Secure Neural Network Inference as a Service with Resource-Constrained Clients

Rik de Vries
Zoltán Ádám Mann
University of Amsterdam
Amsterdam, The Netherlands

ABSTRACT

Applying services computing to neural networks, a service provider may provide inference with a pre-trained neural network as a service. Clients use the service to get the neural network's output on their input. To protect sensitive data, secure neural network inference (SNNI) entails that only the client learns the output; the input remains the client's secret and the neural network's parameters remain the service provider's secret. Several SNNI approaches were proposed and evaluated in environments where both service providers and clients used powerful computers.

In many real settings, for instance in edge computing, client devices are resource-constrained. This paper is the first to investigate the impact of client-side resource constraints on SNNI. We perform experiments with two state-of-the-art SNNI approaches and three neural networks. We vary the compute and memory capacity of the client device and measure the impact on inference time. Our findings show that client-side resource constraints significantly impact the performance and even the applicability of SNNI approaches. The results indicate the limits of current SNNI approaches for resource-constrained clients. Based on the results, we identify research directions to improve SNNI for resource-constrained clients.

CCS CONCEPTS

• **Security and privacy** → *Distributed systems security*; • **Computing methodologies** → *Neural networks*; • **Computer systems organization** → *Client-server architectures.*

KEYWORDS

Machine learning as a service, neural network, privacy-preserving machine learning, inference, edge computing, edge intelligence, multi-party computation, homomorphic encryption

ACM Reference Format:
Rik de Vries and Zoltán Ádám Mann. 2023. Secure Neural Network Inference as a Service with Resource-Constrained Clients. In *Proceedings of 16th IEEE/ACM International Conference on Utility and Cloud Computing (UCC 2023).* ACM, New York, NY, USA, 10 pages. https://doi.org/XXXXXXX.XXXXXXX

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
UCC 2023, December 04–07, 2023, Taormina (Messina), Italy
© 2023 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Machine learning, and neural networks in particular, have gained much attention over the last few years. This is explained by the ability of neural networks to provide new ways and techniques to successfully solve difficult problems like pattern recognition, data analysis, and control [1, 5]. These techniques are highly capable of performing complex cognitive tasks, to the extent of matching or even outperforming humans. This is due to the ability of neural networks to learn from vast amounts of data. After being trained on this original data, in the inference phase, they can make predictions on what a certain output should be for a new input.

A result of the rise of neural networks is the concept of MLaaS (Machine Learning as a Service). In MLaaS, clients can provide input to a pre-trained neural network and receive the corresponding output [32]. This means it is not necessary for clients to have their own training datasets, and perform the training or the inference themselves, thereby lowering the barrier of entry. Another benefit is that all the efforts and computing resources can be pooled to create a better model. However, a large disadvantage is that the input data has to be shared with the service provider, who also learns the output. This data can consist of very sensitive information, such as private conversations, personal images, statistics describing someone's personal life, etc. [34]. An example is presented by recent work that proposes the use of machine learning to detect health anomalies from smartwatch data [3].

Secure neural network inference (SNNI) aims to solve this problem. SNNI means that the following secrecy goals are ensured by the inference process [23]:

- The client's input remains unknown to the server.
- The output of the inference remains unknown to the server.
- The parameter values of the server's neural network (e.g., weights and biases) remain unknown to the client.

SNNI is challenging, but can be achieved by using advanced cryptographic protocols [23].

The vast majority of the research in the field so far has focused on situations where both the client and the service provider use powerful computers. However, in an MLaaS environment, this is often not the case. Here the server often has far greater resources than the client. Especially in the context of edge computing, clients often use resource-constrained devices, such as smart cameras [2, 17, 22]. For this reason, it is necessary to extend the body of knowledge on SNNI to situations with resource-constrained clients.

The aim of this paper is to investigate how current SNNI approaches perform when the client is resource-constrained, and to provide suggestions on how to better deal with such cases. To this

end, we perform an extensive empirical evaluation using two state-of-the-art SNNI approaches, Cheetah and SCI_{HE}, while having the client run on a virtual machine with varying resource allowances.

