UNIVERSITÄT ZU LÜBECK
INSTITUT FÜR
THEORETISCHE INFORMATIK

Trustworthy AI Hardware Accelerator

Vertrauenswürdiger KI Hardwarebeschleuniger

Masterarbeit

verfasst am
Institut für Technische Informatik

im Rahmen des Studiengangs
IT-Sicherheit
der Universität zu Lübeck

vorgelegt von
Celine Thermann

ausgegeben und betreut von
Prof. Dr. Mladen Berekovic

mit Unterstützung von
Dr. Saleh Mulhem und M. Sc. Ahmed Mahmoudi

Lübeck, den 20. Dezember 2022

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides statt, dass ich diese Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

_____

Celine Thermann

Zusammenfassung

KI Beschleuniger bieten eine Lösung für die Defizite von Allzweck Hardware, wenn es um schnelle und energieeffiziente KI geht. Fault-Injection-Angriffe werden normalerweise der Kryptographie zugeordnet. Diese Angriffe haben aber schon lange ihren Weg in andere Welten gefunde. Ein Beschleuniger, der einem solchen Angriff ausgesetzt ist könnte einen signifikanten Teil seiner Genauikeit einbüßen. Aber nicht nur das, denn ein Angreifer könnte auch wertvolle Informationen extrahieren, indem lediglich ein einziges Bit geflipt wird. Unter Nutzung von SystemC wurde ein einfacher Beschleuniger für Neuronale Netze erstellt, der seine Parameter mittels des SpongeWrap Authenticated Encryption (AE) Schema schützt. Das AE Schema schützt die Integrität und die Vertraulichkeit eines verschlüsselten Parameters and sorgt dafür, dass der Beschleuniger zuverlässig arbeitet. Der Effekt eines einzelnen Bit-Flips in einem kleinen Neuronalen Netz zeigt, warum ein Schutzmechanismus wie SpongeWrap sinnvoll ist.

Abstract

Artificial intelligence accelerators provide a solution to the shortcomings of general-purpose hardware when it comes to being fast and energy-efficient AI. Fault injection attacks are usually reduced to the realm of cryptography, but they have long found their way into other fields. An accelerator succumbing to these attacks could suffer a significant drop in accuracy. But not only that, an attacker might be able to extract valuable information by just flipping a single bit. This thesis implements a countermeasure to bit-flips on the parameters of a neural network. Using SystemC, a simple neural network accelerator was implemented that aims to protect its parameters using the SpongeWrap authenticated encryption scheme. The AE allows for confidentiality and integrity protection of the encrypted parameters and makes sure that the accelerator works reliably. The sensitivity of several small neural networks to a single bit-flip helps to understand the effect a single bit-flip might have if no such protection measure was employed.

Contents

1

Introduction

1.1 Defining Trustworthiness

Dependability and trustworthiness are two terms that are often used to describe very similar things in different disciplines. Defining these terms is not trivial and depends on the person that is asked to define them. In [2] they make an effort to define these related terms. The authors were able to distinguish the understanding of these terms into two categories: dependability of computing systems and dependability of systems engineering, by considering various definitions. While the first category finds its application more with researchers, the second category is the one employed by standards in the engineering field.

The common denominator in dictionary definitions of dependability and trustworthiness is „trust“. This indicates that trust is the deciding component in defining both dependability and trustworthiness. Trust is defined in British English as „to believe that someone is good and honest and will not harm you, or that something is safe and reliable“. On the other side, in American English, it is defined as „to have confidence in something or to believe in someone“ or „to hope and expect that something is true“. From these definitions, the authors of [2] concluded that „trust“ exists in the semantic context and brings with it some attributes that might as well include security. They concluded that dependability and thus also trustworthiness is ensured by the semantic existence of „trust“. It encompasses reliability, safety, maintainability, availability, integrity, and confidentiality. Faults, errors, and failures (FEFs) as well as security vulnerabilities are considered to be impairments in this scenario. This is also the definition used over the course of this thesis.

1.2 Trustworthiness of Artificial Intelligence Accelerators

Artificial intelligence has long spread into various applications. Neural networks in particular, are very popular because of their versatility and effectiveness. Over time it became apparent that neural networks are especially work-intensive and need hardware that can handle the increased requirements in computations. Accelerators have the pur-

pose of lifting these bottlenecks by specializing in fulfilling the specific needs of certain tasks. An attacker might be interested in extracting some data or some accelerated model performing a task that is of interest. These goals are a threat to confidentiality and integrity. If the attacker decides that the destruction of the models output quality during training or inference is the goal, then this makes the model unreliable and thus might also have an effect on the safety of the accelerator.

Within the scope of this thesis, a functional model of a neural network accelerator protecting its parameters using an authenticated encryption scheme is implemented. The authenticated encryption scheme of choice is *SpongeWrap*, a scheme that was introduced by the authors of Keccak[1]. They are best known for providing the basis of the SHA-3 hash function. Because the accelerator aims to enhance the computations of neural networks, it is necessary to introduce them and provide insight into their functionality. This is done extensively in Section 2.1. To provide an overview of neural network accelerators Section 2.2 introduces their advantages and possible ways to accelerate computations and provides some examples for such accelerators. Section 2.3 supplies information on fault injection attacks, which is the attack scheme used. Finally, the SpongeWrap authenticated encryption scheme, and its connection to the sponge construction are described in Section 3. The sponge construction is most famously used for Keccak and with it also for SHA-3.

The following chapter describes the implementation and provides a reflection on the impact the added component has on the trustworthiness of the accelerator. A SystemC virtual prototype first introduced in [12] is extended by two new modules: an accelerator module and a SpongeWrap module. SystemC and its capabilities are briefly introduced in section 4.1. The following section 4.2 describes the virtual prototype, the accelerator module and the SpongeWrap module and their interaction with one another. section 4.3 shows the impact of a single bit-flip on one of the weights and biases of a simple neural network. A reflection on the effect of the SpongeWrap on the trustworthiness of the accelerator module is then provided in section 4.4. Finally, everything is summarized in chapter 5, and an outlook on future work is provided.

[1] https://keccak.team/

2

Neural Network Accelerators

Artificial intelligence (AI), *machine learning* (ML), and *deep learning* (DL) are terms that are sometimes used interchangeably. This is especially the case in popular media. AI has been defined in many different ways over time. While some define it as a machine thinking or acting human-like, others focus on rationality. No matter the definition, AI provides a wide variety of practical algorithms. ML is a sub-field of AI, and DL is a specialized subset of algorithms within the ML domain. ML models and algorithms try to make intelligent decisions through learning. Within this space, DL uses *artificial neural networks* to make complex decisions. They have long been integrated into daily applications such as speech recognition and machine translation.

The rising interest in these applications increased the need for fast, energy-efficient, and scalable computations in AI applications. A growing volume of data caused bottlenecks. Specialized hardware in the form of AI accelerators tries to resolve the shortcomings of general-purpose hardware. [11, 21, 23]

2.1 Neural Networks

Neural Networks (NN) take their inspiration from neuroscience. *Neurons* are the nerve cells transmitting and processing information in the brain. They are arranged into networks of nerves that pass electrical impulses between each other. Even though some computers are technically faster than the human brain, they still do not come close. Human logic is based on experience and cannot be fully described by formal logic methods. Thus, it is not easy to emulate for a computer.

NNs are a type of ML algorithm. These algorithms can generally be divided into three types:

1. **Supervised Learning:** Algorithms that learn a function, which maps an input to an output based on some previously observed input-output pairs. In other words, they try to find an approximate function f such that $y \approx f(x)$ for an input x and its target value y. Supervised learning usually is further divided into two main areas:

  - *Classification:* Methods that try to find a mapping from input data to a discrete label. These labels are often also called classes.
  - *Regression:* Methods mapping the input data to some continuous value.
2. **Unsupervised Learning:** An algorithm learning patterns from its inputs even without getting feedback in the form of target values. Examples of unsupervised learning algorithms are both clustering and dimensionality reduction techniques.
3. **Reinforcement Learning:** Algorithms learning from a series of reinforcements in the form of rewards and punishments. The goal is to maximize the reward received from executed actions, even if the environment is unknown. Reinforcement learning finds its application both in self-driving cars and robots, for example.

NNs can implement all three types of ML learning. Algorithms might not fall uniquely into one of these categories. Sometimes a hybrid approach is necessary if pure supervised, unsupervised and reinforcement learning approaches are not applicable. [1, 23]

Basic Functionality and Architecture

The first recognized work on neural networks was published in 1943 by McCulloch and Pitts [19]. They presented a simple neuron model that fired when exceeding a certain threshold value. NNs have come a long way since then. Their popularity has given rise to powerful frameworks such as Tensorflow[2] and Caffe[3], enabling the creation of various types of networks.

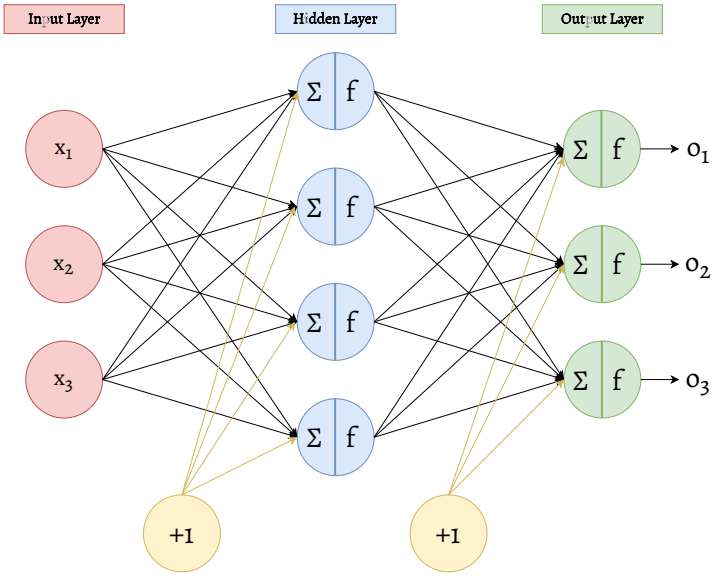


Figure 2.1: Example of a simple Neural Network

An **artificial neuron** (hereafter referred to as just **neuron**), as depicted in Figure 2.2, is the basic computational unit in a neural network. They are typically organized into **layers**. Neurons on consecutive layers are connected by **links**. In its simplest form, neurons

[2] https://www.tensorflow.org/
[3] https://caffe.berkeleyvision.org/

on layer *i* are connected only to neurons on layer *i+1*. This type of neural network is often called *feed-forward neural network*, because data only flows into one direction without running into cycles. This also means that the outputs of neurons are only propagated from neurons on layer *i* to neurons on layer $i + 1$. However, this might not be the case for more complex or specialized NNs. The focus in this chapter however lies on *feed-forward networks*. The **width** of a layer refers to the number of neurons on that layer, while the **depth** of a network describes the number of stacked computation layers. There are three classes of layers:

- **Input Layer:** This is the first layer of the neural network. It represents the input to the neural network and does not perform any computations. This is also why it is often not counted in the number of layers in an NN. The number of neurons on this layer depends on the size of the input presented to the network.
- **Output Layer:** This is the last layer of the neural network. It presents the final output to the user and thus is also the only layer for which the output is entirely observable. The number of neurons on this layer depends on the number of variables to be predicted at once. E.g. in case of a classification task, it would contain as many neurons as there are classes.
- **Hidden Layer:** These are all the layers that are stacked in between the input and output layers. They represent intermediate values in the network. In large parts, the expressiveness of a network is provided by these layers.

Each link between neurons has a **weight** indicating its strength. In practice, the weights between two layers are represented by a **weight matrix** instead of single weights. This matrix can be used to compute the outputs of entire layers in the network more efficiently. In addition to weighted links, each neuron accepts a **bias** weight. This bias can be represented as a link weighing an input that is always +1.

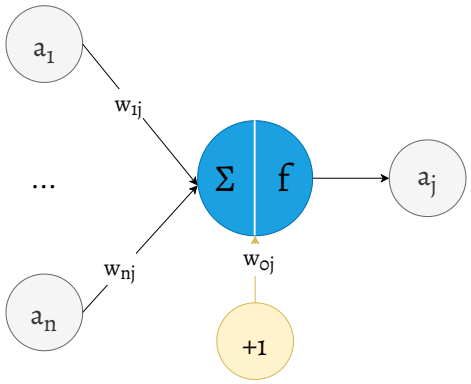


Figure 2.2: Basic structure of a neuron

When receiving some inputs $a_1, \ldots, a_n$ a neuron j first computes the **input function**:

$$\text{in}_j = w_{0j} + \sum_{i=1}^{n} w_{ij} a_i$$

where w_{ij} is the link propagating the output a_i of neuron i to neuron j and w_{0j} is the bias weight of neuron j. Non-linearity is introduced by subsequently applying the neurons **activation function** to derive the output of that neuron:

$$a_j = f(in_j) = f\left(w_{0j} + \sum_{i=1}^{n} w_{ij} a_i\right)$$

This value is often also called **activation**. The activation function can be selected from a number of different functions. An excerpt of the available functions is shown in Table 2.3.

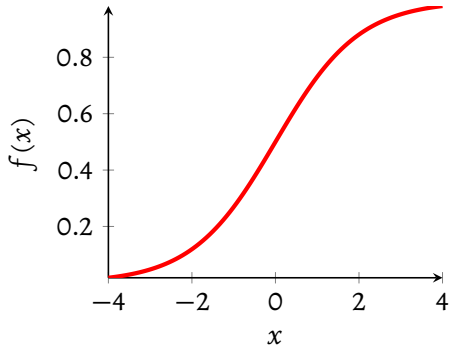
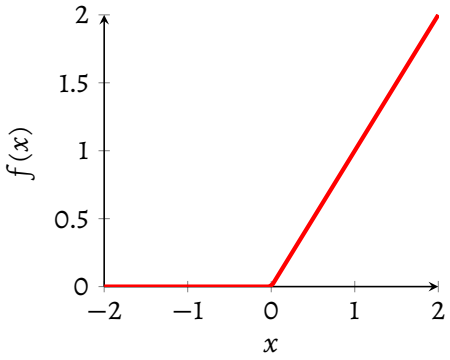
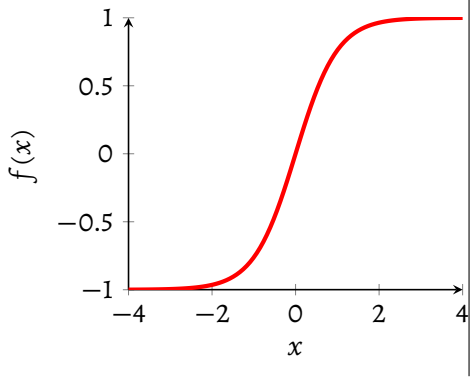
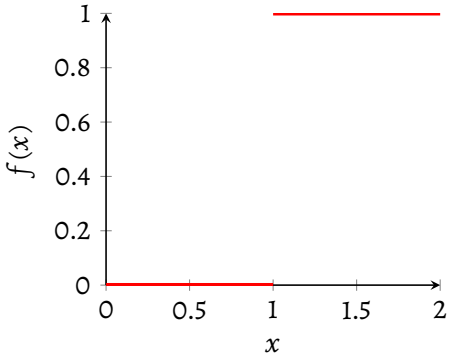
Sigmoid	**Rectified Linear Unit (ReLU)**
$f(x) = \frac{1}{1+e^{-x}}$	$f(x) = \max\{0, x\}$
	
Tanh	**Threshold**
$f(x) = \frac{e^{2x}-1}{e^{2x}+1}$	For $S = 1$ $f(x) = \begin{cases} 1 & , \ x \geq S \\ 0 & , \ other \end{cases}$
	

Table 2.3: Activation functions

Suppose a layer passes on the result of its input function. In that case, this can be considered equivalent to applying the *identity function* $f(x) = x$ instead of one of the (possibly) non-linear functions available. A network using only the linear activation function is not more powerful than a single-layer network. The selection of the correct activation function is usually non-trivial. It should be apparent that all of the functions are centred around zero. The bias parameter serves the purpose of shifting the function to the left or the right.

Many activation functions operate on a single value. This is not the case for the *softmax* function. This function is special in that it operates on a vector $\mathbf{x} = (x_1, \dots, x_n)^T$ instead of a single value:

$$f(\mathbf{x})_i = \frac{\exp x_i}{\sum_{j=1}^{n} \exp x_j} \quad \forall i \in \{1, \dots, n\}$$

The result is a mapping of n values to n probabilities. All entries of the output vector sum up to 1. It is mostly used on the output layer of networks and has some additional properties that make it a popular choice.

NNs are usually represented by computational graphs. An example neural network is shown in Figure 2.1. Circles of different colors act as neurons belonging to a specific layer (indicated above the stacked circles in the same color), while the links between these circles are arrows. Each yellow link is a bias, weighing the constant indicated as a dummy value in the yellow circles. The network is considered to have a depth of two layers (the hidden layer and the output layer) with a width of 3 for both the input layer and the output layer and a width of 4 for the hidden layer. A deep feed-forward network as displayed in Figure 2.1 is also called *multilayer perceptron* (MLP). The actual power of neural networks lies in using multiple layers that make it possible to represent very complex functions. In general, any multilayer NN can be called deep if it has many layers. [1, 11, 23]

Learning in Neural Networks

An NN is capable of learning from examples provided in the form of datasets. There are a lot of freely available datasets e.g., MNIST[4]. Let D be such a dataset containing N examples. Each of these examples $x = x_1, \dots, x_m$ has m elements or attributes. In the case of supervised learning, x is accompanied by a matching label y indicating the correct target value for that example. The dataset can be divided into multiple partial datasets that can be used for different purposes during the training process:

- **Training dataset:** This dataset is used during the training of the network.
- **Validation dataset:** This dataset is used for fine-tuning during the training process. This can include the selection of the model, as well as fine-tuning of fixed parameters called **hyperparameters**, used during training. A validation dataset is not a requirement for the training, even for supervised training algorithms. An example of a validation technique is **cross-validation**. This technique first divides the training data into k parts. One of the segments is kept back for validation, while the other parts are used for training. The process is then repeated k times, such that each of the parts has been used as the validation set once.
- **Test dataset:** This dataset is used to measure the performance of the network after training. The examples in this dataset are distinct from the examples included in the training dataset.

