold of half modulus, and the second phase computes the global maximum if no maximum was found below the threshold. Finally, the masked output layer block returns the Boolean shares of the confidence score and the classified class.

(d) Share Conversion. The design uses arithmetic masking for weighted summations and Boolean masking for non-linear operations like activation function, and output layer. Share conversion should be performed with care otherwise the conversion process itself can leak the secret [34]. We adopt the prior proposed design for the conversions [11], which is basically a pipelined version of the originally proposed scheme by Golic et al [35]. The pipelining reduces the glitches and makes the gadget secure in the probing model.

2) Hardware Support for Runtime Reconfiguration

Our programming model supports the selective activation/deactivation of masking for each layer, which is a key enabler in offering security-performance trade-off. Our hardware has two critical features to this end. First, it re-uses the same datapath for both settings while doubling the throughput for unmasked layers via *runtime dual datapath reuse*. Second, the re-use of the datapath do not reduce the security of masking. Achieving these two properties is non-trivial and requires a

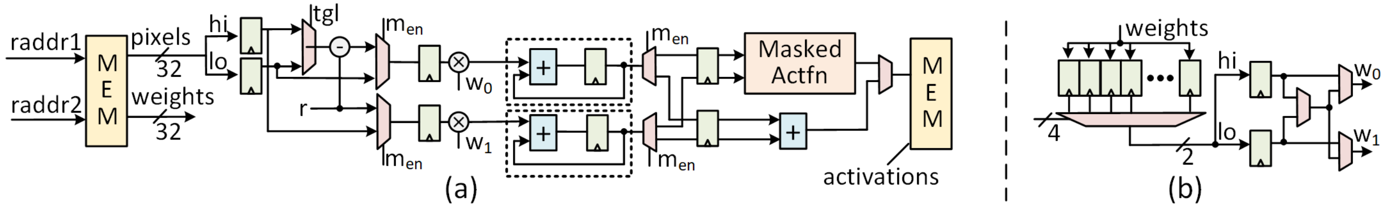

Fig. 6. The input layer schematic that depicts the reconfigurable paths for unmasked and masked modes. (a) shows how the design reads the pixels and multiplexes them based on the toggle (tgl) and masking mode (m_{en}) signals to eventually feed them to the two accumulators (shown inside the dotted lines). Similarly, (b) depicts how the design multiplexes the 32 weights once read to correctly feed them in the accumulator datapath for multiplications.

careful approach as we describe in detail in this subsection.

Fig. 6 shows the datapath of the runtime reconfigurable input layer that supports both unmasked and masked modes. The key components to be reused are the two MAC units that perform weighted summations on the two arithmetic shares when the layer is masked. We develop a design that automatically switches to using the two MAC units for unmasked layers. Each node computation in a neural network involves adding multiple partial products of the previous layer activations with the corresponding weights. These partial product computations are independent of each other. Thus, the design uses the two datapaths to compute the even and odd partial products and accumulate them into two parallel summations. Finally, the design adds the accumulated summations to compute the complete sum.

In Fig. 6 (a), the design loads two consecutive pixels into the hi and lo registers. The goal is to send the partial products of lo and hi to the two MACs when the layer is unmasked. When masked, the design should send arithmetic shares of lo and hi in consecutive cycles; r is a fresh random mask generated by a PRNG. The design alternates the output between lo and hi pixels every cycle using the toggle tgl signal. The second multiplexer sends either the unmasked lo and hi pixels, or the masked shares to the MAC units based on the mask configuration (m_{en}).

Fig. 7 depicts the datapaths for the hidden layer computation. The design uses the XNOR-popcount operation over the previous layer activations to perform weighted summations in the hidden layers. It stores the activations either in clear or as Boolean shares depending on the masking mode. To perform an unmasked layer computation, the design unmasks the XNORed result before sending it to the MAC units. During the masked mode, however, the design first sends the two Boolean shares of the XNORed result to the Boolean-to-arithmetic unit to convert them to arithmetic shares, and then sends them to the MAC units for weighted summations. The scheduling is similar for the rest of the layers.