Our results indicate that current SNNI approaches allow for SNNI with resource-constrained clients up to some point, although their performance is quite sensitive to the client's capacity. Cheetah outperformed SCI_{HE} in most cases, with their performance difference depending significantly on the client's capacity. For both SNNI approaches, the execution time for different CPU allowances follows a hyperbola. When changing memory allowances, there is a threshold: configurations with lower allowances experience a major slowdown or do not run at all, while after this point no significant performance gains are achieved anymore. Delving deeper into the different layer types of the neural networks shows that different types of layers experience different slowdowns as client capacity decreases. The results also show that the required memory for the client is dependent on the largest layer in the network, but not on the depth of the network. Overall, the results give significant new insights into SNNI performance that can serve as a basis for making SNNI with resource-constrained clients more practical.

2 PRELIMINARIES

This section summarizes important background information on neural networks, the secure neural network inference (SNNI) problem, SNNI solution approaches, and the evaluation of such approaches.

2.1 Neural networks

A neural network (NN) computes a function. The input of the NN is a vector consisting of numbers. The output of the NN is usually either a number or a vector (typically of smaller dimension than the input vector). For example, if the NN is used for image classification, the input could be the raw image and the output could be an encoding of the class to which the given image belongs.

In this paper, we focus on feed-forward neural networks. A feed-forward neural network consists of a sequence of *layers*. Each layer takes a vector as input and outputs another vector of potentially different dimension. The input of the first layer is the input to the NN, while the output of the last layer is the output of the NN.

Different types of layers perform different transformations. A useful classification is to differentiate linear and non-linear layers:

- For a *linear* layer, the layer's output is a linear function of the layer's input. For example, for a *fully-connected* layer, the output vector is computed as $y = W \cdot x + b$, where x is the input vector, and W and b are parameters (W is called weight matrix and b is called bias vector). Another example of linear layers are *convolutional* layers, which have a more complicated definition but can also be reduced to scalar products with vectors of known numbers.
- For a *non-linear* layer, the layer's output is a non-linear function of the layer's input. Important examples are activation and pooling layers.
  - In an *activation* layer, the same $\mathbb{R} \to \mathbb{R}$ function is applied to each coordinate of the input; thus, the output has the same dimension as the input. A frequently used example is the ReLU function, given as $\text{ReLU}(x) = \max(0, x)$.
  - In a *pooling* layer, a window of a given size k is swept over the input and the same $\mathbb{R}^k \to \mathbb{R}$ function is applied each time to compute one coordinate of the output. An example is Max-Pooling, where the maximum over the window is computed as output.

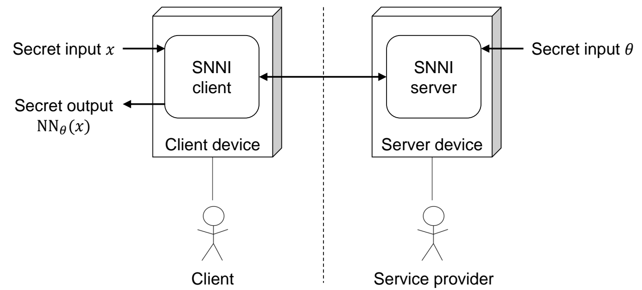

Figure 1: Overview of Secure Neural Network Inference. The service provider possesses the parameter values θ of the neural network. The client provides input x and receives output $\text{NN}_\theta(x)$ of the neural network. Input and output are only known to the client; θ is only known to the service provider.

A NN is *trained* by applying it to inputs for which the correct output is known, and tuning the parameters (e.g., weights and biases) such that the NN's output matches or approximates the expected output well. By training the NN on a large number of inputs, the NN can learn to approximate very complicated functions. An already trained NN can be used for *inference* by applying it to new inputs, yielding insights about those new inputs. For example, a NN trained for an image classification task can be used to classify new images that were not used during training.

2.2 Secure Neural Network Inference

Several versions of the Secure Neural Network Inference (SNNI) problem have been considered in the literature [23]. In the version of the problem considered in this paper, there is a neural network (NN) with a publicly known architecture. A service provider has trained the NN. The training process has identified appropriate values for all parameters in the NN (e.g., the weights and biases). The parameter values form a vector θ, which is the intellectual property of the service provider. Using the trained NN, the service provider provides inference as a service to its clients. A client can provide an input x to the NN, with the aim of receiving the output of the NN to the provided input, denoted as $\text{NN}_\theta(x)$.