[4] http://yann.lecun.com/exdb/mnist/

An indicator of the quality of a neural network is the **accuracy**, i.e. the proportion of examples for which the model is correct to the number of samples the model fails to predict the correct labels. After finishing the training, the accuracy over the test data is computed to give an impression of how well the network is performing. Since it's necessary to know the correct values for the examples, this measure can only be computed for supervised learning algorithms. For some unsupervised algorithms, there is the possibility of computing a different quality measure. E.g. dimensionality reduction techniques try to minimize the error between the input data and the reconstructed data.

Generalization is the ability of an ML model to perform well on unseen data. An NN might adapt to the training data too well, causing it to perform poorly on new samples. This problem is called **overfitting**. The network might also have the opposite problem. Meaning it is not able to perform well on the training data and thus is also unable to generalize well. This problem is called **underfitting**.

To measure how well a network is generalizing during the training, a **loss function** is computed. This function penalizes mismatches between the target value and the predicted value. Different loss functions must be considered. However, not all loss functions can be used for all applications. For example, there is a need to differentiate between a loss function for a regression problem and a loss function for a classification problem. A suitable candidate for a regression problem is the *mean-squared error*:

$$MSE = \frac{1}{r}\sum_{i=1}^{r}(y_i - \hat{y}_i)^2$$

For the targeted outputs $y_1, \dots, y_r$ and the predicted outputs $\hat{y}_1, \dots, \hat{y}_r$ the mean-squared loss is calculated over r samples. Classification problems often make use of the *cross-entropy* loss function:

$$CE = -\sum_{i=1}^{k} y_i \log(\hat{y}_i)$$

In this case, the loss is not computed over a number of samples but over all the possible k classes that might be output by the model. Now let $y_1, \dots, y_k \in \{0, 1\}$ be the correct one-hot encoded class vector for a sample and let $\hat{y}_1, \dots, \hat{y}_k \in [0, 1]$ be the probabilities for the outputs of the network. Since it uses the negative logarithm, a smaller probability output by the network for the correct class will result in a larger value and thus a larger loss.

There are many different kinds of neural network architectures. In addition to everything described in this chapter, there are variations that haven't been touched. This includes different layouts of neurons and links, not only connecting adjacent layers. Some examples of these networks would be the following:

- *Recurrent NNs:* This is a type of NN with feedback connections. This makes it possible to take previous inputs into consideration. These types of networks also support short-term memory. For that reason, they are often used for text processing.
- *Convolutional NNs:* This is a type of NN that specializes in processing data with a known grid-like structure. They use a special type of layer called a *convolutional layer*

that executes a convolution operation on the data. This type of NN is commonly used to process images.

Depending on the application, some types of NNs are better suited to solve a given task. This often depends on the type of data to be processed. [1, 11, 23]

Gradient-Based Learning

Optimization techniques aim to minimize the loss function by adjusting the parameters of the NN. Which optimization algorithm is used depends on the model. The most common algorithm used to optimize NNs is the **gradient descent** algorithm.

The gradient descent algorithm uses the gradient of the loss function with respect to the weight and bias parameters. After initializing the model and creating its **parameter space w**, it is evaluated on the training data. The algorithm then tries to move downhill in the loss function as shown in Algorithm 1:

Algorithm 1 Gradient descent algorithm

Input: Parameter space **w**
  loop until convergence **do**
    for each w_i **in w do**
      $w_i \leftarrow w_i - \alpha \frac{\delta}{\delta w_i} Loss(\mathbf{w})$
    end for
  end loop

The parameter α is called the **learning rate** or **step size**. It is one of the hyperparameters that can be set for a model. This hyperparameter can be set either to a constant value or decays over time. A too-small learning rate will cause slower training. On the other hand, a learning rate that's too large can cause the training loss to increase because the steps are too large.

Depending on the model, computing the gradient of the loss function with respect to a weight or a bias is not trivial. One possibility is the formulation of a closed-form expression for the loss function and calculating the derivatives from there. Still, this would lead to increasingly difficult expressions and thus is only possible for small enough NNs. Instead of using the complete derivatives of the closed-form expression, the **backpropagation algorithm** computes partial derivatives. Using both gradient decent and backpropagation, the following steps can be used to optimize an NN:

1. **Forward Phase:** This phase calculates the outputs for some inputs. Afterwards, the loss for these inputs is computed.
2. **Backward Phase:** This phase computes the gradients of the loss function with respect to the weights and biases.
3. **Update:** The update of the weights and biases in the negative direction of the calculated gradients. This is achieved in the manner of the gradient descent algorithm.

These steps are repeated until the model converges. A single cycle of this process is called an **epoch**. After the training is finished, the model performs only the forward step to provide **inference** results. For classification problems, this would provide the class predicted by the NN.

Computing the derivative of a function composition requires the use of the *chain rule*. The simplest version of this rule is the **univariant chain rule**. This rule calculates the derivative of the composition of a function $f(x)$ with input $g(x)$:

$$\frac{\delta f(g(x))}{\delta x} = \frac{\delta f(g(x))}{\delta g(x)} \cdot \frac{\delta g(x)}{\delta x}$$

The terms on the right side both compute a *local* gradients. The derivatives are computed with respect to the intermediate values received by the function. In the context of NNs, both f and g would be the functions computed by a neuron. Meaning calculating the input function and applying its activation function to the result. The univariant chain rule assumes that a single neuron is connected only to one other neuron. However, this is not the case most of the time. The univariant chain rule can be generalized, such that the case that a neuron is connected to k other neurons can be modeled. This rule is called the **multivariant chain rule**:

$$\frac{\delta f(g_1(x), \dots, g_k(x))}{\delta x} = \sum_{i=1}^{k} \frac{\delta f(g_1(x), \dots, g_k(x))}{\delta g_i(x)} \cdot \frac{\delta g_i(x)}{\delta x}$$

In most cases, it is possible to find multiple paths from one neuron to another. If this is the case, the sum of derivatives over these paths can be computed. Let $y(i)$ be the function computed by the i-th neuron and let $y(j)$ be the function computed by the j-th neuron. The local derivative for a link from neuron i to neuron j is defined as $z(i,j) = \frac{\delta y(j)}{\delta y(i)}$. For a set of paths $\mathcal{P}$ where $|\mathcal{P}| > 0$ from a source neuron s to a sink neuron t the gradient $\frac{\delta y(t)}{\delta y(s)}$ can be computed as:

$$S(s,t) = \frac{\delta y(t)}{\delta y(s)} = \sum_{P \in \mathcal{P}} \prod_{(i,j) \in P} z(i,j)$$

This takes the impact of not directly connected neurons on each other into consideration. Intuitively, path aggregation can be used to create a simple algorithm that calculates the derivatives for an output neuron with respect to another neuron in the network. As the paths are not disjoint for the most part, some derivatives will be unnecessarily recomputed multiple times. In addition, an NN can contain a large number of paths between neurons. Subsequently, this algorithm would have exponential runtime. It can be improved by using dynamic programming. Let $A(i)$ be the set of successors of a neuron i on the next layer. To avoid unnecessary re-computations of derivatives, the following update can be used in a dynamic programming algorithm:

$$S(i,t) \Leftarrow \sum_{j \in A(i)} S(j,t)s(i,j)$$

Using this update rule, the dynamic programming algorithm shown in Algorithm 2 can be used to calculate the derivatives between neurons in an NN.

Algorithm 2 Dynamic programming algorithm for neuron-to-neuron derivatives

Initialize $S(t,t) = 1$
repeat
Select an unprocessed neuron i such that the value of $S(j,t)$ is available for all its outgoing neurons $j \in A(i)$
Update $S(i,t) \Leftarrow \sum_{j \in A(i)} S(j,t)s(i,j)$
until all nodes have been selected

The derivatives until now have all been computed with respect to the neurons themselves. However, what actually is needed are the derivatives of the loss function with respect to the weights and biases. Let $t_1, \dots, t_p$ be the output neurons and $y(t_1), \dots, y(t_p)$ the outputs of these neurons. The loss function is denoted as $L(y(t_1), \dots, y(t_p))$. Derivatives of the loss function with respect to a weight w_{ij} can be computed as follows:

$$\begin{aligned}\frac{\delta L(y(t_1),\dots,y(t_p))}{\delta w_{ij}} &= \left[\frac{\delta L(y(t_1),\dots,y(t_p))}{\delta y(j)}\right]\frac{\delta y(j)}{\delta w_{ij}} \\ &= \left[\sum_{k=1}^{p}\frac{\delta L(y(t_1),\dots,y(t_p))}{\delta y(t_k)}\frac{\delta y(t_k)}{\delta y(j)}\right]\frac{\delta y(j)}{\delta w_{ij}}\end{aligned}$$

Let $\Delta(i) = \frac{\delta L(y(t_1),\dots,y(t_p))}{\delta y(i)}$ be the derivative of the loss function with respect to neuron i. The dynamic programming algorithm presented in Algorithm 2 is modified such that Algorithm 3 is used to calculate the loss-to-weight derivatives.

Algorithm 3 Dynamic programming algorithm for loss-to-weight derivatives

Initialize $\Delta(t_r) = \frac{\delta L(y(t_1),\dots,y(t_p))}{\delta y(t_k)}$ for each $k \in \{1, \dots, p\}$
repeat
Select an unprocessed node i such that the value of $\Delta(j)$ is available for all it's outgoing neurons $j \in A(i)$
Update $\Delta(i) \Leftarrow \sum_{j \in A(i)} \Delta(j)z(i,j)$
until all nodes have been selected
for each edge (i,j) with weight w_{ij} **do** compute $\frac{\delta L(y(t_1),\dots,y(t_p))}{\delta w_{ij}} = \Delta(j)\frac{\delta y(j)}{\delta w_{ij}}$

Subsequently, the classic gradient descent algorithm is applied to update the weights and biases. This approach is even further improved by using vectors and matrices instead of single values. Let $\mathbf{v} = [v_1, \dots, v_d]^T$ and $\mathbf{h} = [h_1, \dots, h_m]^T$ be vectors that appear somewhere in the NN. They could e.g. be a vector of losses or activations of a layer. The derivative from one vector to another is the transposed **Jacobian** matrix:

$$\frac{\delta \mathbf{h}}{\delta \mathbf{v}} = \text{Jacobian}(\mathbf{h}, \mathbf{v})^T$$

The entry $(\frac{\delta \mathbf{h}}{\delta \mathbf{v}})_{ij}$ is $\frac{\delta h_j}{\delta v_i}$ in the $d \times m$ matrix. Now let $\mathbf{v}$ be the outcome of some layer in the network and let $\mathbf{h}$ be the output of the network. Then the derivative of the loss function

with respect to $\mathbf{v}$ can be calculated as:

$$\frac{\delta L(y(t_1),\dots,y(t_p))}{\delta \mathbf{v}} = \frac{\delta \mathbf{h}}{\delta \mathbf{v}} \frac{\delta L(y(t_1),\dots,y(t_p))}{\delta \mathbf{h}} = \text{Jacobian}(\mathbf{h},\mathbf{v})^T \frac{\delta L(y(t_1),\dots,y(t_p))}{\delta \mathbf{h}}$$

If $\mathbf{g}_i$ and $\mathbf{g}_{i+1}$ are the loss gradients of the i-th and $i+1$-th layer and J_i the Jacobian matrix between i-th and $i+1$-th layer, then g_i can be calculated as:

$$\mathbf{g}_i = J_i^T \mathbf{g}_{i+1}$$

Each layer in the network is split into a *linear layer* and an *activation layer* to simplify the gradient computations. The linear layer executes an all-to-one computation by multiplying the weight matrix W and the output of the previous layer. On the other side, the activation layer performs a one-to-one computation using f for each of the vector entries. If $\mathbf{z}_i$, $\mathbf{z}_{i+1}$ and $\mathbf{z}_{i+2}$ are the outputs of layer i and $i+1$ and $i+2$ in the decoupled form respectively, then the forward and backward operations of the linear and activation layer can be described as:

	Linear layer	**Activation layer**
Forward	$\mathbf{z}_{i+1} = W\mathbf{z}_i$	$\mathbf{z}_{i+2} = f(\mathbf{z}_{i+1})$
Backward	$\mathbf{g}_i = W\mathbf{g}_{i+1}$	$\mathbf{g}_{i+1} = \mathbf{g}_{i+2} \odot f'(\mathbf{z}_{i+1})$

Table 2.4: Forward and backward phase for the linear and activation layer

The $\odot$ symbol denotes the element-wise multiplication. Since these computations only calculate the loss-to-neuron gradients, it is still necessary to find the loss-to-weight gradients. In the end, the gradients with respect to the weights are the following:

$$M = \mathbf{g}_i \mathbf{z}_{i+1}^T$$

These gradients are used to update the parameters of the NN. However, optimizing these parameters is a complex problem. There are some challenges that might be encountered:

- **Local minima:** The loss function is not convex in most cases. If that were the case, there would be a single minimum. However, if not, then there are possibly multiple minima that are not the global minimum. It might be difficult for the optimization algorithm to leave a local minimum to find a better model with a lower loss. The local information is not sufficient to escape.
- **Flat regions:** Those with a zero gradient. This could cause the optimization algorithm to get stuck because the weights are not being adjusted.
- **Exploding gradient:** The gradients calculated by the optimization algorithm might get extremely large. This would cause the model to become unstable because the updates are equally as large. [1, 11, 23]

Regularization and Optimizing Training

In addition to the depth and width of the layers, learning rate and the other modifiable parameters mentioned above, there are techniques that help to improve an NNs performance. There are different methods:

- **Early Stopping:** Stop the training once the error on the validation set starts to increase.
- **Dropout:** Some neurons are disabled with a certain probability. This ensures that parts of the model are trained, which would not have been trained if the neurons were enabled.
- **Parameter norm penalties:** Limit the capacity of the network by penalizing the parameters.
  - **L2 Regularization:** Add a penalty to the loss function in the form of the L2 norm:

$$\overline{Loss}(\mathbf{w}) = Loss(\mathbf{w}) + \frac{\alpha}{2}||\mathbf{W}||_2^2 = Loss(\mathbf{w}) + \frac{\alpha}{2}\sum_{w\in\mathbf{w}} w^2$$

    The α parameter adjusts the relative contribution of the additional term. As a result of adding this penalty term the data will appear to have a higher variance and the optimization algorithm shrinks the weights because of this.
  - **L1 Regularization:** Add a penalty to the loss function in the form of L1 norm:

$$\overline{Loss}(\mathbf{w}) = Loss(\mathbf{w}) + \alpha||\mathbf{w}|| = Loss(\mathbf{w}) + \alpha\sum_{w\in\mathbf{w}} |w|$$

    Again, the α parameter adjusts the relative contribution of the additional term. The L1 penalty tends to create a more sparse model, i.e. a model in which more parameters that are 0.

These methods are only a small selection. In addition to improving the performance of the network, the time it takes to train a network can also be enhanced. The classic gradient descent algorithm calculates the loss over the complete training dataset. Algorithms that do this are called **batch** algorithms. Hence the complete dataset has to be loaded for every epoch. This can slow down the training immensely. Under the assumption that the average loss of multiple subsets of examples is a good estimator of the actual loss, a different approach can be used. A **minibatch** algorithm uses a randomly selected subset of the training data and averages the loss after all examples have been passed through the network. This ensures that not all examples have to be kept in memory the whole time. The hyperparameter describing the size of batches is often called **batch size**. Larger batch sizes will ensure a better estimation of the actual loss. **Stochastic gradient descent** is another variation of the classic gradient descent, that uses only one example at a time. [11]

As mentioned before, there are algorithms that can adjust the learning rate over the course of the training. Choosing a suitable learning rate can improve the model's performance significantly. Some popular choices for these algorithms are *AdaGrad*[8] and *Adam*[15].

2.2 Neural Network Accelerators

A lot of focus in AI often falls on the improvement of existing algorithms. However, it is not all about the algorithms. The computations needed for NNs in particular are very expensive. General-purpose processors are often quite limited because they try to cater

to a wide variety of tasks. Instead of using hardware applicable for many applications, **accelerators** aim to provide services for a more narrow set of tasks instead. They come in the form of an additional component that is connected to a host.

Because they do not need to cater to every possible application, accelerators are able to use techniques general-purpose processors can't use. This allows for more exotic and simpler designs that take the specific needs of systems with resource constraints or other requirements into consideration. Components that would otherwise consume a lot of energy can be omitted, while operations specific to the task can be optimized. Advantages that might arise from the use of accelerators are:

- Higher throughput
- Lower latency
- Better energy efficiency
- Increased parallelism
- More efficient use of the memory hierarchy
- Reduced number of cycles per operations
- Lower fetch and decode overhead

Towards these goals, a lot of different accelerators have been presented over time. The landscape of accelerators is very heterogeneous. Getting an overview of all available options is not easy. Still, the most common type of accelerator used in the context of NNs is a GPU. They are especially popular because they are well-suited for performing matrix multiplications. As presented in Section 2.1 NNs make use of matrix multiplication both during the forward and backward passes. Using GPUs for these computations is an obvious choice.