Having such a reconfiguration required careful security considerations throughout the design. If the input layer operates in unmasked mode, then the design adds the final output from the two MAC units and stores the activation result in the memory. However, this path should be disabled in the masked mode to prevent the reconstruction of the original summation. We ensure this using a demultiplexer at the MAC outputs which relays a zero instead of the actual MAC outputs if

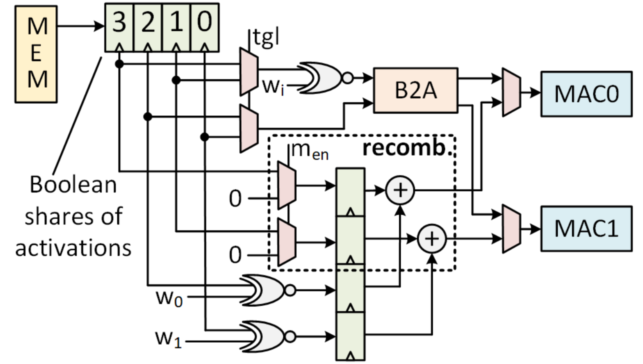

Fig. 7. The reconfigurable hidden layer datapath. Each memory location stores the Boolean shares for two activations–(3,2), and (1,0) are the shares of two activations. Based on m_{en}, the design either recombines the shares (inside recomb.) or sends them to the Boolean-to-arithmetic (B2A) block for conversion to arithmetic shares to be used later during weighted summations.

masking is enabled. We add registers at the demultiplexer outputs to prevent leaky glitches–the summations may get temporarily combined during the masked mode if the m_{en} signal arrives slightly later than the summations. Similarly, in the hidden layer datapath shown in Fig. 7, we add registers at the multiplexer outputs before the design performs the XOR-sum to recombine the shares.

Applying modular arithmetic on the output layer activations perturbs the output and causes incorrect classifications, which has been solved previously using a thresholded maximum computation [11]. We follow the same strategy to correctly compute the inference result. However, instead of reusing the high latency masked output layer also for the unmasked mode, we instantiate a separate unmasked output layer unit. This unit includes only a register and a comparator to keep comparing each confidence score with the previous one and to update the max register if it is greater. It was more advantageous to add this small hardware to accelerate the unmasked output layer result generation. Finally, based on the masking mode, the design either selects the output of the masked output layer or the unmasked output layer and communicates to the core.

3) Masking of Quantized and Convolutional Networks

In this subsection, we present novel solutions for neural networks with higher precision than just one bit and those that can include convolutions. Representing the weights and activations with more than one bit enables the network to capture more subtleties in the parameters and improves the overall classification accuracy because it can better approximate the unknown function. Our solutions are not just confined to BNNs

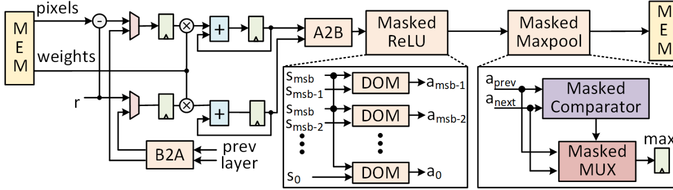

Fig. 8. Masked datapath for a higher precision neural network. Masked ReLU is constructed by performing a secure AND between the MSB and other input bits using DOM-AND gadgets. Masked maxpool is constructed by securely comparing the ReLU outputs and selecting the maximum.

and can be easily extended to neural networks with a higher precision like 4 or 8-bit weights and activations. Next, we describe how to mask an 8-bit quantized network with modular arithmetic.

The unique operations for an 8-bit quantized neural network compared to a BNN include (1) regular 8-bit multiplication of the weights with the pixels instead of multiplexing 2's complement in the input layer and XNOR-popcount in hidden layers, (2) a rectified linear unit (ReLU) function instead of a sign function, and (3) computations of the maximum score in the pooling window rather than simple OR operations. In the baseline unmasked design, the hardware multiplies the 8-bit signed weights in the kernel with the 8-bit signed input pixels sequentially and accumulates the sum along with the bias. It then feeds the sum to the ReLU unit. Equation (1) shows the formulation of the ReLU function with the modular arithmetic

$$\texttt{ReLUMOD}(x) = \begin{cases} 0, & \text{if } x \geq \frac{K}{2} \\ x, & \text{otherwise} \end{cases} \tag{1}$$

where x is the output of the convolution and K is the modulus. A convolution is generally followed by maxpool over the activation values. This essentially involves finding the maximum element in the pooling window and saving the results for the subsequent layer computations.