As shown in Figure 1, SNNI is characterized by multiple secrecy requirements. The client may want to keep both the input and the output secret, as they may represent confidential information (e.g., personal data that must be protected according to applicable privacy laws). The service provider may want to keep the parameter values secret, to ensure that the business model can be sustained. Thus, the SNNI problem consists of computing the output of the NN in such a way that the secrecy requirements are satisfied.

2.3 SNNI solution approaches

To solve the SNNI problem, different cryptographic protocols can be used, such as homomorphic encryption (HE) or secure multi-party computation (MPC). Research in recent years focused on customizing these protocols for SNNI with the aim of reducing their performance overhead [23]. To achieve good performance, state-of-the-art SNNI approaches typically combine multiple cryptographic protocols. They evaluate the NN layer by layer, using for each layer the cryptographic protocol that is best suited for the given type of layer. The evaluation of each layer typically involves computation on both the client and the server side, as well as communication between client and server. Raw data (e.g., the input to the given layer or the parameter values of the given layer) are not transmitted, in order to adhere to the secrecy requirements. Only encrypted, masked, or otherwise obfuscated data is transmitted.

In this paper, we use two state-of-the-art SNNI solution approaches: CrypTFlow2 and Cheetah.

CrypTFlow2 uses an existing MPC approach called additive secret-sharing as overall framework [29]. For evaluating non-linear layers, CrypTFlow2 uses sophisticated and highly optimized custom protocols based on an existing MPC primitive called oblivious transfer. For linear layers, CrypTFlow2 implements two different protocols: one based on homomorphic encryption and another based on oblivious transfer. In contrast to some other SNNI approaches, CrypTFlow2's output is guaranteed to be equal to the output of "normal" (i.e., non-secure) inference. For this reason, CrypTFlow2's SNNI approach is also called Secure and Correct Inference (SCI). CrypTFlow2 provides two specific SNNI approaches, denoted as SCI_{HE} and SCI_{OT}. SCI_{HE} uses homomorphic encryption for linear layers, whereas SCI_{OT} uses oblivious transfer for linear layers. In this paper, we use SCI_{HE} because it incurs less communication and is thus more efficient in edge computing use cases [29].

Cheetah, one of the most recent SNNI approaches, is based on CrypTFlow2, and provides several improvements over CrypTFlow2 [13]. For non-linear layers, Cheetah provides improved versions of CrypTFlow2's protocols. For truncation (i.e., division by a power of 2), which is used to restore a fixed bitlength after a multiplication, Cheetah allows a small error, enabling a significant speedup of the protocol. A new protocol for oblivious transfer (called silent OT extension) is used in the protocol for comparison of numbers, which is the basis for multiple further protocols, such as for ReLU. For linear layers, Cheetah also uses homomorphic encryption as SCI_{HE}, but in a more sophisticated way, which makes some operations faster and the conversion between protocols for linear and non-linear layers smoother.

2.4 Evaluation of SNNI solution approaches

SNNI approaches have usually been evaluated in settings where both client and server devices are similarly powerful. For example, the Cheetah paper presented the results of experiments performed using two cloud servers with 2.70 GHz CPU and 16 GB RAM [13]. Often, NNs for image classification tasks are used. These NNs can vary significantly in terms of complexity, from just a couple of layers and hundreds of parameters to tens or hundreds of layers and millions of parameters. In this paper, we report results of experiments on three NNs: SqueezeNet, ResNet50, and DenseNet121.

SqueezeNet was specifically created to achieve relatively high accuracy with a limited size of less than 500 thousand parameters [14]. *ResNet50* has over 23 million parameters, and is thus significantly larger than SqueezeNet [9]. *DenseNet121* is even larger than ResNet50, featuring hundreds of layers [11].

3 METHODOLOGY

To evaluate how the SNNI approaches perform in situations with resource-constrained clients, we need to simulate client devices with different capacity. There are several ways to achieve this goal. One option would be to use a combination of Linux tools like `cpulimit`[1] to limit CPU time, disabling processor cores in the operating system, using `setrlimit`[2] to limit memory etc. Another method would be to use the Control Groups (`cgroups`[3]) kernel feature. A third option is to use Virtual Machines to limit resources.