GPUs are only one of the possible types of accelerators. There are those that are closer to general-purpose processors and others that are closer to application-specific integrated circuits (ASIC). While field-programmable gate arrays (FPGA) are closer to ASIC designs, GPUs are much closer to the general-purpose side. That said, an architecture might use both general-purpose and specialized components that have been specifically designed to meet the requirements of the accelerated application. The selection of the correct accelerator depends not only on the task, but also on the available hardware and the desired ecosystem of the accelerator.

NNs have a high computational complexity that increases with the number of parameters in a network. The Multiply-and-accumulate (MAC) operations commonly used for these computations have the potential to be parallelized. In other words, the matrix multiplications can be accelerated amongst other optimizations.

Some of them can only be applied to certain types of layers e.g. convolution layers. Others try to apply computational transformations to reduce the number of expensive multiplications. But even though the number of multiplications can be reduced, there are still a lot of them left. Each of them needs some weights or other data from the memory. This can present a bottleneck in the sense, that data might be repeatedly loaded from energy-

hungry and slow memory. The three accesses to memory for each MAC can accumulate over time and thus limit the improvements to be achieved.

Reading and writing to and from fast and energy-efficient memory is desirable. A memory hierarchy made up of several levels of memory with different energy costs yields the possibility to reduce energy consumption. Some accelerators can leverage this hierarchy by optimizing the data flow through the levels of the hierarchy. The optimized dataflow attempts to minimize the accesses to less energy-efficient memory by reusing data in cheap memory as much as possible. Some approaches also use processing-in-memory to accelerate computations.

The measures presented up until now aim to reduce the complexity of the operations to be executed for an NN. However, optimization can begin at model creation. Quantization of the parameters, i.e. mapping the parameters to a lower precision, reduces the storage costs and can lower the computation requirements. If the model at hand is sparse, then the parameters can be compressed. In addition, the hardware can be modified such that zero values are skipped during computation. Using the same weight in multiple places or completely removing weights that are close to zero may be beneficial. Some of these techniques however can affect the accuracy of the gradients and might have to be avoided during the training process to ensure a well-trained model. There are many other more or less complex techniques to achieve the desired performance.

Accelerators have many more fields of application. Some of them are cryptography, graphics and multimedia. They each have their specialized accelerators. Just because they are well-suited for specific tasks doesn't mean that they can't be applied to a different task. One of these cases is GPUs that are both used to accelerate graphic computations but are incidentally also useful for the matrix computations in NNs. While NVIDIA GPUs are a popular choice, there are others like Intel Habana Gaudi[5] or Googles TPU[6]. In addition to these industry-made accelerators, there are also a lot that was published through academia. [22, 25]

2.3 Fault Injection Attacks: Security and Reliability Threat

Fault injection attacks are best known in the field of cryptography, where they target some kind of secret used in a cryptographic context. With the rise of AI and with it NNs, they found another application. In comparison to side-channel analysis where an attacker acts passively, these type of attacks needs the attacker to actively act upon the target. This means that an attacker tries to observe some faulty behavior by passing faulty data to the model or by trying to otherwise change the state in such a way that it deviates from the specified behavior. Usually, the system under attack is considered to be in the possession of the attacker. However remote attacks on instances that, e.g. deployed in a

[5] https://habana.ai/
[6] https://cloud.google.com/tpu

cloud are also possible.

When attacking cryptographic hardware there usually is only the key to be protected. However, this is not the case for NNs. There are different motivations an attacker might have for an attack:

- *Training data extraction:* The attacker tries to extract the training data from the network. Neural networks are data-hungry and need a large dataset for training. Collecting enough data is both time intensive and expensive. It might also include confidential data that should not get into the hand of a third party. This would, e.g. be the case for biometric data that an attacker could use to unlock systems they are not supposed to have access to. During training, the attacker could try to influence the responses of the network by faulting the training data. An example of attacks aiming to extract the training data is the membership inference attack[24].
- *Structure and parameter extraction:* The attacker tries to extract the structure and/or the parameters of the NN. As mentioned before, training an NN needs a lot of data. If an attacker does not have access to a large dataset, then stealing a model might be attractive. This theft would make it possible to clone the model and produce counterfeits. An example for these types of attacks is the SNIFF[7] attack on the softmax output layer of a neural network.
- *Disrupting training and classification output:* NNs are deployed in an increasingly large number of applications. Disruption could cause serious damage when targeting the a model. In the case of autonomous cars such an attack could even cost human lives. The attacker tries to either mislead the model during training or inference. Examples for these types of attacks would be the single bias attack (SBA) or the gradient descent attack (GDA)[17].

Attacks on cryptographic assets and AI assets differ quite a lot. According to Kerckhoff's Principle, the security of a device should not depend on the secrecy of the implementation. For cryptography, the attacker is assumed to know the implementation. Yet even with this knowledge, the adversary should not be capable to recover the key. This is not the case for NNs, since the structure and parameters are also included in the assets that are to be protected. In addition, the recovery of a cryptographic key has to be exact since otherwise decrypting, signing, or other operations will not be successful. The extraction for NNs can also be imperfect.

Different networks can react very differently to attacks. Even though they can be fault tolerant, this highly depends on the network itself. While larger NNs might not be affected by faults, this might not be the case for smaller networks. As a result, it can not be generally said that neural networks are fault tolerant. Injecting faults is possible in any of the layers of a network. However, some layers are more popular than others. E.g. the SNIFF targets the last layer of an NN in case it uses the softmax activation function on that layer. The output layer in particular makes a good target for attackers trying to cause misclassifications. In general, some areas of attack include the

- *Input to the network:* A faulty input can trick the network into misclassifying an input.

- *Input to a neuron:* The input to a neuron can be faulted in two places. First is the output of a neuron on the previous layer, affecting all the computations that use this output. Second is the input of a specific neuron. In this case, the fault influences only the computations in that specific neuron.
- *Weights:* A fault in a weight affects only the computation of the neuron that uses the faulty weight.
- *Bias:* A fault in a bias only affects the computation of the neurons it belongs to. Since weights are always multiplied with an input to the neuron, the change caused by a fault in the bias should normally be smaller than a change in a comparable weight.
- *Product:* Inject a fault into the product of a weight and input. This also affects only the neuron computing the product. A fault can also cause a large change in the input function of the affected neuron.
- *Summation:* Injecting a fault into the summation computed by a neuron.
- *Activation function:* Inject a fault during the computation of the activation function. It has been shown that, e.g. ReLU activation can be influenced by fault such that the output is always zero when injecting fault on an instruction level.

Faults can be injected into hardware by, e.g. lasers, reduced voltage, clock glitches and rowhammer attacks. However, it should be noted that faults also occur naturally from aging, thermal issues or cosmic rays. It can be difficult to secure a system against fault injection attacks. It is possible to monitor both the voltage and the clock. As larger models are more fault tolerant, it might also be an option to increase the number of neurons used in the network to provide redundancy. In case the weights and biases are faulted while in memory, checks for faults using a check bit or other integrity-checking techniques are useful. This is also the approach chosen for the technique presented later on. [7, 26, 28]

Fault injection attacks are usually exclusively viewed as a security threat. However, this is not the case anymore. They are both a security and reliability threat. An attacker trying to extract either the training data, the structure or the parameters might threaten both the confidentiality and integrity of the model and the training data. On the other side, the disruption of the training and classification output is a threat to the reliability. A certain degradation in the quality of the outputs will make it impossible for the output to solve the task accurately. This is especially concerning if the NN, e.g. in the form of an NN accelerator, is deployed in a safety-critical context. Reliability and security are interconnected. An unreliable accelerator might give rise to faults that compromise the security. On the other hand, an insecure accelerator yields the possibility of degrading the reliability.

3

Authenticated Encryption using the Duplex Construction

The sponge construction is most well-known for its use in *Keccak*, which was standardized as *SHA-3* [10] by *NIST* in 2015. However, the construction is much more versatile and has possible applications in message authentication, stream encryption, and authenticated encryption. Using a fixed permutation has advantages and limitations compared to the popularly utilized block ciphers.

The duplex construction is related to the sponge construction. It enables alternating absorbing and squeezing operations while retaining the same security as the sponge construction. This makes it possible to create a single-pass authenticated encryption scheme. [4, 5]

3.1 Authenticated Encryption

Authenticated encryption (AE) schemes aim to provide both *privacy* and *authenticity*. The authenticity may depend not only on the message but also on some optional data A (often called **header** or **associated data**). Unlike the **message** M, the header remains unencrypted and is transmitted like this alongside the encrypted data. Such schemes can often be found under the term *Authenticated Encryption with Associated Data* (AEAD).

There are several ways to construct such algorithms. In the most general case, they can be constructed through generic composition of an *encryption scheme* and an *authentication scheme*. Depending on the order of application of these building blocks, different approaches provide a different level of security.

An encryption algorithm *Enc* gets key K_{Enc} and the message M to generate the ciphertext C, while the authentication algorithm *Auth* gets key K_{Auth} and either M or C to produce a tag T. The following three approaches to AE using separate encryption and authentication schemes can be defined:

– **Authenticate-then-Encrypt** (AtE): AtE first computes the tag T on the plaintext M and

then encrypts M to get the ciphertext C. Using *Enc* and *Auth* this can be described as $C = Enc(K_{Enc}, P||Auth(K_{Auth}, P))$. AtE is e.g. used by *SSL*.
- **Encrypt-then-Authenticate** (EtA): EtA first encrypts the plaintext M to get the ciphertext C and then calculates the tag T on C. Using *Enc* and *Auth* this can be denoted as $(C, T) = (Enc(K_{Enc}, P), Auth(K_{Auth}, Enc(K_{Enc}, P))$. EtA is used, e.g. by *IPsec*.
- **Encrypt-and-Authenticate** (EaA): EaA first encrypts the plaintext M to get the ciphertext C. The tag T is calculated separately on M. Using *Enc* and *Auth* this can be described as $(C, T) = (Enc(K_{Enc}, P), Auth(K_{Auth}, P))$. EaA is used, e.g. by *SSH*.

Of these three approaches, only EtA is generally secure, as presented in [16]. For each of them, an optional header A can be added to get an AEAD scheme. Alternatively to the generic composition, there are dedicated schemes like GCM [9] that are also standardized by NIST, that provide AE. The AE scheme that is used over the course of this work falls into the category of dedicated schemes. For that, two algorithms have to be defined that are used later on:

- **Wrapping algorithm** W: The wrapping algorithm receives the key K, the header A, and the message M. It produces the ciphertext C and the tag T. This can be denoted as $W(K, A, M) = (C, T)$. The authentication is performed over both A and M. Thus, the integrity of both the encrypted and unencrypted data can be ensured.
- **Unwrapping algorithm** U: The unwrapping algorithm receives the key K, the header A, the ciphertext C and the tag T. It produces either the plaintext M or a failure. Only if the authentication is successful will M be returned. A failure indicates that a forgery or fault was detected and thus the authentication failed. A successful decryption is denoted by $U(K, A, C, T) = M$.

In literature, these algorithms are often described by the terms *encryption* and *decryption*. However, since the paper introducing the duplex construction [5], the main component of the AE scheme to be introduced, uses the wrap and unwrap terminology, these terms are used continuing from this point onward.

W and U are assumed to be deterministic. Meaning that for $(A, M) = (A', M')$ the output will be the same if the same key is used to wrap both inputs. To counter this, a nonce can be included in the header A.

In this context, the security of authenticated encryption schemes can be defined by the advantage of an attacker concerning the two security goals to be achieved:

- *Privacy:* Without knowledge of the key, the output of the wrapping function looks random. This means an attacker is not able to differentiate between the output of the wrapping function and a random output. For an attacker $\mathcal{A}$ the privacy requirement can formally be described as:

$$\mathrm{Adv}^{\mathrm{priv}}(\mathcal{A}) = \left| Pr[K \xleftarrow{\$} \mathbb{Z}_2^k : \mathcal{A}[W(K, \cdot, \cdot)] = 1] - Pr[\mathcal{A}[R(\cdot, \cdot)] = 1] \right|$$

  Let $K \xleftarrow{\$} \mathbb{Z}_2^k$ be the random choice of the key K from the set of keys of length k and let $R(\overline{A, M})$ be the output of a random oracle for a sequence of header-message pairs.

The attacker tries to distinguish the output of W with key K from the output of a random oracle R. Since a cryptographic algorithm is not truly random, there will always be a way to attack the algorithm, even if a brute-force search is necessary.

– *Authenticity*: The probability that an attacker can successfully create a valid forgery. For an attacker $\mathcal{A}$ the authenticity requirement can formally be described as:

$$\text{Adv}^{\text{auth}}(\mathcal{A}) = Pr[K \xleftarrow{\$} \mathbb{Z}_2^k : \mathcal{A}[W(K,\cdot,\cdot)] \text{ creates a forgery}]$$

Let $K \xleftarrow{\$} \mathbb{Z}_2^k$ be the random choice of the key K from the set of keys of length k. The attacker receives the output of W with key K and tries to forge a valid ciphertext-tag pair. The advantage describes the probability that the attacker is actually capable of doing so.

In an ideal system, two random oracles $\mathcal{RO}_C$ and $\mathcal{RO}_T$ are used to generate the key stream and the tag. This way, sequences of key streams do not reveal any information about tags and vice versa. The two random oracles can be built from a single oracle using domain separation. Now let RO_{WRAP} be this ideal system in the general case, where a sequence of header and message pairs are encrypted and authenticated sequentially. For this idealized system, the advantages of the attacker are as follows:

$$\text{Adv}^{\text{priv}}_{\text{ROWRAP}}(\mathcal{A}) \leq q2^{-k} \text{ and } \text{Adv}^{\text{auth}}_{\text{ROWRAP}}(\mathcal{A}) \leq q2^{-k} + 2^{-t}$$

The attacker can make at most q queries for a key K of length k and a tag T of length t. In addition, it is assumed that every (A, M) is a nonce. Since only q queries are allowed, the probability that an attacker will guess the correct key for a query is 2^{-k}. For the authenticity, the attacker also needs to guess the correct tag. Since the tag has length t, the probability for a correct guess is 2^{-t}. For a more detailed proof, please refer to [5, 4, 3].

3.2 Sponge Construction

The sponge construction was introduced in [6]. It receives an arbitrary length input and applies a **permutation** f to its state of **state size** b and a **padding rule** *pad* with **rate** r. Only the first r bits are directly affected by the input, while the $c = b - r$ last bits, called the **capacity**, are never directly affected. The padding rule has to fulfil some conditions for the sponge construction to be secure. A padding rule is called *sponge-compliant* if it never results in the empty string and if it fulfils the following condition:

$$\forall n \geq 0, \forall M, M' \in \mathbf{Z}_2^* : M \neq M' \Rightarrow M||pad(M,r) \neq M'||pad(M',r)||0^{nr}$$

$pad(M,r)$ generates the padding of M to a multiple of r bits. An example of such a padding rule is the *multi-rate padding rule*. This rule first appends a 1-bit, then a number of 0 bits and a single 1-bit, such that the resulting padded message has a length that is a multiple of the rate.

It has been shown that c determines the security level that can be achieved for a sponge instance. The sponge construction is illustrated in Figure 3.1. After initializing the state

to zero, the input is padded using the padding function *pad* and cut into r-bit blocks. Then the construction proceeds in two stages:

1. **Absorbing:** The construction updates the state by first XORing an r-bit input-block into the state and then applying the permutation function f until all input blocks have been processed.
2. **Squeezing:** The first r bits of the state are returned and the state is updated by applying f. This continues until the desired output length is reached. Depending on the requested amount of bits, the output might have to be truncated to the correct length.

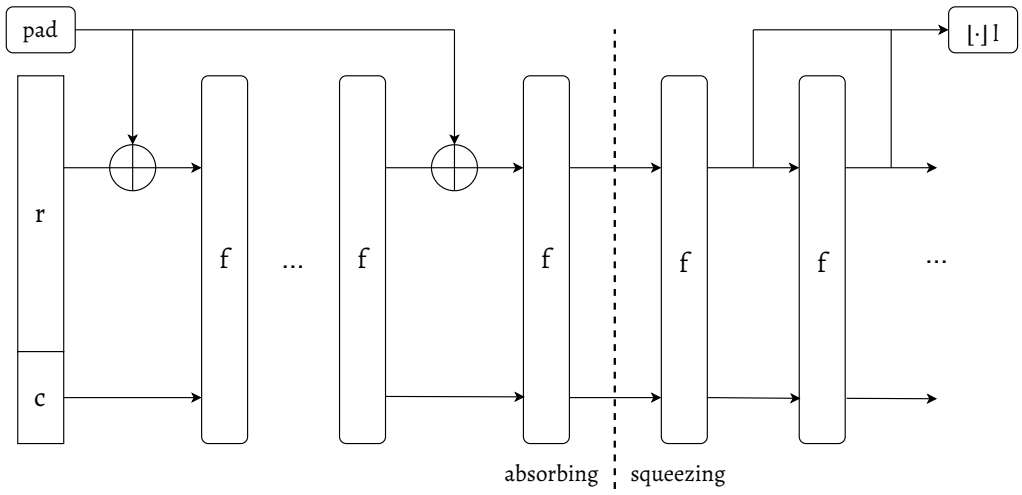


Figure 3.1: The sponge construction

This is described in Algorithm 4. 0^b generates an all zero bit string of length b, while $\lfloor x \rfloor_n$ is the truncation of a string to n bits.