Fig. 8 shows the datapath for a fully-masked higher-precision neural network. In the masked design, the hardware can still use arithmetic masking of input pixels before the multiplication with the weights and perform convolutions in parallel for the two arithmetic shares. At the completion of each convolution, the hardware sends the arithmetic shares of the resulting sum to the activation function. The activation function needs to threshold the shares based on the MSB of the actual sum. At the Boolean level, the ReLU function is essentially a logical AND operation of the inverted MSB with all the other bits of the input. Since this is a Boolean function, the design first needs to convert the arithmetic shares of the summation to Boolean shares. It can then perform a secure AND operation of the MSB with the rest of the bits to yield the two Boolean shares of the activation result.

Finally, to perform the Maxpool, the hardware has to compare the Boolean shares of activations in the pooling window with each other and find the maximum value. This is achieved by using a masked comparator to compare the confidence scores, and a masked multiplexer to select the higher score, following the DOM-AND gates as in prior work [11].

TABLE I
AREA BREAKDOWN OF THE DESIGN IN TERMS OF MAJOR COMPONENTS.

Component	FF	LUT
uart2bus	153	223
Host Interface	23	40
PicoRV32	649	1400
CMD	12	15
SNNU	3118	2883
Misc	9	143
Total	**3981**	**4797**

V. IMPLEMENTATION RESULTS AND COMPARISONS

We present the area and performance results that substantiate our efficiency claims over software-only solutions compared to previous work. We also present the side-channel validations for our proposed solution.

A. Area and Latency

The total area cost of our proposed solution is 3981 FFs and 4797 LUTs on Spartan-6 FPGA using Xilinx ISE 14.7. We chose this FPGA as it is used in Sakura-G side-channel verification board. Table I shows the area of individual components. Most of the area contribution is from the SNNU as expected. In terms of latency, the increase in the number of cycles from the fully unmasked to fully masked mode is 2.077×. The exact latency varies linearly with the number of layers and number of nodes per layer. Our design consumes 4997 cycles in the unmasked mode and 10,150 cycles in the masked mode for a BNN with 64 input nodes, 2 hidden layers with 64 neurons, and 10 output nodes. In terms of the memory footprint of our library, the binary has 82 fixed additional instructions contributing 328 bytes.

Table II compares the area and latency of our solution with prior works. It is difficult to make exact comparisons between the works because of the varying hyperparameters, parallelization modes, and implementation platforms (FPGA versus ASIC versus microcontroller). Still, we try our best to provide the comparison for completeness. We first choose a common hyperparameter set of 784 input nodes, one hidden layer of 512 nodes, and an output layer of 10 nodes for comparison; [13] already uses this configuration. All the works have a throughput of 1 summation/cycle[9]. Thus, we assume that the latency varies linearly with the total number of summations per inference, which is given as the sum of products of the number of nodes in two subsequent layers.

We scale the originally quoted latency (the Latency column) of the works to the expected latency of our chosen hyperparameter set (the Latency (N) column) for each work. The ASIC work [12] does not quote any latency numbers, thus, we assume it to be equal to the number of weighted summations. The results show that the ASIC and FPGA solutions have a comparable latency[10] of around 4×10^5, which is much lower

[9]The ASIC solution [12] actually has a throughput of 8 summations/cycle but we assume only 1 PE instantiation for this comparison.

[10]We refer to clock cycles. The actual latency might be much lower for an ASIC because of the high design frequency compared to FPGAs.

TABLE II
AREA AND PERFORMANCE COMPARISON WITH PRIOR WORKS.

Work	Area (LUT+FF)	Latency (cycles)	Latency (N) (cycles)	Programmable
[5]	17457	2.94×10^6	4.2×10^5	No
[11]	10644	2.91×10^6	4.1×10^5	No
[13]	NA[1]	1.97×10^7	19.68×10^6	Yes
[12]	NA[2]	NA[3]	4×10^5	No
This work	**8778**	**10150**	4.67×10^5	**Yes**

[1] microcontroller-based solution; no LUT/FF equivalents;
[2] ASIC solution; no LUT/FF equivalents;
[3] No latency numbers in the manuscript;

than that of the microcontroller-based solution, as expected. However, while the microcontroller is programmable, the ASIC and FPGA are not. Our proposed solution is almost as fast as the hardware solutions and provides the same programmability benefits as that of a microcontroller. Thus, our hardware-software co-design-based solution gives the best of both worlds by providing both programmability and high performance without sacrificing security.