The first option would be feasible, although it is important to make sure there are no other processes in the system that influence the experiments. Control groups are also a viable option, as this feature is designed for such purposes. However, both of these methods have a major shortcoming when it comes to restricting memory. Although they do allow memory restrictions to be placed on a process, they handle this in unrealistic ways. These methods allow for two possible actions when a process reaches its memory limit. It can either wait for memory to be released or simply kill the process. Choosing any of these options would not be realistic, as regular devices would choose to swap memory pages instead. This is where virtual machines are significantly more appropriate: they handle out-of-memory situations in a more realistic way and perform page swaps. This is why we chose this method.

Thus, in this paper, measurements are performed by running the client of the SNNI process in a virtual machine with restricted resources, and the server on the host machine with unrestricted resources. The exact specifications of the (virtual) machine can be found at the end of this section.

Time measurement. The goal of this paper is to find out what the real-world impact is of having a resource-constrained client. We are interested in the scenario with just 2 parties, where data is immediately available. With these goals in mind, wall-clock time is chosen as the metric to measure execution time. More specifically, the time is measured by running the Linux `time` command in the virtual machine after connecting to this machine via `ssh`.

Experimental setup. All tests were run on a Xiaomi Mi Notebook Air 13.3 2018, a laptop with an Intel Core i7-8550U CPU and 8GiB of memory. The machine was running Ubuntu 20.04 LTS and so was the virtual machine. The virtual machine was run using VirtualBox version 6.1.38_Ubuntu r153438. The virtual machines were all configured with 2GiB of swap space. For Cheetah, the latest available software was used at the time of writing (commit 0b63d6f2cfe979a446a7999ee78d705b6ef5ab81 of the OpenCheetah library on GitHub[4]). For SCI_{HE}, the version that was used for testing is the updated version by [13], available in the same repository.

[1]https://manpages.ubuntu.com/manpages/trusty/man1/cpulimit.1.html
[2]https://linux.die.net/man/2/setrlimit
[3]https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/6/html/resource_management_guide/ch01
[4]https://github.com/Alibaba-Gemini-Lab/OpenCheetah

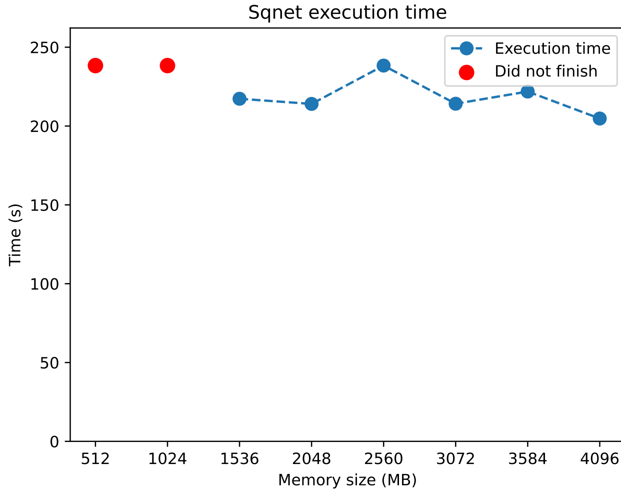

Figure 2: Execution time of the SCI$_{HE}$ client running SqueezeNet for different memory configurations, running on a single core

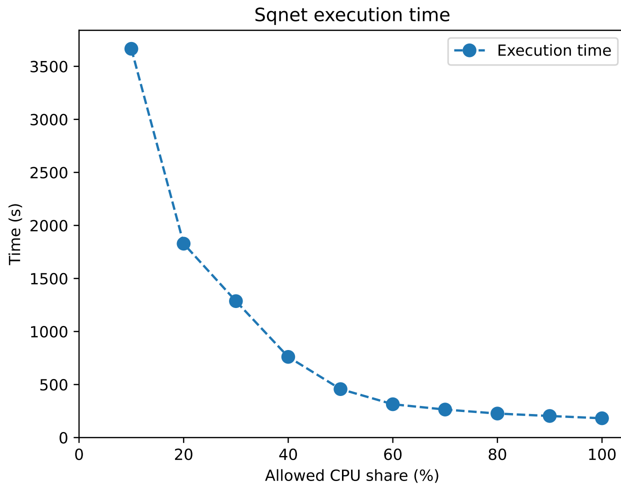

Figure 3: Execution time of the SCI$_{HE}$ client running SqueezeNet with different CPU allowances, running on a single core with 4096MB of memory

4 RESULTS

This section presents the results of our experiments, first with SCI$_{HE}$ and then with Cheetah.