Algorithm 4 SPONGE[f,pad,r]

method SPONGE(M, l)
  $P = M||pad(M, r)$
  Let $P = P_0||P_1||\ldots||P_w$ where $|P_i| = r$
  $s = 0^b$
  for $i = 0$ to w **do**
    $s = s \oplus (P_i||0^{b-r})$
    $s = f(s)$
  end for
  $Z = \lfloor s \rfloor_r$
  while $|Z| < l$ **do**
    $s = f(s)$
    $Z = Z||\lfloor s \rfloor_r$
  end while
  return $\lfloor Z \rfloor_l$
end method

Lines 2 to 4 in Algorithm 4 describe the creation of the padded message blocks as well as the initialization of the sponge constructions state. The pseudocode from lines 5 to 8 is the absorbing stage, while lines 9 to 14 are the squeezing stage. It's assumed that $r < b$ and $M \in \mathbb{Z}_2^*$. In general, it should be hard to find a specific state s from the padded input P. On the other side, it should also be hard to find s from the output Z. In other words, both the absorbing and the squeezing steps are hard to invert.

Using a sponge with domain separation, make it possible to create an authenticated encryption scheme. One of these calls is used to generate the key stream, while the second call is used to generate the tag. This AE scheme would result in two separate calls to the sponge construction. To implement the previously mentioned RO$_{\text{WRAP}}$, the random oracles $\mathcal{RO}_C$ and $\mathcal{RO}_T$ would need to be replaced by calls to the sponge construction.

When looking at classical attacks on hash functions, the focus is usually on collision and (second) preimage attacks. In general, for an n-bit hash function these attacks have a complexity of $2^{n/2}$ and 2^n respectively. However, in the case of sponge construction, the complexity is also dependent on the capacity. The collision attack for the sponge has a complexity of $\min\{2^{c/2}, 2^{n/2}\}$, while the complexity for both the preimage attack and the second preimage attack is $\min\{2^{c/2}, 2^n\}$. These attacks try to leverage paths in the state space of the sponge construction to attack. [4, 5]

3.3 Duplex construction

Much like the sponge construction, the **duplex construction** applies a fixed-length **permutation** f, a **padding rule** *pad* and a **rate** r. In contrast to the sponge construction, there is the possibility to request outputs in between calls to the construction. This process is illustrated in Figure 3.2.

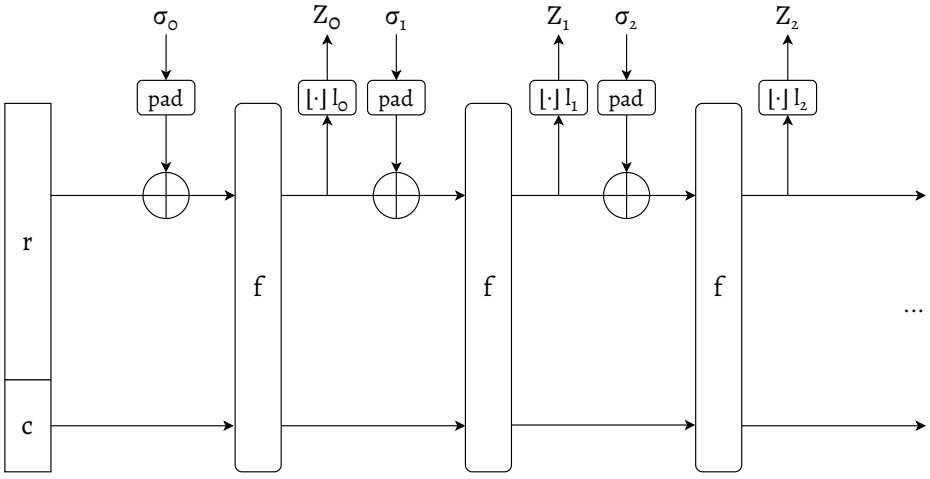


Figure 3.2: The duplex construction

Let D be an instance of the duplex construction, called a *duplex object*. Similar to the

sponge construction the duplex construction has **state size** b and a **capacity** $c = b - r$, which is never directly affected by the input. For every call to D a number of l bits can be requested. At most r bits can be retrieved at once, i.e. $l \leq r$ must hold. It is assumed that an input σ to D is short enough such that padding this input will result in a single r-bit block. The duplex object uses an additional **duplex rate** ρ, that indicates the size of its input blocks. The input is divided into blocks of ρ bits and is then padded afterward. A *mute* call is a call to D that requests no output, i.e. $l = 0$, while a *blank* call is a call to D where σ is the empty string. Algorithmically, this process can be divided into a *initialize* and *duplexing* function for D as described in Algorithm 5.

Algorithm 5 DUPLEX[f,pad,r]

method INITIALIZE( )
  $s = 0^b$
end method

method DUPLEXING(σ, l)
  $P = \sigma || pad(M, r)$
  $s = s \oplus (P || 0^{b-r})$
  $s = f(s)$
  return $\lfloor s \rfloor_l$
end method

Let $\rho_{\max}$ be the maximum ρ for which a block of ρ bits including its padding will result in a single r bit block. $\rho_{\max} > 0$ since otherwise the input would never be processed. As for the sponge construction $r < b$ and $M \in \mathbb{Z}_2^*$ are valid. The state $s \in \mathbb{Z}_2^b$ is maintained across calls. Each σ is chosen such that $\sigma \in \bigcup_{n=0}^{\rho_{\max}} \mathbb{Z}_2^n$.

The duplex construction inherits its security from the sponge construction. This can be shown by reducing the duplex construction to the sponge construction. For the i-th call to D with (σ_i, l_i) the output of the duplex construction is the same as the output of the sponge construction for a specifically formatted input:

$$Z_i = D.duplexing(\sigma_i, l_i) = sponge(\sigma_0 || pad_0 || \sigma_1 || pad_1 || \dots || \sigma_i, l_i)$$

Let pad_i be the padding for input σ_i. This way, multiple calls to the duplex construction can be reduced to a single call to the sponge construction.[5]

3.4 SpongeWrap

The SpongeWrap authenticated encryption scheme makes heavy use of the duplex construction. It is capable of performing authenticated encryption in a single pass. An instance W is initialized using a key K. Afterward, requests for wrapping and unwrapping can be executed. Both the blocks generated as the key stream and the tag depend on all the data that has been input with the previous requests to the underlying duplex object D. The duplex object retains its state during the whole process.

When receiving a wrapping request, the header A and the message M are forwarded to D. The cryptogram is then generated block-wise from the current state of D. A message block M_i is encrypted to $C_i = B_i \oplus Z_i$ where Z_i is the response to the previous call to D. The l-bit tag T is the response to the last message block. In case of $l > \rho$ the tag is extended using blank calls. In the end, C and T are returned. This approach for wrapping and unwrapping is described in Algorithm 6. The algorithm uses a duplex object D to both encrypt the message and generate the tag. The frame bit added to each of the blocks passed to D provides domain separation. It separates the blocks to be used for the key stream generation from the blocks to be used to generate the tag. This has to be considered, when choosing the correct ρ i.e. it is only possible to choose $\rho \leq \rho_{\max} - 1$. If this is not considered padding an input for the duplex will result in two blocks instead of one.

Because SpongeWrap uses the duplex construction its security is the security of the duplex construction. As previously mentioned, the security of the duplex can be reduced to the security of the sponge construction. So if the sponge is secure, then SpongeWrap is also secure. The advantage of an attacker for privacy and authenticity can be described by using the terms from Section 3.1 and some additional terms:

$$\mathrm{Adv}^{\mathrm{priv}}_{\mathrm{SpongeWrap}[f,pad,r,\rho]}(\mathcal{A}) < q2^{-k} + \frac{N(N+1)}{2^{c+1}}$$
$$\mathrm{Adv}^{\mathrm{auth}}_{\mathrm{SpongeWrap}[f,pad,r,\rho]}(\mathcal{A}) < q2^{-k} + 2^{-t} + \frac{N(N+1)}{2^{c+1}}$$

As before the attacker can only make q queries to the algorithm, while $|K| = k$ and $|T| = t$. The new parameter N describes the number of calls to the permutation f.

SpongeWrap comes with it some advantages and limitations. For one the output is produced in a single pass, making it unnecessary to make two separate calls. Each of the block's input into the construction, only a single call to the permutation function is needed. In addition, it also does not need a whole block cipher, but only uses a fixed-length permutation. The whole construction is very flexible in that strings requiring authenticated encryption and those only needing authentication can be supported. Also, intermediate tags for wrap requests can be produced. Through the duplex construction, it inherits the security bounds against attacks from the sponge construction, as well as the property of not being expandable. That being said, so long as the capacity is large enough, the construction will be secure.

However, on an algorithmic level SpongeWrap is not easily parallelized, because the state always depends on the previous inputs. This might make it less desirable compared to other options that can be parallelized in such a manner. In addition to this, the algorithm does not require a nonce. Thus by nature, the output is deterministic, if no additional requirement for a nonce is added. As mentioned before, this problem can be solved by including a nonce in the header. [4, 5]

This thesis assumes a single bit-flip attack model for the attacker. A single bit-flip is introduced by an attacker into one of the parameters used by the accelerator to calculate

Algorithm 6 The SpongeWrap authenticated encryption scheme

method INITIALIZE(K)
  Let $K = K_0||K_1||\ldots||K_u$ with $|K_i| = \rho$ for $i < u$, $|K_u| \leq \rho$ and $|K_u| > 0$ if $u > 0$
  $D.initialize()$
  for $i = 0$ to $u - 1$ **do**
    $D.duplexing(K_i||1, 0)$
  end for
  $D.duplexing(K_u||0, 0)$
end method

method WRAP(A, B, l)
  Let $A = A_0||A_1||\ldots||A_v$ with $|A_i| = \rho$ for $i < v$, $|A_v| \leq \rho$ and $|A_v| > 0$ if $v > 0$
  Let $B = B_0||B_1||\ldots||B_w$ with $|B_i| = \rho$ for $i < w$, $|B_w| \leq \rho$ and $|B_w| > 0$ if $w > 0$
  for $i = 0$ to $v - 1$ **do**
    $D.duplexing(A_i||0, 0)$
  end for
  $Z = D.duplexing(A_v||1, |B_0|)$
  $C = B_0 \oplus Z$
  for $i = 0$ to $w - 1$ **do**
    $Z = D.duplexing(B_i||1, |B_{i+1}|)$
    $C = C||(B_{i+1} \oplus Z)$
  end for
  $Z = D.duplexing(B_w||0, \rho)$
  while $|Z| < l$ **do**
    $Z = Z||D.duplexing(0, \rho)$
  end while
  $Z = \lfloor Z \rfloor_l$
end method

method UNWRAP(A, C, T)
  Let $A = A_0||A_1||\ldots||A_v$ with $|A_i| = \rho$ for $i < v$, $|A_v| \leq \rho$ and $|A_v| > 0$ if $v > 0$
  Let $C = C_0||C_1||\ldots||C_w$ with $|C_i| = \rho$ for $i < w$, $|C_w| \leq \rho$ and $|C_w| > 0$ if $w > 0$
  Let $T = T_0||T_1||\ldots||T_x$ with $|T_i| = \rho$ for $i < x$, $|T_x| \leq \rho$ and $|T_x| > 0$ if $x > 0$
  for $i = 0$ to $v - 1$ **do**
    $D.duplexing(A_i||0, 0)$
  end for
  $Z = D.duplexing(A_v||1, |C_0|)$
  $B_0 = C_0 \oplus Z$
  for $i = 0$ to $w - 1$ **do**
    $Z = D.duplexing(B_i||1, |C_{i+1}|$
    $B_{i+1} = C_{i+1} \oplus Z$
  end for
  $Z = D.duplexing(B_w||0, \rho)$
  while $|Z| < l$ **do**
    $Z = Z||D.duplexing(0, \rho)$
  end while
  if $T = \lfloor Z \rfloor_l$ **then**
    return $B_0||B_1||\ldots||B_w$
  else
    return Error
  end if
end method

the output. These parameters are located in the memory and might be susceptible to both fault injection by an attacker and random bit flips caused by unreliable hardware. Depending on the position of the fault, the result is a decrease in the quality of outputs computed by the NN. In some cases, an attacker is also able to extract the parameters of the network. Even though NNs are considered to be fault tolerant, this can not generally be stated for all of them. Fault tolerance is highly dependent on the architecture of the network and the number of parameters. This is also shown to be the case for several smaller networks in Section 4.3.

Towards mitigating this issue, the SpongeWrap AE scheme is employed. The NNs parameters are encrypted and stored in memory. Upon retrieval, parameters are decrypted using the unwrapping operation. In case a fault was detected, this is indicated by a flag. SpongeWrap will be used in its deterministic variant without the addition of a nonce. A correction of the fault is not required. Regardless of the source of the fault SpongeWrap should be able to detect that something is amiss.

4

SystemC Implementation

The functional model of the accelerator is implemented in SystemC for an existing virtual prototype (VP). SpongeWrap is implemented in several C++ classes, that provides the cryptographic functionality. A SystemC module makes use of these classes to decrypt some values. To provide an impression of the impact of a single-bit flip, an analysis of multiple simple NNs is performed. Some of the results of this analysis are described in this chapter as well. Lastly, the results are put into perspective concerning the trustworthiness achieved by the functional model.

4.1 SystemC

In theory, hardware can be designed by simply combining a number of logic gates. However, even though this might appear to be a feasible option, this approach proves too work-intensive for more complex structures. *Hardware description languages* (HDL) such as VHDL or Verilog provide a level of abstraction that makes it possible to create more complex designs more efficiently. They have been widely adapted for designs on the *behavioral* and *register transfer level* (RTL). This increased the productivity in designing hardware immensely in comparison to previous modeling approaches.

But even this more refined approach to hardware design can not keep up with the increasing demand for even more complex designs. Oftentimes it is not productive to create models at the level of single bits. In addition, modules have long since ceased to be independent. They interact with other components on the same chips, which influences the design process considerably. Therefore they have to be taken into consideration when creating simulations. As a result, this process becomes increasingly slow and inefficient.

SystemC is a C++-based alternative to the usual HDL approach in hardware design. The standard is supported by the Open SystemC Initiative (OSCI), now Accellera System Initiative[7], and was standardized by IEEE first in 1666-2005[14] and then later in 1666-2011[13]. Version 1.0 provides the common HDL features, while Version 2.0 comes with more general and abstract communication components and events. It is composed of a C++ class

[7] https://www.accellera.org/

library and a simulation kernel that can be used to build and simulate models on the system, behavioral or RTL level. Using C++ brings the advantage of vast modeling possibilities and popularity in the software community. Still, there exist a lot of constructs that are not suitable for hardware designs. This makes it necessary to adhere to a certain subset of constructs that can sensibly be synthesized if needed.

The library provides all the features that are usually also provided by HDLs. In addition, it furthermore supplies options for system-level design. This way, modeling on lower and higher levels of abstraction is possible. A number of features make this possible. An overview of all SystemC elements is provided in Figure 4.1. Features build on top of each other and use functionality provided by other features:

- **Modules:** Complex systems are decomposed into multiple smaller components called modules. They provide the description of a SystemC model and encapsulate some parts of the functionality.
- **Signals and Ports:** Low-level method of connecting modules equivalent to those found in HDLs. Ports can support signals of both standard C++ data types and custom data types. They are either input, output, or bidirectional.
- **Methods and Threads:** Both methods and threads allow the modeling of concurrent behavior in the model. They can be triggered by variables in their sensitivity list. While methods model combinational behavior, threads model sequential behavior. Methods do not have their own thread of execution, and they ca not be suspended. On the other side, these two things are both possible for threads. Threads may also be re-activated.
- **Time and Clock:** Both can be specified. Methods and threads can be made sensitive to the clock.
- **Testbenches:** The testbenches can be specified using all the tools available in C++ since they do not need to be synthesizable. This way, they are more flexible than the testbenches that can be created in HDLs.
- **Data Types:** C++ data types are implementation dependent. For that reason, they might not be suited for certain applications. SystemC provides data types where a number of bits can be specified. In addition, both 2-valued and 4-valued logic values and vectors are available. Certain data types affect the simulation time greatly.
- **Events and Sensitivity:** Events can be both specified and notified. They make it possible to model complex interactions.
- **Interfaces and Channels:** The latter allow a separation of the computational functionality and the communication functionality. Interfaces provide method declarations that are then implemented by channels. Methods are visible to the port and thus are usable by the module. SystemC provides some pre-implemented primitive channels like signals, mutex, and fifo. Hierarchical channels make it possible to implement complex protocols.

Once a model has been created using the tools provided by SystemC, it can be compiled and linked with the class library using a C++ compiler. Testbenches are compiled at this point as well. The resulting executable serves as a simulator that can be debugged by means of C++ debuggers such as gdb. SystemC also provides capabilities to generate

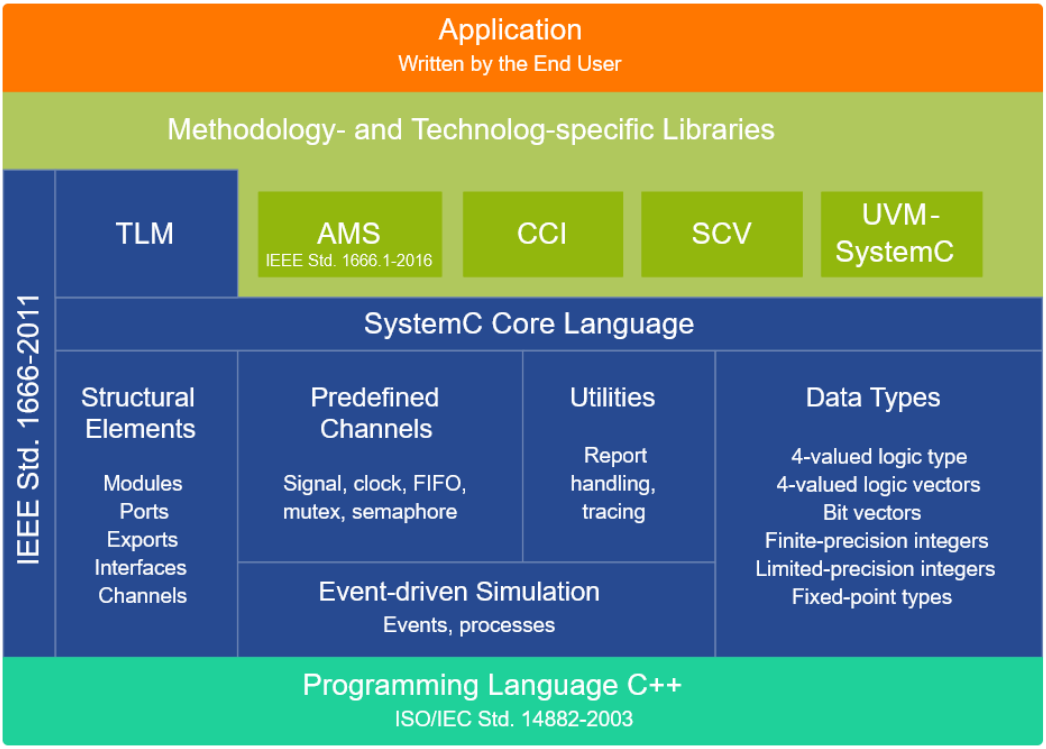


Figure 4.1: Overview SystemC library [8]

trace files during simulation.