B. Side-Channel Validation

We use both DPA and test vector leakage assessment (TVLA) methods to perform side-channel validations [15]. We use Sakura-G as the testing platform and Picoscope 3206D as the oscilloscope. We run the design at 10 MHz and set the sampling frequency of the oscilloscope at 125MHz. Following prior published works at HOST [36][11], we validate the empirical security with 1M power traces.

1) DPA Results.

We conduct DPA to demonstrate the side-channel vulnerability of neural network hardware implementations against a targeted attack. Furthermore, we also show how our proposed solution resists such an attack. Fig. 9 shows the results of the DPA attack on the unmasked and masked implementations. We target the activation function for the attack, use the hamming distance power model[12], and hypothesize on the possible weights. We set 8 input pixels to be non-zero, and hypothesize on the corresponding weights for those pixels. This reduces the number of hypotheses from 2^{64}, to 2^8. [13] Fig. 9 (a) clearly shows a high correlation only for the correct hypothesis, at the exact point in time when the activation is computed. The leakage is statistically significant after 6000 measurements. Fig. 9 (b) shows the same attack on the masked implementation, which quantifiably fails even with 1M power traces. Note that the latency of the target operation in the masked design ($3.2\mu s$) is twice compared to that of the unmasked design ($1.6\mu s$) because the unmasked design uses both the datapaths to quickly compute the summations as discussed in Section IV-C2.

[11]Some works also just used 100k traces [4], [37].

[12]The memory storing the activations is not reset between measurements.

[13]Note that our threat model assumes chosen-plaintext as in prior works; hence, such attacks can be conducted.

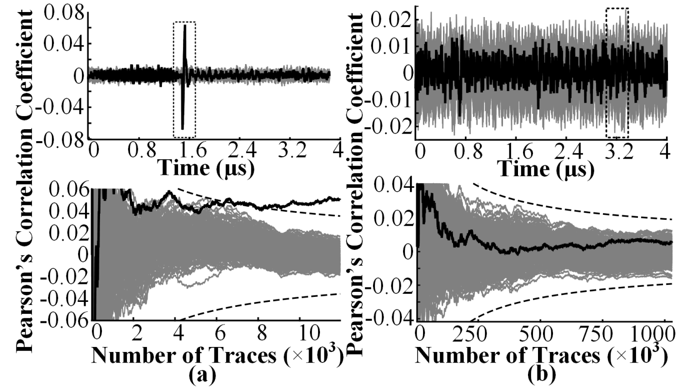

Fig. 9. Figure (a) top plot shows a high correlation peak only for the correct hypothesis in black; the bottom plot shows its evolution with the number of traces, which becomes statistically significant with confidence of 99.99% (shown by the dotted lines) at 6000 traces. Figure (b) shows the same results for the masked design, where we neither observe a high correlation peak for the correct hypothesis nor does the correlation become statistically significant before 1M traces demonstrating resistance to the DPA attack.

2) TVLA Results.

Since DPA is insufficient and atypical to exhaustively evaluate side-channel security, we also conduct the more generic TVLA test. We conduct TVLA on four masked neural network configurations C1 to C4 for a robust side-channel validation. For all the experiments we set the hyperparameters as 1 input layer with 64 nodes, 2 hidden layers with 64, and an output layer with 10 nodes. C1 has all the layers unmasked, C2 is all the layers masked, C3 has only the second hidden layer unmasked, and C4 has only the output layer unmasked.

Fig. 10 shows the results of our experiments. Figures 10 (a) and (c) depict the overall power trace of a fully unmasked and fully masked configuration. We can clearly observe the two hidden layer computations using simple power analysis as the two high amplitude bands; the bands have higher amplitude in the case of the masked design compared to the unmasked design, which is expected because of higher activity of PRNGs and masked datapaths. Also, the layer power activity is lower than that of the baseline processor activity in the unmasked design, which is why the power drops during the layer computations, and the power bumps are actually between the layer executions. Also, the execution time for the masked design is twice that of the unmasked configuration, which is expected because of the dual path reuse optimization in our design for the unmasked layers.