4.1 SCI$_{HE}$

The first NN to be tested is SqueezeNet, using SCI$_{HE}$. The results of constraining the client's memory and CPU are shown in figs. 2 and 3, respectively.

As fig. 2 shows, SqueezeNet under SCI$_{HE}$ requires at least 1.5GiB of memory, or will not finish. If it gets less than this amount, the process will be terminated as it runs out of both memory and swap space. Above 1.5GiB of memory, the process benefits very little, if at all, from extra memory. The graph in fig. 3, showing the execution

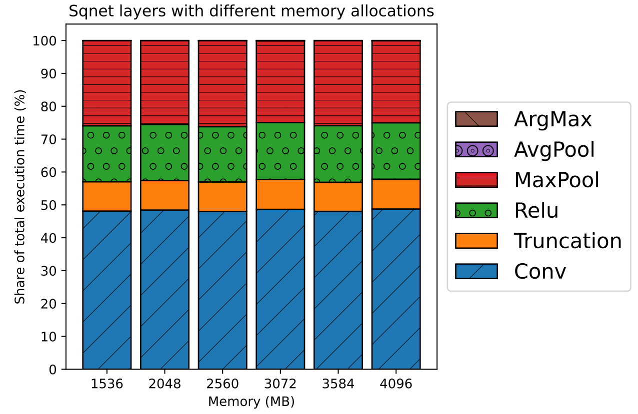

Figure 4: SCI$_{HE}$ client execution time distribution for different memory allocations while executing SqueezeNet

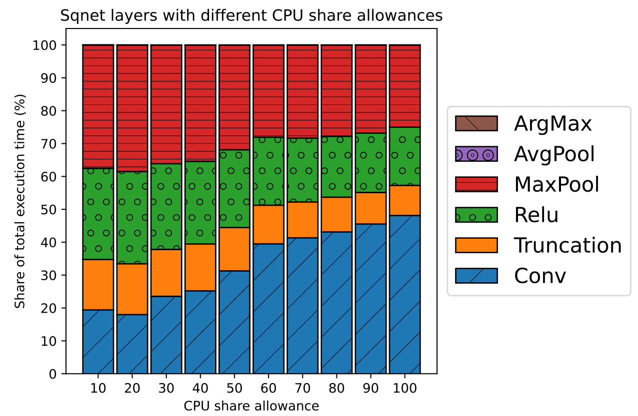

Figure 5: SCI$_{HE}$ client execution time distribution for different CPU allowances while executing SqueezeNet

time for different CPU allowances, looks like a hyperbole. This is not accidental: if the inference process requires N instructions to be executed and the machine has a capacity of C instructions per second, it takes $t = N/C$ seconds to execute the process. if N is constant, this yields a hyperbola for t as a function of C.

Looking more into the details, we can decompose the execution time into the execution times per layer type. The corresponding relative execution times per layer type are shown in figs. 4 and 5.

In these figures it is clear that the time spent calculating `ArgMax` and `AvgPool` layers in SqueezeNet using SCI$_{HE}$ is negligible. For the memory sizes that ran successfully, SCI$_{HE}$ does not have significant differences in the time distribution over different layer types.

This is different when looking at the CPU chart (fig. 5), however. The share of time spent executing convolutional layers decreases as we allocate less processing power to SCI$_{HE}$. This means that convolutional layers are less limited by a weaker CPU compared to other layers in the network.

Inference with the larger ResNet50 and DenseNet121 NNs did not finish in any of the tests with SCI$_{HE}$. Even in the most relaxed case with 4096MB memory and 100% CPU allowance, neither was

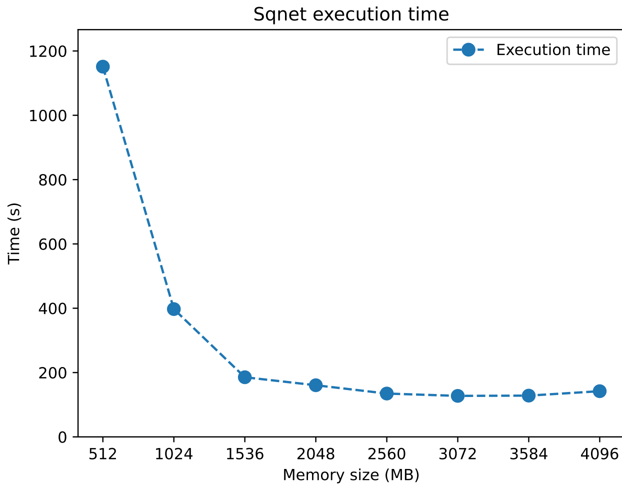

Figure 6: Execution time of the Cheetah client running SqueezeNet with different memory sizes, on a single core