Hardware, software, and testbenches are simulated in the same environment using the same specification language. Depending on the level of abstraction, there might not even be a distinction between software and hardware in the model. This division can be undertaken at a later time during the design phase. SystemC provides possibilities to model at the system level with untimed functional models or transaction-level models, at the behavioural level and the RTL level. Transitioning from a higher level of abstraction can be achieved by both automatic and manual means. At the RTL level, a design can finally be synthesized into a netlist. If constructed correctly, testbenches are reusable for all abstraction levels.

SystemC ships some methodology-specific libraries that provide tools for different modeling approaches. One of them is transaction-level modeling (TLM). This approach tries to solve the problem of high simulation time for complex timed systems. Creating and simulating a precisely timed model is very time-consuming. However, such a precise model might not be necessary.

TLM is a high-level modeling approach that uses the abstract communication components provided by SystemC. Transactions are triggered when interface methods are called. A transaction object contains information like the command, start address, the data length,

and a data pointer. Transactions can be delayed to get a more accurate timing model. TLM puts the emphasis on the kind of data that is transferred from and to destinations in the model. The actual implementation of the transfer is of less importance. TLM models are approximately timed using the timing annotations provided by SystemC. This approach makes it easier to swap out elements and thus makes testing and experimenting more efficient.

Components in the model build on top of the simulation kernel and might use other mechanics to interact with it. This kernel is the backbone of the actual simulation. It executes the processes and updates and advances the simulation time, i.e. it performs the actual simulation and provides results as intended by the designer. [12, 18, 20]

4.2 Functional Model

The VP presented in [12] provides different pre-defined virtual platforms in a public repository[9]. The functional NN accelerator is added to one of these virtual prototypes as a new SystemC module. The same is true for SpongeWrap. In the following subsections, the VP and the two SystemC modules are described.

RISC-V based Virtual Prototype

The *basic* VP platform is can be found under *riscv-vp/vp/src/platform/basic*. Classes and modules that are shared between VPs are provided in *riscv-vp/vp/src/platform/common*. In addition, the core functionality for the VPs is given in *riscv-vp/vp/src/core*. The VP delivers both functionalities for the 32-bit address space and the 64-bit address space. The 128-bit address space variant is not available as of writing. All relevant classes for the 32-bit version are available in the *rv32* directory, while the classes for the 64-bit version can be found in the *rv64* directory. Classes that can be used by both versions are supplied in the *common* directory. It should be noted that the 64-bit version is an extension that was added after the publication of the original paper. The *basic* VP uses the 32-bit version.

C/C++ applications can be compiled using the RISC-V GNU Compiler Toolchain[10]. The resulting executable can then be run on the VP by passing it as a command line argument. Debugging with GDB is enabled using a console argument as well. In this case, the VP acts as a server, while the GDB debugger works as the client. The repository supplies several example applications that can be run on the VP.

In addition to the debugging support, other features can be enabled by means of command-line arguments. Options supported by all VPs provided are provided by the *Options* class in the *riscv-vp/vp/src/platform/common* directory. They include the input file itself, as well as the debug capabilities and the interception of system calls. This would, e.g. make it possible to use standard C/C++ libraries. In addition, each of the VPs provides its own

[9] https://github.com/agra-uni-bremen/riscv-vp
[10] https://github.com/riscv-collab/riscv-gnu-toolchain

specific options. For the *basic* VP, these options are supplied by the *BasicOptions*. This class is included in the *main.cpp* file of this platform. Options specific to the *basic* VP include the address spaces for all components, as well as the size of the main memory. Both classes provide the possibility to parse command line arguments by using the *boost* C++ library.

Such as for any program using the SystemC library, the entry point is not the usual *main* function, but the SystemC typical *sc_main* function. It is located alongside the *BasicOptions* class in the *main.cpp* file. Upon entry into the program, the command line arguments are parsed and interpreted as options to be set for the VP using the method provided by the *BasicOptions* class. Afterward, the random generator is seeded using the current time, and the system global quantum is set. Afterward, all of the VPs components are initialized:

- **Instruction Set Simulator** (ISS): The ISS simulates the core of the VP. The *basic* VP uses only a single 32-bit RISC-V RV32IM core. However, the implementation also allows for the use of the reduced RV32E instruction set. Other VPs might also be multicore. The class models and uses things like registers, control and status registers (CSRS) and provides methods to read and manipulate them. In addition, it also manages interrupts and naturally executes the instructions loaded from the memory. Optionally a system call handler can be passed to an *ISS* instance. The *ISS* will then intercept and handle system calls directly. Since the class is specific to the address space size, the *ISS* class and its header can be found in the *rv32* directory.
- **SimpleMemory**: The *SimpleMemory* models a simple memory module. Upon receiving a transaction or a direct memory access (DMA), the desired actions are executed. A *SimpleMemory* instance can also be turned into read-only memory. The **SimpleMemory** class can be found in *riscv-vp/vp/src/platform/common/*.
- **SimpleTerminal**: The *SimpleTerminal* models a simple terminal module. Upon receiving a TLM write command, it writes data at the data pointer of the transaction is pointing at to the command line. The *SimpleTerminal* class can be found in *riscv-vp/vp/src/platform/common/*.
- **UART**: The *UART* class models a Universal Asynchronous Receiver Transmitter (UART) module. Depending on the current mode of the module, either an input character is forwarded to the guest system or the next input is interpreted. The module is notified via an interrupt in case a new character has been received. The *UART* class and all associated classes can be found in *riscv-vp/vp/src/platform/common/*.
- **ELFLoader**: The *ELFLoader* class loads an executable and linkable format (ELF) file into the main memory and extracts relevant information. The ELF file is provided to the VP as a command line argument. Since the *ELFLoader* class is needed for all of the VPs, it can be found in *riscv-vp/vp/src/core/common/*.
- **SimpleBus**: The *SimpleBus* class models a simple bus connecting all the components of the VP. It receives transactions from components and forwards them to the respective target. Targets are identified by their address. Addresses are transformed from global to local ones upon receiving the transaction. All addresses for components are specified in the *BasicOptions*. Since the *SimpleBus* class is used for multiple platforms,

it can be found in *riscv-vp/vp/src/platform/common*.

- **CombinedMemoryInterface**: The *CombinedMemoryInterface* class provides an access point to the memory for the core. It supplies functionality that generates transactions, which are then passed to the bus. Since the *CombinedMemoryInterface* class is specific to the 32-bit address space, it can be found in the *rv32* directory.
- **SyscallHandler**: The *SyscallHandler* class models the system call handler provided to the core. System calls are executed by redirecting them to the host system running the VP. Arguments have to be passed from the VP to the host system, and return values have to be integrated back into the guest system. The handler provides two interfaces: one to allow the ISS to delegate system calls to the host and another one that allows the system handler to access and control the ISS. All the relevant classes and files can be found in the *rv32* directory.
- **FE310_PLIC**: The platform level interrupt controller (PLIC) used in the *basic* VP is based on the PLIC described in FE310-G000 manual[11]. This PLIC is a newer addition, that was added after the publication of [12]. The PLIC complies with the PLIC described in [27]. Global interrupt sources are connected to interrupt targets. The targets are usually hardware threads. Each of the interrupt sources receives a reference to the PLIC. This reference can be used to trigger an interrupt, which is then handled by the *FE310_PLIC* class as external interrupts. Since the *FE310_PLIC* class is used in several of the provided platforms, it can be found in *riscv-vp/vp/src/core/common/*.
- **CLINT**: The core local interrupt controller provides local timer interrupts with a memory-mapped configuration. Relevant to the CLINT is the mtime register and the mtimecmp register. The mtime register is shared between cores, while there is a mtimecmp register available for each of the cores in the VP. In this case, it is a single register because the VP has a single core. Once the value of mtime is greater or equal to the value of mtimecmp, a timer interrupt is generated. The CLINT is considered to be core functionality and thus can be found in *riscv-vp/vp/src/core/common/*.
- **SimpleSensor**: The *SimpleSensor* class was implemented by the authors of [12] to show the principles of modeling peripherals and extending the VP. The sensor provides a 64-byte data frame that is periodically updated. In addition, two 32-bit registers *scaler* and *filter* are provided in the module. The *scaler* value determines the speed at which the sensor data is generated, while the *filter* value determines the post-processing applied to the sensor data. The *SimpleSensor* class can be found in the *riscv-vp/vp/src/platform/basic/* directory since it is specific to the *basic* VP.
- **SimpleSensor2**: The *SimpleSensor2* class provides the same functionality as *SimpleSensor*. However, in contrast to the *SimpleSensor* class, it uses some functionality to reduce the implementation size of the *transport* function. Such as the *SimpleSensor* class, the *SimpleSensor2* class can be found in *riscv-vp/vp/src/platform/basic/*.
- **BasicTimer**: The *BasicTime* class models a timer triggering an interrupt through the PLIC every millisecond. The *BasicTime* class is specific to the *basic* VP and thus can be found in *riscv-vp/vp/src/platform/basic/*.
- **SimpleMRAM**: The *SimpleMRAM* class models the magnetoresistive random access memory (MRAM) of the VP. To represent the non-volatility of the MRAM, the mod-

[11] https://sifive.cdn.prismic.io/sifive/4faf3e34-4a42-4c2f-be9e-c77baa4928c7_fe310-g000-manual-v3p2.pdf

ule saves its contents to a file. The file is opened upon initialization of the objects and closed upon the destruction of the object. Transactions allow read and write access at the desired position in the file. The *SimpleMRAM* class can be found in *riscv-vp/vp/src/platform/basic*.

- **SimpleDMA**: The *SimpleDMA* class models the direct memory access (DMA) component of the VP. This allows components to access the main memory independently of the core. Upon receiving a transaction, the module generates transactions to the main memory. Not all operations are implemented. The *SimpleDMA* class can be found in the *riscv-vp/vp/src/platform/basic/* directory.
- **Flashcontroller**: The *Flashcontroller* class models a flash memory controller. The flash memory is represented by a file from which the module can read and write to. It provides a buffer from which data can be read and written into the file in case the contents of the buffer have been changed. By default, the buffer has a size of 512 bits. It provides a register to set the address of the targeted block and a register containing the size of the block to be loaded. Finally, a transaction can be used to load the desired data into the buffer, which can then be either written or read from. Since the *Flashcontroller* class is specific to the *basic* VP, the class can be found in the directory *riscv-vp/vp/src/platform/basic/*.
- **EthernetDevice**: The *EthernetDevice* class models a device connected via an ethernet port to the VP. It provides some memory-mapped configuration registers that provide information for incoming and outgoing data. The module can use its PLIC reference to trigger an interrupt if data is available at the socket. If data is received, it is written into the main memory. On the other side, if it is to be sent, it gets copied from the position in the main memory and into the send buffer. It provides functionality to initialize a network and to send and receive data frames. Since the *EthernetDevice* is specific to the *basic* platform, it is provided in the *riscv-vp/vp/src/platform/basic/* directory.
- **Display**: The *Display* class models a simple display peripheral. This module allows for simple drawing actions like clearing the display, filling the frame with color or drawing a line. If used with the provided application in the *riscv-vp/env/basic/vp-display/* directory a virtual display in the form of a new window can be used to display all the drawing actions. This is possible through shared memory between the VP and the display. This simple display module is specific to the *basic* platform and thus can be found in *riscv-vp/vp/src/platform/basic/* directory.
- **DebugMemoryInterface**: The debug memory interface allows for debugging capabilities that are not available using the CombinedMemoryInterface. Read operations on the memory print the read data to the console. Write operations, on the other side, indicate if all data has been written to the main memory. Otherwise, an error message is displayed.

The VP has a direct memory interface (DMI) that can be used to optimize instruction fetching (*instr_memory_if*) and other accesses to the main memory (*data_memory_if*). These optimizations are enabled using the *use-instr-dmi* and the *use-data-dmi* command line arguments. If both are to be used, they can be enabled at once using the *use-dmi* argument. In case no DMI is used, the core simply uses the *CombinedMemoryInterface* to access mem-

ory. In that case, every access to the main memory generates a transaction that is routed through the bus. If the instruction DMI is used, the *InstrMemoryProxy* is queried instead of the *CombinedMemoryInterface*. On the other side, if the data DMI is used, an instance of the *MemoryDMI* class is added to the *CombinedMemoryInterface*.

In some cases, the bus might need to be locked by some component. For that reason, the VP provides a shared *BusLock* pointer that can be provided to components. The first such component receiving this shared pointer is the *CombinedMemoryInterface*. It uses the lock for store and load operations. Once the operation is finished, the bus is unlocked again. To make sure that the DMA respects the bus locking, an additional *PeripheralWriteConnector* is provided in the *bus.h* file. Upon receiving a transaction, this module checks if the bus is locked. Only if it is unlocked the transaction will be forwarded to the bus.

The entry point of the application to be run on the VP can be extracted from the *ELFLoader*. Subsequently, the VP tries loading the executable image into the main memory. Afterward, the single core is initialized using a *instr_memory_if*, a *data_memory_if*, *CLINT*, the entry point and the stack pointer. If *use-instr-dmi*, *use-data-dmi* or *use-dmi* options are enabled, the *instr_memory_if* and *data_memory_if* provide direct memory access. If one or both are not enabled, then these interfaces are simply references to the *CombinedMemoryInterface*. Finally, the *SyscallHandler* is also initialized using the memory data, the memory start address and the heap address of the application. The core is registered with the system call handler. Only if the system calls are to be intercepted a reference to the handler also passed to the core.

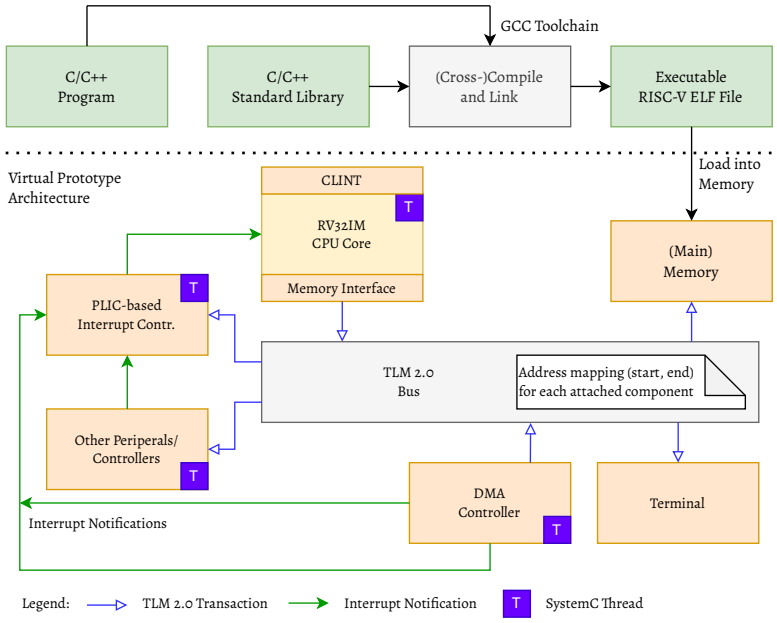


Figure 4.2: Virtual prototype overview. Based original figure in [12].

To enable the bus to forward transactions to the correct component, the *SimpleBus* instance receives a mapping of the start and end addresses for each of the components.

The *CombinedMemoryInterface*, the *DebugMemoryInterface* and the *PeripheralWriteConnector* are then connected to the target sockets of the bus. The DMA is subsequently connected to the *PeripheralWriteConnector*. Afterward, the initiator sockets of the bus are connected to the target sockets of the components that have not been connected yet. Both the PLIC and CLINT receive a reference to the core, and all the components that trigger interrupts receive a reference to the PLIC. Finally, the simulation is started, and thus the application is executed on the VP.

The authors of the VP added both debugging and tracing capabilities. Debugging and tracing can be enabled by the respective command line arguments. Only a simple time model was implemented for the prototype. The architecture of the VP is shown in Figure 4.2.

SpongeWrap Module

The implementation of the SpongeWrap module is decomposed into two parts: the cryptographic implementation of SpongeWrap and the SystemC module itself. All classes pertaining to the cryptographic implementation can be found in the *SpongeWrap* directory of the *basic* platform. The implementation of the Keccak-f function, the padding and the duplex are mostly based on the implementations that can be found in the Keccak GitHub repository[12]. *SpongeWrap* has been implemented by referring to the pseudocode provided in the base paper of SpongeWrap.