Figures 10 (b) and (d) show the TVLA evaluations of configurations C1 and C2. All the layers are unmasked in C1; thus, the t-scores cross the TVLA threshold of ± 4.5. C2 has all the layers masked, and therefore, we do not see the t-scores crossing the threshold throughout the execution except the input layer. This is because the design loads the pixels while computing the arithmetic shares, and that load operation results in the input correlations. This has been observed in prior works too [5], [11], and is verified by running an experiment with buffered arithmetic shares instead of generating them on-the-fly. We also verify our design with that approach. We used 1M traces in each fixed and random dataset to validate the

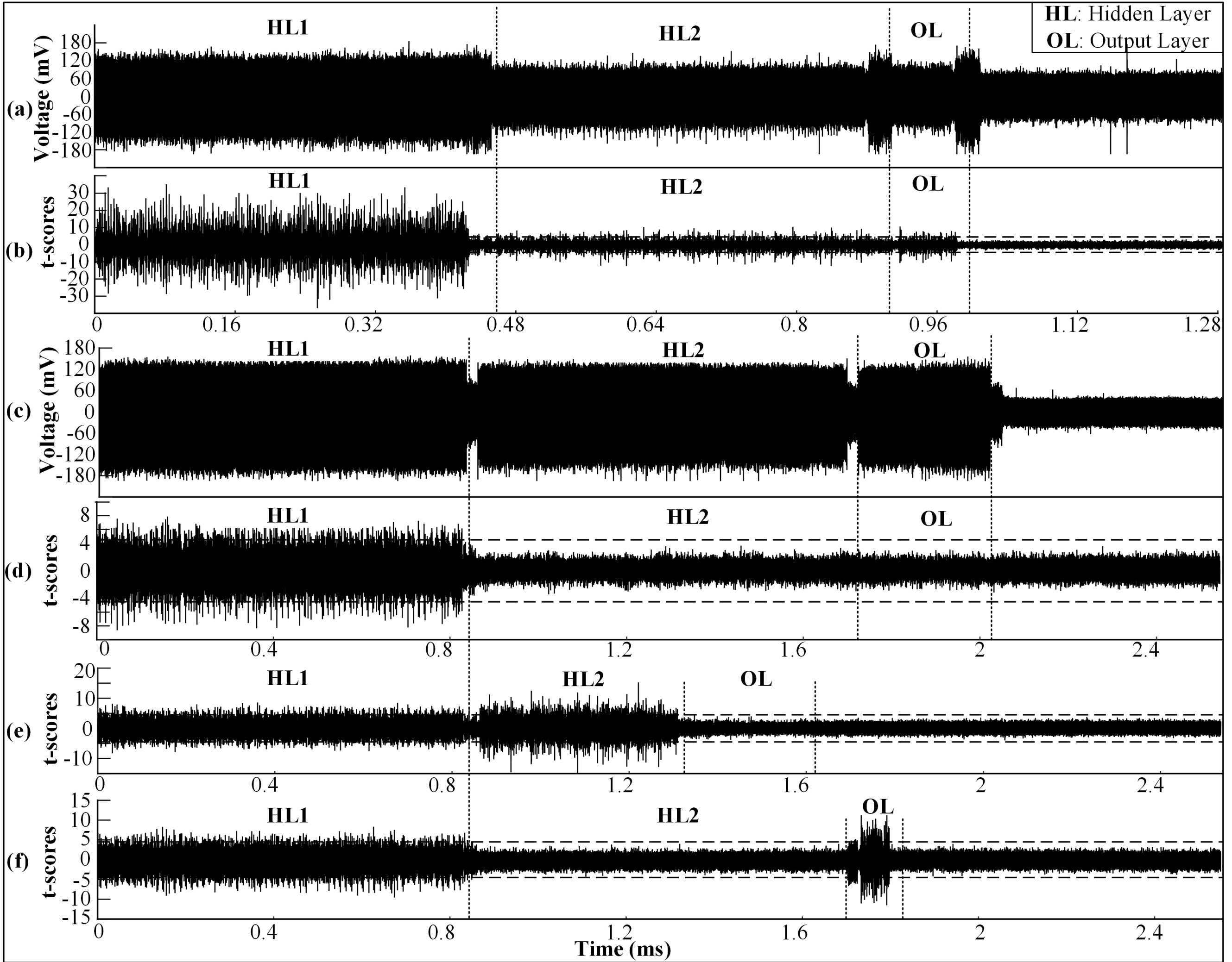

Fig. 10. The figure shows averaged power consumption and TVLA results from our experiments. Figures (a) and (c) show the mean power trace for the unmasked and masked designs, where, we observe higher activity in the masked design as expected. Figures (b), (d), (e), and (f) show the t-scores observed for the configurations C1-C4 in order. C1 and C2 are fully-unmasked and fully-masked designs, therefore, we observe t-scores higher than ±4.5 (denoted by horizontal dashed lines) throughout the trace in (b), but not in (d). C3 has only HL2 unmasked, and C4 has only OL unmasked. Hence, the t-scores cross the threshold only during those regions in (e) and (f). The t-scores cross the threshold during input layer computations with masking because of input correlations.

side-channel security following prior works [5]. (e) and (f) show the TVLA results for configurations C3 and C4. Masking is disabled in the second hidden layer in C3, therefore, we observe t-scores greater than the threshold during the second hidden layer computation. The time of execution is reduced to half with our optimization. Masking is re-enabled for the output layer and we can observe the security empirically as well because the t-scores stay within the limit of ±4.5 throughout the output layer computation. For configuration C4, we observe no leakage during the second hidden layer computation because masking is enabled, and higher t-scores crossing the TVLA threshold during the output layer execution because masking is disabled. Thus, the design successfully masks/unmasks the layers based on user configuration.

C. Existing Coprocessors for ML Acceleration