The Keccak-f permutation The SpongeWrap AE allows the use of any fixed-size permutation that can be applied to the state of the duplex construction. Keccak-f is the permutation function that is also used by the SHA-3 hash function. [10] provides an extensive description of the Keccak-f function. The permutation function is divided into three files: *Permutation.h*, *KeccakF.h* and *KeccakF.cpp*. These files contain the *Permutation* class on the one side and the *KeccakF* class on the other side. The *KeccakF* class inherits from the *Permutation* class. This allows to implement other permutation functions without having to replace the function in several places.

The *Permutation* class provides two methods to be implemented for any of the permutation functions: the overloading of the () operator and a *getWidth* function. Overloading the () operator makes it easier to use the permutation function in code, while the *getWidth* function returns the size of the permutation. In addition to these methods, a class inheriting from *Permutation* can have many more methods and variables. This is also the case for the *KeccakF* class.

There are several variables provided for a *KeccakF* object. These variables supply information about the Keccak-f instance but are also helpful variables:

- `unsigned int permSize`: The size of the permutation in bits. In the case of the Keccak-f function, the only valid values are 25, 50, 100, 200, 400, 800 or 1600 bits.

[12] https://github.com/KeccakTeam/KeccakTools

- `unsigned int laneSize`: The size of the lanes in the state of the Keccak-f function. Valid sizes are 1, 2, 4, 8, 16, 32 or 64 bits.
- `unsigned int l`: The parameter selecting the size of the permutation. It can be calculated from a given state size b as follows:

$$l = \log_2(b/25)$$

  Since the only state sizes the Keccak-f function supported are 25, 50, 100, 200, 400, 800 and 1600 bits, the only values l can take are 1, 2, 3, 4, 5 and 6.
- `unsigned int numRounds`: The number of rounds the permutation is to be executed for. The needed number of rounds is calculated as specified using the l parameter.
- `vector<unsigned int> rhoOffsets`: The offsets used in the rho step of the Keccak-f function. They are calculated in the constructor of the class.
- `vector<uint64_t> roundConstants`: The round constants used in the iota step of the Keccak-f function. They are calculated in the constructor of the class.
- `uint64_t mask`: A bit mask used during computations to mask smaller lanes in the `uint64_t` lane value. The first *laneSize* bits are 1, all others are 0.

The *KeccakF* class provides both a public empty constructor and a public non-empty constructor. The empty constructor had to be added to ensure that everything compiles. `KeccakF(unsigned int permSize)` creates a Keccak-f permutation function of a certain size. An invalid permutation size triggers an exception. Other than that, all variables previously described are initialized according to the specification. In addition to implementing the methods provided by the *Permutation* class, the *KeccakF* class has multiple public and private methods and functions:

- `unsigned int index(int x, int y)`: To ease the implementation and improve readability, the state of the permutation is represented by a `vector<uint64_t>`. For that reason, the 2-dimensional indices used to index the state have to be translated into their 1-dimensional equivalent.
- `vector<unsigned int> getRhoOffsets()`: Retrieves the *rhoOffsets* of a *KeccakF* object.
- `vector<uint64_t> getRoundConstants()`: Retrieves the *roundConstants* of a *KeccakF* object.
- `void operator()(uint8_t* bytes)`: Applies the permutation to a given state.
- `unsigned int getWidth()`: Retrieves the *permSize* of a *KeccakF* object.
- `void bytesToState (uint8_t* bytes, vector<uint64_t>& out)`: The *bytesToState* method converts some given bytes into a valid state for the Keccak-f function. Depending on the lane size, single bits or whole bytes have to be transferred into the output vector. In the first case, the method uses the *mask* to mask all the bits that do not belong to that lane. This is repeated until all bits have been transferred. By shifting the given bytes in an appropriate manner, all the bits are transferred into the state vector. Full bytes are simply transferred by shifting the `uint64_t` lane value and applying the OR operator.
- `void stateToBytes (vector<uint64_t>& state, uint8_t* out)`: The *stateToBytes* method converts a given state into its byte representation. Like the *bytesToState* method, it differentiates between those lanes containing single bits and those containing full

bytes. Again the lane values are shifted and bits as well bytes are transferred to the output.
- `void initRhoOffsets()`: The *initRhoOffsets* method calculates the rho offsets according to the specification.
- `void initRoundConstants()`: The *initRoundConstants* method calculates the round constants according to the specification. This is done by means of a linear feedback shift register (LFSR) that is implemented in a separate function.
- `void theta (vector<uint64_t>& state)`: The theta step of the Keccak-f permutation function, implemented according to the specification.
- `void rho (vector<uint64_t>& state)`: The rho step of the Keccak-f permutation function, implemented according to the specification.
- `void pi (vector<uint64_t>& state)`: The pi step of the Keccak-f permutation function, implemented according to the specification.
- `void chi (vector<uint64_t>& state)`: The chi step of the Keccak-f permutation function, implemented according to the specification.
- `void iota (vector<uint64_t& state int round)`: The iota step of the Keccak-f permutation function, implemented according to the specification. Because each round uses a different round constant, the *round* parameter indicates which constant to use.
- `bool rc (int t)`: The linear feedback shift register (LFSR) is used to generate the round constants. It returns a single bit of the round constant at a time.
- `void Rnd (vector<uint64_t>& state, int round)`: The *Rnd* function executes a single round of the permutation. It applies one step after another for a given *round*.
- `uint64_t ROL (uint64_t value, unsigned int round)`: The *ROL* method rotates a 64 bit *value* left by a specified amount of *positions*. To ensure that the lane size is respected, the mask has to be applied.

The Multi Rate Padding Like the permutation function, the padding function is the one used by Keccak. The *padding.h* file provides two classes: *DataBlock* and *Padding*. The *DataBlock* class supplies functionality that makes it easier to work on blocks of bits. The *Padding* class is inherited by all implementations of the padding function. The functionality of the padding function is split over four files: *padding.h*, *padding.cpp*, *MultiRatePadding.h* and *MultiRatePadding.cpp*. Again splitting the padding function this way makes it possible to simply replace the padding function in the SpongeWrap implementation without having to change it all too much.

The implementation of the *DataBlock* class can be found in *padding.cpp*. A *DataBlock* keeps track of the number of bits *numBits* it contains, as well as its contents in the form of a `uint8_t` vector *data*. An empty *DataBlock* can be created by means of the *DataBlock*() constructor. The `DataBlock(vector<uint8_t> data, unsigned int numBits)` on the other side creates a *DataBlock* from some given *data* with *numBits* bits. There are several public methods that allow the manipulation of a *DataBlock* object:

- `void appendBit (unsigned int bit)`: Appends a given bit to the block. If necessary, a new byte has to be added to the current *data* vector.
- `vector<uint8_t getData()`: Retrieves the *data* vector currently contained in the *DataBlock*.

- `unsigned int getNumBits()`: Retrieves the number of bits currently contained in the *DataBlock*.
- `void append (uint8_t* input, unsigned int start, unsigned int end)`: Appends a certain part of the *input* to the *DataBlock*. The *start* and *end* values indicate the starting and ending bit in the *input* bytes.
- `DataBlock xorBlocks(DataBlock a, DataBlock b)`: Computes the exclusive-or of two *DataBlock* objects.
- `void truncateBlock (unsigned int l)`: The method truncates a *DataBlock* to a desired length *l*.

In addition to these variables and methods, the *padding.cpp* file also provides the empty *Padding* constructor to ensure that everything compiles. Like the *KeccakF* class, the *Padding* class also provides two methods that are to be implemented by inheriting classes. The *pad* method adds padding up to a multiple of *r* to a given *DataBlock*. In addition, a *getPaddingSize* returns the padded size of a message of a certain length. The *MultiRatePadding* class only implements the two classes given by the *Padding* class:

- `void pad (unsigned int r, DataBlock& data)`: Appends a padding of the form $10 * 1$ to a *DataBlock*. It makes use of the *appendBit* method.
- `unsigned int getPaddedSize (unsigned int inputSize, unsigned int r)`: Computes the output size for a given input size if a message of that length was to be padded with a rate of *r*.

The Duplex Construction The *Duplex* class provides all the functions pertaining to the duplex construction. Towards this, the class has several variables:

- `unique_ptr<uint8_t> state`: The current state of the duplex construction.
- `Permutation* perm`: The permutation function used by the duplex construction.
- `Padding* pad`: The padding rule used by the duplex construction.
- `unsigned int r`: The rate used by the duplex construction.
- `unsigned int max_rho`: The maximum rho that can be used for the duplex construction with a given padding rule.

A *Duplex* object is initialized using a permutation function, a padding function and the desired rate. The constructor ensures that both *r* and *max_rho* assume valid values. The *max_rho* is calculated in the constructor. As described by the specification, the state is initialized to `0`. Other than the constructor, the class provides three methods:

- `void calculateMaxRho()`: Calculates the maximum possible rho. Makes use of the *getPaddedSize* method of the padding function *pad*.
- `void duplexing (uint8_t* input, unsigned int inputLength, uint8_t* output, unsigned int l)`: Implements the duplexing operation as described in the specification. The desired output of length *l* is returned in the *output* parameter.
- `uint8_t* getState()`: Retrieves the current state of the *Duplex* object.

The SpongeWrap Authenticated Encryption Scheme The *SpongeWrap* class provides the main functionality to the final SpongeWrap module. Towards this, the class only contains the duplex rate *rho* and the internal *duplex* object. A *SpongeWrap* instance receives a *Permutation* and *Padding* object that are passed to the *duplex* object alongside the

rate *r*. The *key* is not passed in the *wrap* and *unwrap* methods but in the constructor. In total the class provides four methods:

- `vector<DataBlock> divideIntoBlocks (uint8_t* input, unsigned int inputSize, unsigned int blockSize)`: The *divideIntoBlocks* function divides a given *input* into blocks of *blockSize* bits.
- `DataBlock connectBlocks (vector<DataBlock>& input)`: The *connectBlocks* method retrieves the bits from all given blocks and joins them into a single *DataBlock*.
- `void wrap (uint8_t* header, unsigned int headerLen, uint8_t* message, unsigned int messageLen, uint8_t* ciphertext, uint8_t* tag, unsigned int tagLen)`: Implements the Wrap function described in the specification. The key is added in the constructor instead of adding it in this method as well. Makes use of both *divideIntoBlocks* and *connectBlocks* to generate the output. The *DataBlock* class provides the backbone and enables easy implementation of the required steps.
- `void unwrap (uint8_t* header, unsigned int headerLen, uint8_t* ciphertext, unsigned int ciphertextLen, uint8_t* message, uint8_t* tag, unsigned int tagLen)`: Implements the Unwrap function described in the specification. The key is added in the constructor instead of adding it in this method as well. Makes use of the *divideIntoBlocks* function. The *DataBlock* class provides the backbone and enables easy implementation of the required steps.

The SpongeWrap SystemC Module A *SpongeWrapModule* component requires various parameters. Most of these are required to locate values in the memory components of the VP. The configuration of the SpongeWrap instance is hard-coded into the *SpongeWrapModule* struct. The constructor requires the following parameters in addition to the module name:

- `uint64_t start_ciphertext`: The start address of the ciphertext in the main memory.
- `uint64_t end_ciphertext`: The end address of the ciphertext in the main memory.
- `uint64_t ciphertext_chunk_size`: The size of each ciphertext chunk in bytes, i.e. of each continuously encrypted data block.
- `uint64_t start_tags`: The start address of the tags in memory. Refers either to the main memory or to the read-only memory (ROM).
- `uint64_t end_tags`: The end address of the tags in memory. Refers either to the main memory or to the read-only memory (ROM).
- `uint64_t tag_len`: The length of the tag in bits.
- `uint64_t start_key`: The start address of the key in the main memory.
- `uint64_t key_len`: The length of the key in bits.
- `bool use_ROM`: A boolean value indicating if the tags are to be stored in the main memory or the read-only memory (ROM).

Other than initializing the class variables, the constructor performs other necessary operations. This includes the initialization of the output buffer *decrypted* to the ciphertext chunk size, the permutation function *perm* and the padding function *pad*. Finally, the transport method is registered for transactions received from the bus. The *SpongeWrapModule* component is attached to the bus using a *simple_target_socket*. All incoming transactions are handled by this method.

The module has three memory-mapped registers:

- *TO_DECRYPT:* The input to the *SpongeWrapModule*.
- *DECRYPTED:* The decrypted plaintext. Contains all 0 if the decryption fails.
- *DECRYPTION_SUCCESSFUL:* A flag indicating if the decryption was successful.

All functionality is provided during the processing of an incoming transaction in the *spongeWrap-Transport* method. This method differentiates between multiple cases:

- *Write on TO_DECRYPT*: Deposits the value to be decrypted in the TO_DECRYPT register and triggers the decryption.
- *Read on DECRYPTED:* Copies the decrypted plaintext to the pointer contained in the transaction.
- *Read on DECRYPTION_SUCCESSFUL:* Copies the flag indicating a successful decryption to the pointer contained in the transaction.

All other incoming transactions are considered to be invalid. The actual decryption is performed by a separate method to keep the implementation clean. The *decrypt* method retrieves both the ciphertext and the tag from memory using a TLM transaction. Depending on the initialization, the tag is either fetched from the main memory or the ROM. Finally, a *SpongeWrap* is created based on the cryptographic implementation using the existing variables. Afterward, the retrieved ciphertext is unwrapped using the *unwrap* method. If the decryption is successful, the decrypted plaintext is output into the DECRYPTED register, and the DECRYPTION_SUCCESSFUL register is set to 1. Otherwise, the DECRYPTED register is set to 0, and the DECRYPTION_SUCCESSFUL register is set to 0.

The *SpongeWrapModule* component is added in the *sc_main* function like all the other components. It has both a target and an initiator socket. The target socket is attached to the bus to receive read and write transactions for the module registers. An initiator socket is needed to fetch the ciphertexts and tags from memory. The VP was also extended by an additional ROM module that is attached to the bus using only a target socket.

Functional Accelerator Module

The accelerator is a SystemC module that is attached to the bus just like the SpongeWrap module. To simplify the implementation, the functional accelerator model provides only support for multi-layer perceptrons (MLPs) with 8-bit integer parameters. All layers use either the ReLU or the Softmax activation function. Like the SpongeWrap module, the accelerator is divided into the actual module and the code providing the actual computational functionality. The *Inference8BitInt* class provides the functionality for inference computations, while the *acceleratorWrapper* SystemC module makes use of this functionality. This module will further also be referred to as the *accelerator module*. The module implements threads that perform the inference, as well as a transport method handling all incoming TLM transactions.

Only two of the five methods and functions are actually used outside of the *Inference8BitInt* class. All others are only ever used in the class itself. On construction a new object is provided with a description of the NN it is supposed to provide inference results for:

- `int num_layers`: The number of layers in the network. This includes only the hidden and output layers.
- `int* input_sizes`: The number of inputs each of the layers receives. Includes the number of neurons on the input layer.
- `int* output_sizes`: The number of neurons the layer has to pass its output to.
- `int* activations`: The activation functions used for each of the layers. Only the ReLu and the Softmax activation function have been implemented.
- `int8_t* weights`: All weight associated with the network. The weights for all layers are passed in a continuous array of 8-bit integers. Which parameters belong to which layer can be reconstructed using the other information given to the *Inference8BitInt* object.
- `int8_t* biases`: All biases associated with the network. The biases for all layers are passed in a continuous array of 8-bit integers. Which parameters belong to which layer can be reconstructed using the other information given to the *Inference8BitInt* object.

The start and end indices for the weights and biases are pre-computed in the constructor. This ensures that the values don't have to be recomputed unnecessarily. One of the methods used outside the class is the *getOutputSize*() method, which supplies the user with the output size of the network. This makes it e.g. possible to reserve enough space in an array for the inference results. The heart of the implementation is provided by the `vector<float> layer(vector<float> input, int l)` method. This method is provided with an input and a layer index. It then computes the layer function for the *l*-th layer of the neural network and returns the result. It performs both matrix multiplication and vector addition. Subsequently, the activation function is applied. To keep the implementation more manageable and to make it possible to implement other activation functions easily, the ReLu and the softmax activation function have been separated into their own functions. The two functions are implemented in `void relu(float *x, int len)` and `void softmax(float *x, int len)`.

Lastly the `void inference(float* x, int len, float* out)` provides inference results to the user. It takes an input *x* of length *len* and returns output using the *out* parameter. It applies the layer function once for each of the layers in the network. Some simple test networks are provided as text files in the *accelerator* sub-directory of the *basic* VP platform. One of these test models is loaded in the `sc_main` function of the VP. Line by line all the information needed by the *Inference8BitInt* class is extracted from the file. The weights are encrypted using the *wrap* function provided by the *SpongeWrap* class. Afterward, both the encrypted weights and the unencrypted biases are loaded into the main memory. The tag and the randomly generated key used for the encryption are both placed in separate read-only memory (ROM). Then both the SpongeWrap and the *acceleratorWrapper* module are connected to the bus. Some address information is additionally provided to the accelerator module to enable the retrieval of the parameters from the main memory and the ROM.

The *acceleratorWrapper* model has both an initiator and a target socket. On the target socket, the module receives transactions for its registers or its buffers. Two events synchronize the inference and the setting of the flag indicating a successful inference. Because the module wraps the *Inference8BitInt* class, it necessarily needs all the information needed to construct one of the class objects. All information except the weights and biases are directly passed to the module. There are six memory-mapped registers and buffers:

- *START_INFERENCE:* Once set the register triggers a thread to be unlocked. This thread then executes the inference on some input given in the *in_buffer*.
- *INFERENCE_SUCCESSFUL*: This flag register is set once the inference was successfully completed.
- *IN*: The buffer receiving inputs from the application running on the VP.
- *OUT*: The buffer providing the inference results to the application running on the VP.