Some existing works have built coprocessors for a RISC-V core to accelerate ML [38]–[40]. However, none of them focus on side-channel security—instead, they exclusively focus on conventional design metrics such as power, performance, and area. Our hardware-software co-designed approach is the first one to consider side-channel security as a design dimension. Our goal is not to provide the most power/performance/area-optimized solution but to demonstrate the risk of side channels and provide a side-channel-aware hardware-software co-design framework. We quantify the relative overheads of the defense, which is likely to preserve if the hardware is deeper pipelined or parallelized for optimizations. Most prior works try to integrate a systolic array to the RISC-V core, either through the RISC-V vector extension or through a custom coprocessor interface. Our proposed mitigation techniques are applicable to all these works too, and the overheads should linearly scale with the number of processing elements. Further research is needed to quantify the exact amount, and such works can utilize the analysis and results of this work.

VI. Conclusion

A custom hardware solution is suitable for efficient masking of *cryptographic* applications, which have been thoroughly

standardized by NIST/ISO. By contrast, there are no such standards for neural networks. Therefore, addressing the flexibility/usability needs while preserving hardware efficiency is a critical concern, which has not been explored before. This paper demonstrates a hardware/software co-design approach for *flexibly and efficiently* implementing the side-channel masking for neural networks. Doing so required a full-stack approach that tackles the application software model, compiler/microcode augmentations, micro-architecture integration, and silicon verification. The results have shown the promise of our approach in terms of integration to a higher-level language and reasonable overheads. Our paper thus informs future system designers, software developers, computer architects, and silicon hardware engineers about the best practices and future challenges.

References

[1] M. Jagielski, N. Carlini, D. Berthelot, A. Kurakin, and N. Papernot, "High accuracy and high fidelity extraction of neural networks," in *29th {USENIX} Security Symposium ({USENIX} Security 20)*, 2020.

[2] N. Carlini, M. Jagielski, and I. Mironov, "Cryptanalytic extraction of neural network models," in *Advances in Cryptology - CRYPTO 2020 - 40th Annual International Cryptology Conference, CRYPTO 2020, Santa Barbara, CA, USA, August 17-21, 2020, Proceedings, Part III*, ser. Lecture Notes in Computer Science, vol. 12172. Springer, 2020, pp. 189–218.

[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 2019, Santa Clara, CA, USA, August 14-16, 2019*. USENIX Association, 2019, pp. 515–532.

[4] 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 2020, San Jose, CA, USA, December 7-11, 2020*. IEEE, 2020, pp. 197–208.