An *acceleratorWrapper* component is supplied with all of the information that has not been stored in the memory of the VP. The constructor also registers the transport method. Moreover, two threads are registered. Each is sensitive to one of the events defined in the module:

- **run_inference**: Runs the inference itself. The thread contains a loop that is being blocked by one of the events. Once this event is triggered by a change in the *START_INFERENCE* register, the inference is performed. The thread requests the *SpongeWrap* module to retrieve and decrypt the weights from the main memory. The biases are fetched from the main memory by the accelerator module itself. Then all the relevant information is passed to an *Inference8BitInt* object, which performs the inference.
- **set_inference_done**: Once *run_inference* concludes its computations, it triggers the other event. As a consequence, the while loop in the *set_inference_done* method is unblocked and the *INFERENCE_SUCCESSFUL* flag register is set.

All incoming TLM 2.0 transactions are handled by the *acceleratorTransport* method. Necessary read and write accesses to the registers and buffers are provided. Invalid operations will throw an exception.

4.3 Experimental Fault Injection Attacks

To visualize the impact of a single bit flip on a NN some experiments on simple networks were undertaken. Twelve different models were trained using the MNIST and the Iris dataset[13]. These datasets vary greatly in size. The MNIST provides 60.000 training samples and a separate test dataset of 10.000. These samples are images of handwritten numbers. Consequently, the dataset provides a total of 10 classes. On the other hand, the Iris dataset is very small in comparison. It contains a total of 150 samples from 3 different classes. It turned out that the lack of training data made the Iris networks much harder to train. All models were trained multiple times for an increasing number of epochs. In

[13] https://archive.ics.uci.edu/ml/datasets/iris

the end, a model was selected that had the best validation accuracy.

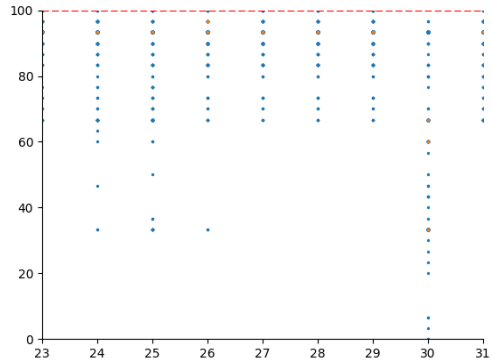
The models are kept small on purpose to make the analysis feasible. All MNIST models are single-layer MLPs. This is because the input layer alone has 784 neurons. Even adding a single hidden layer would cause the network to become too large. An analysis of such a network would take a considerable amount of time. The Iris models have four layers. This is possible because the input layer only has four inputs. The larger number of layers was chosen to make for a more interesting comparison. Another difference between the models lies in the type of parameters used. Some models were quantized using the *qkeras*[14] library. The others were trained using the standard interface provided by keras[15]. These networks use 32-bit float parameters. In some cases, restrictions on the parameters were imposed. Some of the models only use positive parameters, while others use both positive and negative parameters.

During the attacks, a single bit was flipped in a single parameter of the model. Afterward, the models were tested on their test data. For each of the bit flips the accuracy of the faulty model, as well as the layer, the position in the weight matrix or the bias vector and naturally the bit flip position, was recorded. For the 8-bit version, all bits have been flipped once, while for the 32-bit version the nine highest bits were flipped. Doing anything more than this would have caused a signification time increase for the analysis. Still, this type of analysis is very time and work-intensive. Completely analyzing larger models might not be possible at all.

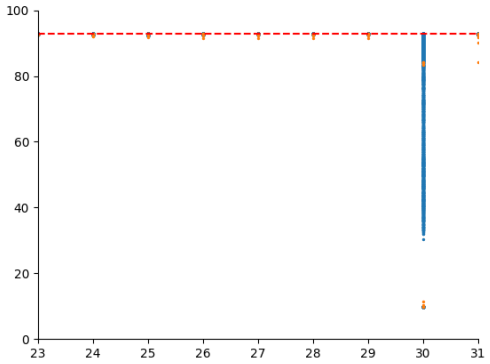
In the following, the results for a few of the models will be presented. For each of them, the x-axis represents the position of the bit flip and the y-axis the accuracy of the faulty model. Weights and biases can be differentiated by their color: blue dots are weights, while orange dots represent the biases. The red dotted line represents the accuracy of the original model. The graphs that are not presented in depth in this section can be found in the appendix of this thesis.
Figure 4.3a, Figure 4.3b, Figure 4.3c and Figure 4.3d are an excerpt of the graphs resulting from the analysis. To make these graphs clearer and more understandable, each layer of a network received its own graph. This is particularly interesting because layers can react differently to a bit flip. In general, it's difficult to predict the reaction of a NN to a fault since they are often very complex.

Each of the graph show very different results. In Figure 4.3b and Figure 4.3c the points are clustering in the same place. On the other side Figure 4.3a and Figure 4.3d are less clustered and more spread out. In some cases, the accuracy can completely be destroyed. However, it is safe to assume that after a certain degradation in accuracy the model becomes unusable. The two MNIST figures allow for another observation: not all bits are vulnerable in the same way. Those bits that cause a large change when flipped are also more likely to cause a drop in accuracy. In this case, the 30th bit of the float is extremely vulnerable. This makes sense because this bit is part of the exponent and thus flipping

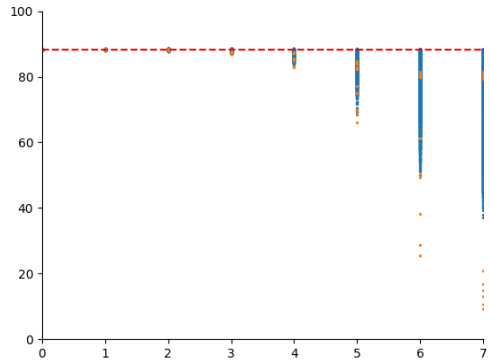
[14] https://github.com/google/qkeras
[15] https://keras.io/



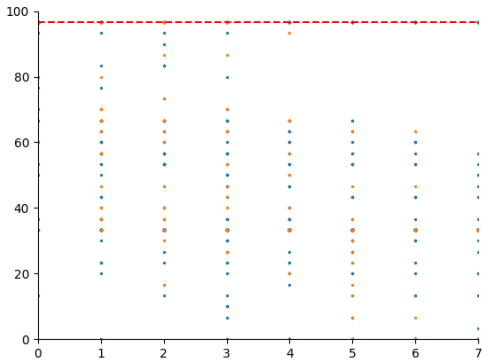
(a) 3. Layer of an Iris NN using 32-bit floats



(b) MNIST NN using 32-bit floats



(c) MNIST NN using 8-bit integers



(d) 1. Layer of an Iris NN using 8-bit integers

this bit can either cause a great increase or decrease in the value of that parameter.

4.4 Trustworthiness Analysis

In the introduction, trustworthiness was defined as encompassing reliability, safety, maintainability, availability, integrity and confidentiality. These goals are impaired by faults, errors and failures (FEFs) as well as security vulnerabilities. This definition allows for consideration of both natural faults i.e. faults caused by aging, thermal issues etc. and „unnatural“ faults i.e. those that are caused by an attacker. Fault injection attacks are often only considered in the context of cryptography. However, they are as much a problem for an AI accelerator. Corrupted inputs or a faulty model might cause the model to become essentially useless. For smaller models, this was shown in the previous section. In general these smaller models are much more susceptible to changes in their parameters. It became very clear, that each model has at least one parameter, that would bring the accuracy close to zero. Larger models behave differently in that sense. First of all, it is not feasible to protect all parameters. It makes much more sense, to use the type of

analysis performed in the previous section to identify vulnerable layers and parameters and to only protect those. In the case of Figure 4.3c and Figure 4.3b the defined peaks allow the identification of vulnerable bits. These peaks might not be as pronounced for a larger model and thus can be used to only protect the relevant parameters.

To prevent model corruption, a SystemC module implements the SpongeWrap authenticated encryption scheme. This module works in conjunction with an accelerator module providing inference results for simple neural networks. It requests the SpongeWrap module to fetch the encrypted values from the main memory, decrypt these values, and return them to the accelerator. The latter can then make use of the decrypted values. Due to the nature of authenticated encryption schemes, both the confidentiality and the integrity of the retrieved values are ensured. Only if the authenticated encryption module communicates that the decryption was successful will the accelerator use the decrypted values. The security of the SpongeWrap AE scheme is derived from the sponge construction. Thus it is very unlikely that an attacker is able to forge valid values to poison the accelerator's outputs. This naturally requires that the instance used during these operations is initialized with values of appropriate size that can't be brute-forced by an attacker.

The accelerator is programmed such that it only performs the inference if the needed values have been decrypted successfully. If this is not the case, the inference is aborted. Consequently, only valid parameters can be loaded from memory. So long as the accelerator is otherwise unbothered, the addition of the SpongeWrap module will ensure that it works reliably. If the accelerator were to be used in a safety-critical application one could also argue that it also ensures safety. This is because a reliable accelerator would prevent undesirable effects to occur. The availability of the accelerator could even be impaired by the addition of the SpongeWrap module. Where previously faulty values would have been loaded and used, now this is not the case. If a faulty weight is detected, the inference is suspended and thus will not appear available, because no inference results were output.

The SpongeWrap instance used by the module is deterministic. This allows an attacker to know which values encrypted under the same key have the same value. This could pose an issue, e.g. for sparse networks, where a lot of the parameters are zero. An attacker would know in this case, which of the values is zero. In addition, the implementation is quite slow, making it poorly suited for large-scale applications, where time is key. Among other things, this is because SpongeWrap is not easily parallelized. Its output depends on all previous inputs. Because of that, it might not be the method of choice for the protection of an AI accelerator. Still, it provides a step in the direction of trustworthy AI accelerators. It provides protection for the integrity,

5

Conclusion

5.1 Summary

In this thesis, a functional neural network accelerator protecting its weights using the SpongeWrap authenticated encryption scheme is implemented in SystemC. The resulting accelerator module and the SpongeWrap module are attached to the bus of a virtual prototype. The weights of the accelerator have previously been encrypted using the SpongeWrap AE scheme. Both the weights and their associated tags are then stored in the memory of the virtual prototype. Upon receiving an inference request, the accelerator requests the SpongeWrap module to decrypt the weights such that they can be used during inference.

To show the impact of a single-bit flip on a neural network, several simple neural networks were created and analyzed. The results show that each of these neural networks has some parameter that would cause the accuracy to decrease dramatically once a bit is flipped. The SpongeWrap module implemented alongside the accelerator circumvents these malicious attacks by only decrypting the values on demand from the accelerator. Bit flips are detected by the module and the inference is aborted in that case. This way, not only the confidentiality and integrity of the weights are secured, but also the reliability of the accelerator.

5.2 Future Works

The current implementation simply encrypts all the weights a neural network has and loads them into memory. Decrypting each weight for a large network is infeasible. By analyzing the influence of bit flips in certain parameters on the NNs accuracy, vulnerable weights can be discovered. Only these weights can then be encrypted to reduce the number of weights that have to be decrypted for each inference. Larger models are likely to make use of activation functions, layers or techniques that are not yet implemented for the accelerator. It was noted before that SpongeWrap is quite slow. The modularity of the virtual prototypes makes it possible to replace the authenticated encryption scheme

with a different one. This makes it possible to replace SpongeWrap with an authenticated encryption scheme of choice.

Bibliography

[1] Aggarwal, C. C. In: *Artificial Intelligence*. Springer Cham, 2021, pp. viii, 167, 169, 173, 201, 203–205, 211–213, 216, 219, 221, 223–226, 228, 232–233, 235–237, 240–243, 280, 315.

[2] Bauer, B., Ayache, M., Mulhem, S., Nitzan, M., Athavale, J., Buchty, R., and Berekovic, M. On the Dependability Lifecycle of Electrical/Electronic Product Development: The Dual-Cone V-Model. In: *Computer* 55(09):99–106, 2022.

[3] Bellare, M. and Namprempre, C. *Authenticated Encryption: Relations among notions and analysis of the generic composition paradigm*. 2000.

[4] Bertoni, G., Daemen, J., and Peeters, M. Cryptographic sponge functions. In: 2011.

[5] Bertoni, G., Daemen, J., Peeters, M., and Assche, G. V. *Duplexing the sponge: single-pass authenticated encryption and other applications*. Springer Berlin Heidelberg, 2011.

[6] Bertoni, G., Daemen, J., Peeters, M., and Assche, G. V. Sponge Functions. In: 2007.

[7] Breier, J., Jap, D., Hou, X., Bhasin, S., and Liu, Y. *SNIFF: Reverse Engineering of Neural Networks with Fault Attacks*. 2021.

[8] Duchi, J., Hazan, E., and Singer, Y. Adaptive Subgradient Methods for Online Learning and Stochastic Optimization. In: *Journal of Machine Learning Research* 12:2121–2159, 2011.

[9] Dworkin, M. *Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC*. 2007.

[10] Dworkin, M. *SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions*. 2015.

[11] Goodfellow, I., Bengio, Y., and Courville, A. Deep Learning. In: `http://www.deeplearningbook.org`. MIT Press, 2016, pp. 2, 3, 9, 13–14, 101–102, 118–119, 164–165, 186, 226–227, 230, 232, 273, 281–282, 285, 326, 422, 424, 426, 442–443, 255–256, 275–276, 291.

[12] Herdt, V., Große, D., Le, H. M., and Drechsler, R. Extensible and Configurable RISC-V Based Virtual Prototype. In: *2018 Forum on Specification & Design Languages (FDL)*. 2018.

[13] IEEE Standard for Standard SystemC Language Reference Manual. In: *IEEE Std 1666-2011 (Revision of IEEE Std 1666-2005)*, 2012.

[14] IEEE Standard SystemC(R) Language Reference Manual. In: *IEEE Std 1666-2005*, 2006.

[15] Kingma, D. P. and Ba, J. Adam: A Method for Stochastic Optimization. In: *CoRR*, 2015.

[16] Krawczyk, H. The Order of Encryption and Authentication for Protecting Communications (or: How Secure Is SSL?) In: *Advances in Cryptology — CRYPTO 2001*. 2001.

[17] Liu, Y., Wei, L., Luo, B., and Xu, Q. Fault Injection Attack on Deep Neural Network. In: *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 2017, pp. 131–138.

[18] Marwedel, P. In: *Embedded System Design*. 4th ed. Springer Cham, 2021, pp. 93–94, 97.

[19] McCulloch, W. S. and Pitts, W. A logical calculus of the ideas immanent in nervous activity. In: 1943.

[20] Panda, P. R. SystemC - a modeling platform supporting multiple design abstractions. In: 2001.

[21] Peccerillo, B., Mannino, M., Mondelli, A., and Bartolini, S. A survey on hardware accelerators: Taxonomy, trends, challenges, and perspectives. In: *Journal of Systems Architecture* 129, 2022.

[22] Peccerillo, B., Mannino, M., Mondelli, A., and Bartolini, S. A survey on hardware accelerators: Taxonomy, trends, challenges, and perspectives. In: *Journal of Systems Architecture* 129, 2022.

[23] Russell, S. and Norvig, P. In: *Artificial Intelligence: A Modern Approach*. 3rd ed. Pearson, 2010, pp. 2, 12, 109–110, 235, 694–696, 719, 727–729.

[24] Shokri, R., Stronati, M., Song, C., and Shmatikov, V. *Membership Inference Attacks against Machine Learning Models*. 2016.

[25] Sze, V., Chen, Y. hsin, Yang, T.-J., and Emer, J. S. Efficient Processing of Deep Neural Networks: A Tutorial and Survey. In: *Proceedings of the IEEE* 105:2295–2329, 2017.

[26] Tajik, S. and Ganji, F. *Artificial Neural Networks and Fault Injection Attacks*. 2020.

[27] Waterman, A., Lee, Y., Avizienis, R., Patterson, D. A., and Asanović, K. *The RISC-V Instruction Set Manual Volume II: Privileged Architecture Version 1.9.1*. 2016.

[28] Xu, Q., Arafin, M. T., and Qu, G. Security of Neural Networks from Hardware Perspective: A Survey and Beyond. In: 2021.