[5] ——, "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*. IEEE, 2020, pp. 51:1–51:9. [Online]. Available: https://doi.org/10.1145/3400302.3415649

[6] 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 2020*. IEEE, 2020.

[7] W. Hua, Z. Zhang, and G. E. Suh, "Reverse engineering convolutional neural networks through side-channel information leaks," in *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*. IEEE, 2018, pp. 1–6.

[8] S. Potluri and A. Aysu, "Stealing neural network models through the scan chain: A new threat for ml hardware," in *2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, 2021, pp. 1–8.

[9] M. Méndez Real and R. Salvador, "Physical side-channel attacks on embedded neural networks: A survey," *Applied Sciences*, vol. 11, no. 15, p. 6790, 2021.

[10] A. Dubey, R. Cammarota, V. Suresh, and A. Aysu, "Guarding machine learning hardware against physical side-channel attacks," *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, vol. 18, no. 3, 2022.

[11] A. Dubey, A. Ahmad, M. A. Pasha, R. Cammarota, and A. Aysu, "Modulonet: Neural networks meet modular arithmetic for efficient hardware masking," *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, vol. 2022, no. 1, pp. 506–556, 2022.

[12] S. Maji, U. Banerjee, S. H. Fuller, and A. P. Chandrakasan, "A threshold-implementation-based neural-network accelerator securing model parameters and inputs against power side-channel attacks," in *2022 IEEE International Solid-State Circuits Conference (ISSCC)*, vol. 65. IEEE, 2022, pp. 518–520.

[13] K. Athanasiou, T. Wahl, A. A. Ding, and Y. Fei, "Masking feedforward neural networks against power analysis attacks," *Proceedings on Privacy Enhancing Technologies*, vol. 2022, no. 1, pp. 501–521, 2022.

[14] D. B. Roy, T. Fritzmann, and G. Sigl, "Efficient hardware/software co-design for post-quantum crypto algorithm SIKE on ARM and RISC-V based microcontrollers," in *IEEE/ACM International Conference On Computer Aided Design, ICCAD 2020, San Diego, CA, USA, November 2-5, 2020*. IEEE, 2020, pp. 35:1–35:9. [Online]. Available: https://doi.org/10.1145/3400302.3415728

[15] B. J. Gilbert Goodwill, J. Jaffe, and P. Rohatgi, "A testing methodology for side-channel resistance validation," 2011, http://csrc.nist.gov/news_events/non-invasive-attack-testing-workshop/papers/08_Goodwill.pdf.

[16] P. Kocher, J. Jaffe, and B. Jun, "Differential power analysis," in *Annual international cryptology conference*. Springer, 1999, pp. 388–397.

[17] Y. Ishai, A. Sahai, and D. Wagner, "Private circuits: Securing hardware against probing attacks," in *Annual International Cryptology Conference*. Springer, 2003, pp. 463–481.

[18] A. Duc, S. Dziembowski, and S. Faust, "Unifying leakage models: from probing attacks to noisy leakage." in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2014, pp. 423–440.

[19] S. Faust, V. Grosso, S. M. D. Pozo, C. Paglialonga, and F. Standaert, "Composable masking schemes in the presence of physical defaults & the robust probing model," *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, vol. 2018, no. 3, pp. 89–120, 2018.

[20] O. Reparaz, B. Bilgin, S. Nikova, B. Gierlichs, and I. Verbauwhede, "Consolidating masking schemes," in *Advances in Cryptology - CRYPTO 2015 - 35th Annual Cryptology Conference, Santa Barbara, CA, USA, August 16-20, 2015, Proceedings, Part I*, ser. Lecture Notes in Computer Science, R. Gennaro and M. Robshaw, Eds., vol. 9215. Springer, 2015, pp. 764–783. [Online]. Available: https://doi.org/10.1007/978-3-662-47989-6_37

[21] S. Mangard, E. Oswald, and T. Popp, *Power analysis attacks: Revealing the secrets of smart cards*. Springer Science & Business Media, 2008, vol. 31.

[22] K. A. Andrew Waterman, "The RISC-V Instruction Set Manual Volume I: Unprivileged ISA," https://github.com/riscv/riscv-isa-manual/releases, 2022.