A

Appendix

A.1 SpongeWrap Module

```
#include <systemc>
#include <tlm_utils/simple_target_socket.h>
#include <tlm_utils/simple_initiator_socket.h>
#include "SpongeWrap/KeccakF.h"
#include "SpongeWrap/MultiRatePadding.h"
#include "SpongeWrap/SpongeWrap.h"

using namespace std;

struct SpongeWrapModule: public sc_core::sc_module {

    // The SpongeWrap configuration
    unsigned int permSize = 1600;
    unsigned int rate = 1088;
    unsigned int rho = 1080;
    KeccakF *perm;
    MultiRatePadding *pad;

    // Incoming transactions meant for the SpongeWrap module
    // I.e. accesses to the registers
    simple_target_socket<SpongeWrapModule> tsock;

    // Connection to the target memory
    simple_initiator_socket<SpongeWrapModule> isock;

    // The start of the ciphertext data in the memory
    uint64_t START_CIPHERTEXT;

    // The end of the ciphertext data in the memory
    uint64_t END_CIPHERTEXT;

    // The number of bytes per encrypted line
    uint64_t CIPHERTEXT_CHUNK_SIZE;

    // The start of the tag data in the memory
    uint64_t START_TAGS;

    // The end of the tag data in the memory
    uint64_t END_TAGS;
```

```
// The length of the tags
uint64_t TAG_LEN;

// The start of the key in the memory
uint64_t START_KEY;

// The length of the key
uint64_t KEY_LEN;

// Register containing the line index of the value to be decrypted
uint32_t to_decrypt = 0;

// Decrypted data generated by the module
// Contains all 0 in case decryption fails
uint8_t *decrypted;

// Register containing a flag indicating if the decryption was successful
// 0 if decryption was usuccessful
// 1 if decryption was successful
uint8_t decryption_successful = 0;

// Indicates if a ROM is to be used for the tags
bool use_ROM;

// Memory mapped registers
enum {
    TO_DECRYPT_ADDR = 0x00,
    DECRYPTED_ADDR = 0x08,
    DECRYPTION_SUCCESSFUL_ADDR = 0x04,
};

// We need to pass an interrupt number
SC_HAS_PROCESS(SpongeWrapModule);

SpongeWrapModule(sc_core::sc_module_name, uint64_t start_ciphertext, uint64_t
    end_ciphertext, uint64_t ciphertext_chunk_size,
uint64_t start_tags, uint64_t end_tags, uint64_t tag_len, uint64_t start_key,
    uint64_t key_len, bool use_ROM = false) :
START_CIPHERTEXT(start_ciphertext), END_CIPHERTEXT(end_ciphertext),
    CIPHERTEXT_CHUNK_SIZE(ciphertext_chunk_size),
START_TAGS(start_tags), END_TAGS(end_tags), TAG_LEN(tag_len), START_KEY(start_key),
    KEY_LEN(key_len), use_ROM(use_ROM)
 {
  // Init the output buffer
    decrypted = new uint8_t[CIPHERTEXT_CHUNK_SIZE];

    // Init the permutation and the padding fucntion
    perm = new KeccakF(permSize);
    pad = new MultiRatePadding();

    // Register the transport method
    tsock.register_b_transport(this, &SpongeWrapModule::spongeWrapTransport);
}

/**
 * @brief b_transport method for transactions meant for the SpongeWrap module.
 *
 * @param trans The received transaction.
 * @param delay The delay passed with the transaction.
 */
void spongeWrapTransport (tlm::tlm_generic_payload &trans, sc_core::sc_time &delay) {

    // The command received with the transaction
```

```
    auto cmd = trans.get_command();

    // The register that was targeted by the transaction
    auto addr = trans.get_address();

    // Pointer to the data
    auto ptr = trans.get_data_ptr();

    // The length of the access
    auto len = trans.get_data_length();

    // Write commands directed to the TO_DECRYPT register
    if ((addr == TO_DECRYPT_ADDR) && (cmd == tlm::TLM_WRITE_COMMAND)) {

        // The complete register has to be written
        assert(len == 4);

        // Set the index of the value to be decrypted
        to_decrypt = *((uint32_t*)ptr);

        // Reset the flag
        decryption_successful = 0;

        // Start the decryption
        decrypt();

    // Read commands directed to the DECRYPTED data frame
    // I.e. retrieving the decrypted data
    // Is all zero in case the decryption failed
    } else if ((addr >= DECRYPTED_ADDR) && (cmd == tlm::TLM_READ_COMMAND)) {

        // Copy the data
        memcpy(ptr, &decrypted[addr-8], len);

    // Read commands directed to the DECRYPTION_SUCCESSFUL register
    } else if ((addr == DECRYPTION_SUCCESSFUL_ADDR) && (cmd ==
        tlm::TLM_READ_COMMAND)) {
        *((uint8_t*)ptr) = decryption_successful;
    }
    else {
        cout << "Invalid command!" << endl;
    }
}

void decrypt() {
    // Find the starting address of the ciphertext data to be retrieved
    uint64_t chunk_ciphertext_idx = START_CIPHERTEXT + (to_decrypt *
        CIPHERTEXT_CHUNK_SIZE);

    // Check that we don't read out of bounds
    if (chunk_ciphertext_idx + CIPHERTEXT_CHUNK_SIZE-1 > END_CIPHERTEXT) {
        cout << "Index of ciphertext is out of bounds!" << endl;
        return;
    }

    // Set up the transaction to retrieve the ciphertext
    tlm::tlm_generic_payload* trans1 = new tlm::tlm_generic_payload;
    sc_core::sc_time delay1 (0, SC_NS);
    uint8_t ciphertext[CIPHERTEXT_CHUNK_SIZE];
    trans1->set_address(chunk_ciphertext_idx);
    trans1->set_data_length(CIPHERTEXT_CHUNK_SIZE);
    trans1->set_command(tlm::TLM_READ_COMMAND);
    trans1->set_data_ptr(reinterpret_cast<unsigned char*>(&ciphertext));
```

```
// Retrieve the ciphertext
isock->b_transport(*trans1, delay1);

// Insert a bit flip into the ciphertext
//int indexFault = rand() % (CIPHERTEXT_CHUNK_SIZE * 8);
//int byteIndex = (indexFault - (indexFault % 8)) / 8;
//uint8_t mask = (uint8_t)1 << (7-(indexFault% 8));
//uint8_t byteFaulty = ciphertext[byteIndex] ^ mask;
//ciphertext[byteIndex] = byteFaulty;

// Find the starting address of the tag data to be retrieved
uint64_t chunk_tag_idx = START_TAGS + (to_decrypt * (TAG_LEN/8));

// Set up the transaction to retrieve the tag
tlm::tlm_generic_payload* trans2 = new tlm::tlm_generic_payload;
sc_core::sc_time delay2 (0, SC_NS);
uint8_t tag[TAG_LEN/8];
trans2->set_address(chunk_tag_idx);
trans2->set_data_length(TAG_LEN/8);
trans2->set_command(tlm::TLM_READ_COMMAND);
trans2->set_data_ptr(reinterpret_cast<unsigned char*>(&tag));

// Retrieve the tag either from the memory of the ROM
if (use_ROM) {
    isock->b_transport(*trans2, delay2);
} else {
    isock->b_transport(*trans2, delay2);
}

// Set up the transaction to retrieve the key
tlm::tlm_generic_payload* trans3 = new tlm::tlm_generic_payload;
sc_core::sc_time delay3 (0, SC_NS);
uint8_t key[KEY_LEN/8];
trans3->set_address(START_KEY);
trans3->set_data_length(KEY_LEN/8);
trans3->set_command(tlm::TLM_READ_COMMAND);
trans3->set_data_ptr(reinterpret_cast<unsigned char*>(&key));

// Retrieve the key either from the memory or the ROM
if (use_ROM) {
    isock->b_transport(*trans3, delay3);
} else {
    isock->b_transport(*trans3, delay3);
}

// Setup all the necessary objects for the decryption
SpongeWrap decObject(perm, pad, rate, rho, key, KEY_LEN);

// Decrypt the ciphertext
uint8_t message[CIPHERTEXT_CHUNK_SIZE];
int result = decObject.unwrap((uint8_t *) "", 0, ciphertext,
    CIPHERTEXT_CHUNK_SIZE*8, message, tag, TAG_LEN);

// Transfer the decrypted data to the output
if (result != -1) {
    for(uint64_t i = 0; i < CIPHERTEXT_CHUNK_SIZE; i++) {
        decrypted[i] = message[i];
    }
    decryption_successful = 1;
} else {
    for(uint64_t i = 0; i < CIPHERTEXT_CHUNK_SIZE; i++) {
        decrypted[i] = 0;
```

```
            }
            decryption_successful = 0;
        }
    }

};
```

A.2 Accelerator Module

```
#ifndef ACCELERATORWRAPPER_H_
#define ACCELERATORWRAPPER_H_

// Adding debug logging capabilities to see transaction details
#include<gp2str.h>
#ifndef DEBUG_LOG
#define DEBUG_LOG 0
#endif

#include <systemc>
#include <tlm_utils/simple_target_socket.h>
#include <tlm_utils/simple_initiator_socket.h>

#include <fstream>
#include <iostream>
#include <sstream>

#include <memory>
#include <vector>

#include<cstdint>
#include <unistd.h>

#include<accelerator/Inference8BitInt.h>

using namespace std;

struct acceleratorWrapper: public sc_core::sc_module {

    // Incoming transactions for the accelerator from the bus
    simple_target_socket<acceleratorWrapper> tsock;

    // Outgoing transactions to the main memory and rom (if used)
    simple_initiator_socket<acceleratorWrapper> isock;

    //Events to start the inference
    sc_event e1,e2;

    //////**********MLP parameters ******************////////

    // The number of layers in the MLP
    int num_layers;

    // The input size of each layer
    int *input_sizes;

    // The output sizes of each layer
    int *output_sizes;
```

```
  // The activation function of each layer
  int *activations;

  // The total number of weights
  int weights_total = 0;

  // The total number of biases
  int biases_total = 0;

  //////**********Memory-mapped registers and buffers ******************////////
  unsigned int in_size = 1024;
  unsigned int out_size = 1024;
  float *in_buffer = new float[in_size];
  float *out_buffer = new float[out_size];

  enum {
      START_INFERENCE_ADDR = 0x00,
      INFERENCE_SUCCESSFUL_ADDR = 0x04,
  IN_START=0x08,
  OUT_START=0x2000,
  };

  // Register to start inference
  uint8_t start_inference = 0;

  // Register to output the status of inference 1 if finished
  uint8_t inference_done = 0;

  // The address of the TO_DECRYPT register in the SpongeWrap module
  uint64_t TO_DECRYPT_ADDR;

  // The start address of the biases in the main memory
  uint64_t START_BIAS_MEM;

  // The address of the successful flag of the SpongeWrap module
  uint64_t DECRYPTION_SUCCESSFUL;

  // The address of the decryption buffer in the SpongeWrap module
  uint64_t DECRYPTED_ADDR;

  // Constructor
  SC_HAS_PROCESS(acceleratorWrapper);

  acceleratorWrapper(sc_core::sc_module_name, int num_layers,
  int *input_sizes, int *output_sizes, int *activations):
  num_layers(num_layers),input_sizes(input_sizes),
output_sizes(output_sizes), activations(activations),
in_size(input_sizes[0]), out_size(output_sizes[num_layers-1]) {

    // Register the transport method for the socket
    tsock.register_b_transport(this, &acceleratorWrapper::acceleratorTransport);

      // Run the threads triggered by events
      SC_THREAD(run_inference);
      sensitive << e1;
      SC_THREAD(set_inference_done);
      sensitive << e2;

      // Calculate the total number of weights and biases
      for (int i = 0; i < num_layers; i++) {
        weights_total += input_sizes[i] * output_sizes[i];
        biases_total += output_sizes[i];
      }
```

```
}

// Destructor of the acceleratorWrapper class
~acceleratorWrapper(){
  delete[] in_buffer;
  delete[] out_buffer;
}


void acceleratorTransport (tlm::tlm_generic_payload &trans, sc_core::sc_time &delay){

  // Retrieve the command from the transaction
  tlm::tlm_command cmd = trans.get_command();

  // Get the address from the transaction
  uint32_t addr = trans.get_address();

  // Get the data pointer from the transaction
  auto *ptr = trans.get_data_ptr();

  // Get the length from the transaction
  auto len= trans.get_data_length();

  // Logging messages to see transaction details
  #if DEBUG_LOG == 1
    std::string module_name1 = std::string("Transaction entering ") +
        sc_object::name() + std::string(" with generic payload details ") ;
    SC_REPORT_INFO(module_name1.c_str(), gp2str(trans).c_str());
  #endif

  // Write access to the input buffer
  if ((cmd == tlm::TLM_WRITE_COMMAND) && (addr >= IN_START) && (addr < in_size*4 +
      IN_START)) {

    assert(len == 4);
    int idx = (addr - IN_START)/4;
    in_buffer[idx] = *((float*)ptr);

  // Write access to the start flag
  } else if ((addr == START_INFERENCE_ADDR) && (cmd == tlm::TLM_WRITE_COMMAND)) {

    assert(len == 1);

    // Set the flag
    start_inference = *((uint8_t*)ptr);

    // Ensure that it's set to 1
    assert(start_inference == 1);

    // Trigger the inference process
    if(start_inference == 1){
      e1.notify();
    }


  // Read access to the successful flag
  } else if ((addr == INFERENCE_SUCCESSFUL_ADDR) && (cmd == tlm::TLM_READ_COMMAND)) {

    assert(len == 1);

    // Reads the status register
    *((uint8_t*)ptr) = inference_done;
```

```
    // Reset the flag
    inference_done = 0;

  } else if ((cmd == tlm::TLM_READ_COMMAND) && ( addr >= OUT_START) && (addr <
      (out_size * 4 + OUT_START)) ){

    int idx = (addr - OUT_START)/4;
    memcpy(*((float*)ptr), &out_buffer[idx], len);

    for (int i = 0; i < 10; i++) {
      cout << out_buffer[i] << endl;
    }

  // All unknown commands
  } else {
    sc_assert(false && "unsupported tlm command");
  }
  delay += sc_core::sc_time(10, sc_core::SC_NS);
}

// No argument must be passed here as used as a thread
void run_inference(){

  // Needs to run in a thread and wait for an event
  // This event will start inference
  while(true){

    // Running in a parallel thread
    // Gets triggered every time an event is received
    wait(e1);

    // Retrieve the weights using the SpongeWrap module
    int8_t weights[weights_total];
    sc_core::sc_time delay (10, SC_NS);
    for (unsigned int i = 0; i < weights_total; i++) {

      // Provide the index to be decrypted
      tlm::tlm_generic_payload* trans_start_decryption = new tlm::tlm_generic_payload;
        trans_start_decryption->set_address(TO_DECRYPT_ADDR);
        trans_start_decryption->set_data_length(4);
        trans_start_decryption->set_command(tlm::TLM_WRITE_COMMAND);
        trans_start_decryption->set_data_ptr(reinterpret_cast<unsigned char*>(&i));

        isock->b_transport(*trans_start_decryption, delay);

        // Retrieve the flag indicating if the decryption was successful
        uint8_t successful_flag = 0;
        tlm::tlm_generic_payload* trans_decryption_successful = new
            tlm::tlm_generic_payload;
        trans_decryption_successful->set_address(DECRYPTION_SUCCESSFUL);
        trans_decryption_successful->set_data_length(1);
        trans_decryption_successful->set_command(tlm::TLM_READ_COMMAND);
        trans_decryption_successful->set_data_ptr(reinterpret_cast<unsigned
            char*>(&successful_flag));

        isock->b_transport(*trans_decryption_successful, delay);

        // Retrieve the decrypted weight
        assert(successful_flag == 1 && "decryption failed! Interference stopped.");
        int8_t decrypted_weight;
        tlm::tlm_generic_payload* trans_decrypted= new tlm::tlm_generic_payload;
        trans_decrypted->set_address(DECRYPTED_ADDR);
        trans_decrypted->set_data_length(1);
```

```
            trans_decrypted->set_command(tlm::TLM_READ_COMMAND);
            trans_decrypted->set_data_ptr(reinterpret_cast<unsigned
                char*>(&decrypted_weight));
            isock->b_transport(*trans_decrypted, delay);
            weights[i] = decrypted_weight;
        }

        // Retrieve the biases from the main memory
        int8_t biases[biases_total];
        for (unsigned int i = 0; i < biases_total; i++) {
        tlm::tlm_generic_payload* trans_biases = new tlm::tlm_generic_payload;
        int8_t bias;
        trans_biases->set_address(START_BIAS_MEM + i);
        trans_biases->set_data_length(1);
        trans_biases->set_command(tlm::TLM_READ_COMMAND);
        trans_biases->set_data_ptr(reinterpret_cast<unsigned char*>(&bias));
        isock->b_transport(*trans_biases, delay);
        biases[i] = bias;
        }

          // The object performing the inference
          Inference8BitInt inference(num_layers, input_sizes, output_sizes, activations,
              weights, biases);

          // The result of the inference for the test input
          float result[inference.getOutputSize()];

          // Run the inference
          inference.inference(in_buffer, 784, result);

        // Notify the other thread that the inference is finished
        e2.notify();
      }
    }

    void set_inference_done(){

      while(true){

        // Wait for the event indicating inference completion
        wait(e2);
        inference_done = 1;
      }
    }
};

#endif /* ACCELERATORWRAPPER_H_ */
```

A.3 Extended Virtual Prototype

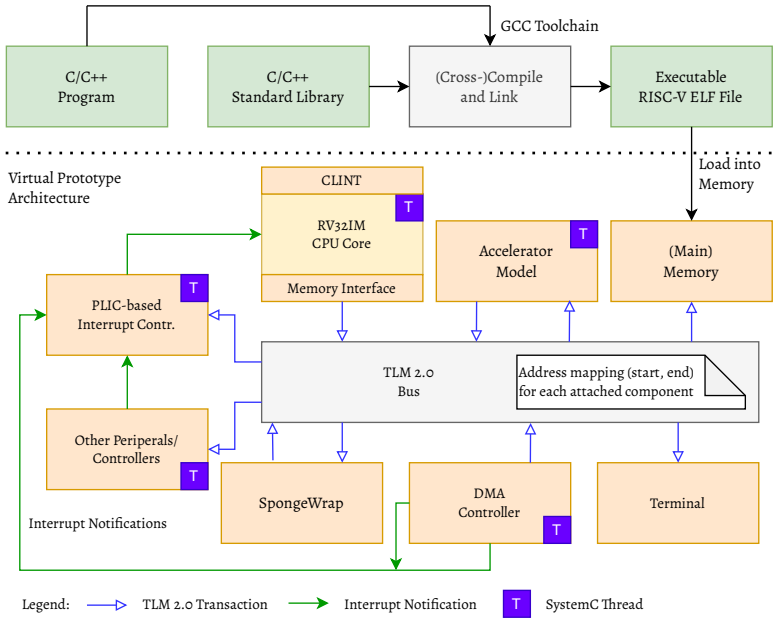


Figure A.1: The virtual prototype including the SpongeWrap and the accelerator module