[23] S. Nikova, C. Rechberger, and V. Rijmen, "Threshold implementations against side-channel attacks and glitches," in *Information and Communications Security, 8th International Conference, ICICS 2006, Raleigh, NC, USA, December 4-7, 2006, Proceedings*, ser. Lecture Notes in Computer Science, vol. 4307. Springer, 2006, pp. 529–545.

[24] H. Groß, S. Mangard, and T. Korak, "Domain-oriented masking: Compact masked hardware implementations with arbitrary protection order," in *Proceedings of the ACM Workshop on Theory of Implementation Security, TIS@CCS 2016 Vienna, Austria, October, 2016*, B. Bilgin, S. Nikova, and V. Rijmen, Eds. ACM, 2016, p. 3.

[25] OpenTitan. (2022) Opentitan. https://github.com/lowrisc/opentitan.

[26] Y. LeCun, P. Haffner, L. Bottou, and Y. Bengio, "Object recognition with gradient-based learning," in *Shape, Contour and Grouping in Computer Vision*, ser. Lecture Notes in Computer Science, D. A. Forsyth, J. L. Mundy, V. D. Gesù, and R. Cipolla, Eds., vol. 1681. Springer, 1999, p. 319. [Online]. Available: https://doi.org/10.1007/3-540-46805-6_19

[27] K. Fukushima, S. Miyake, and T. Ito, "Neocognitron: A neural network model for a mechanism of visual pattern recognition," *IEEE transactions on systems, man, and cybernetics*, no. 5, pp. 826–834, 1983.

[28] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, and Y. Bengio, "Quantized neural networks: Training neural networks with low precision weights and activations," *The Journal of Machine Learning Research*, vol. 18, no. 1, pp. 6869–6898, 2017.

[29] M. Courbariaux, Y. Bengio, and J.-P. David, "Binaryconnect: Training deep neural networks with binary weights during propagations," in *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 2*, ser. NIPS'15. Cambridge, MA, USA: MIT Press, 2015, p. 3123–3131.

[30] L. Deng, G. Li, S. Han, L. Shi, and Y. Xie, "Model compression and hardware acceleration for neural networks: A comprehensive survey," *Proceedings of the IEEE*, vol. 108, no. 4, pp. 485–532, 2020.

[31] Google. (2022) The sequential model. https://www.tensorflow.org/guide/keras/sequential_model.

[32] E. Trichina, D. D. Seta, and L. Germani, "Simplified adaptive multiplicative masking for aes," in *International Workshop on Cryptographic Hardware and Embedded Systems*. Springer, 2002, pp. 187–197.

[33] S. J. Pan and Q. Yang, "A survey on transfer learning," *IEEE Transactions on knowledge and data engineering*, vol. 22, no. 10, pp. 1345–1359, 2009.

[34] L. Goubin, "A sound method for switching between boolean and arithmetic masking," in *International Workshop on Cryptographic Hardware and Embedded Systems*. Springer, 2001, pp. 3–15.

[35] J. D. Golic, "Techniques for Random Masking in Hardware," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 54, no. 2, pp. 291–300, 2007.

[36] M. A. KF, V. Ganesan, R. Bodduna, and C. Rebeiro, "Param: A microprocessor hardened for power side-channel attack resistance," in *2020 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*. IEEE, 2020, pp. 23–34.

[37] K. Ramezanpour, P. Ampadu, and W. Diehl, "Rs-mask: Random space masking as an integrated countermeasure against power and fault analysis," in *2020 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*. IEEE, 2020, pp. 176–187.

[38] M. S. Louis, Z. Azad, L. Delshadtehrani, S. Gupta, P. Warden, V. J. Reddi, and A. Joshi, "Towards deep learning using tensorflow lite on risc-v," in *Third Workshop on Computer Architecture Research with RISC-V (CARRV)*, vol. 1, 2019, p. 6.

[39] N. Wu, T. Jiang, L. Zhang, F. Zhou, and F. Ge, "A reconfigurable convolutional neural network-accelerated coprocessor based on risc-v instruction set," *Electronics*, vol. 9, no. 6, p. 1005, 2020.

[40] A. Reuther, P. Michaleas, M. Jones, V. Gadepally, S. Samsi, and J. Kepner, "Survey of machine learning accelerators," in *2020 IEEE high performance extreme computing conference (HPEC)*. IEEE, 2020, pp. 1–12.