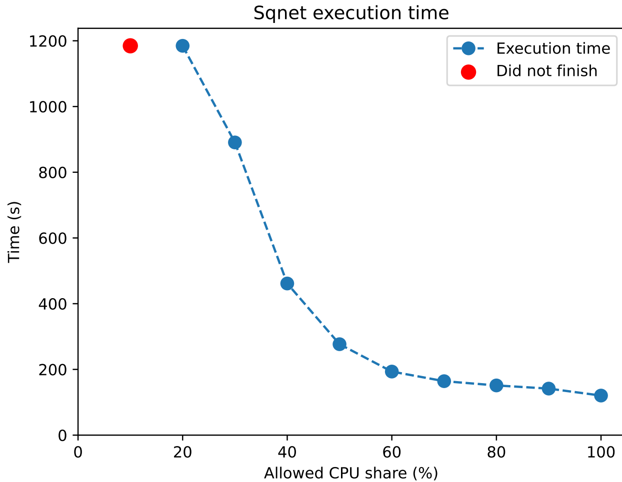

Figure 7: Execution time of the Cheetah client running SqueezeNet with different CPU allowances, running on a single core with 4096MB of memory

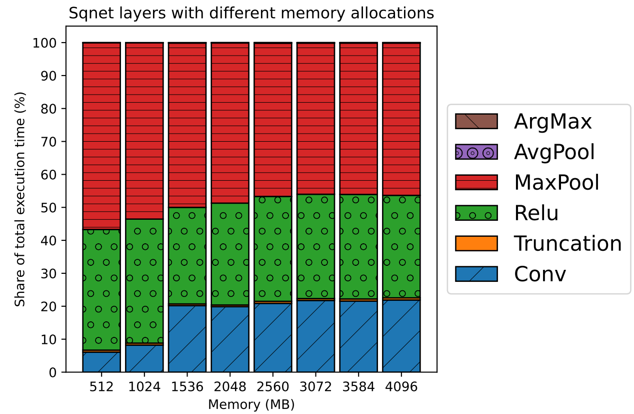

Figure 8: Cheetah client execution time distribution for different memory allocations while executing SqueezeNet

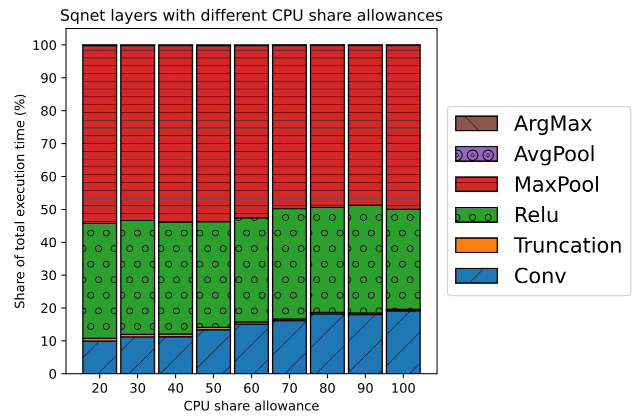

Figure 9: Cheetah client execution time distribution for different CPU allowances while executing SqueezeNet

able to reach the end of the inference. Both networks required too many resources and froze after a few layers. Even after multiple hours, the inference did not progress.

4.2 Cheetah

4.2.1 SqueezeNet. The results of running Cheetah on SqueezeNet, with varied client-side memory and CPU, are shown in fig. 6 and fig. 7, respectively. Cheetah seems to use around 1.5 to 2GB of memory when executing SqueezeNet, which explains the sharp drop in execution time until that point, and the diminishing returns afterward. Page faults start to appear when there is too little memory available, resulting in very slow memory accesses and thus to a major slowdown of the whole process. In terms of absolute execution time, Cheetah clearly outperforms the previously tested SCI_{HE}. For similar memory configurations, Cheetah shows a decrease in total execution time ranging from 14.7% to 43.5%. These specific cases are for memory allocations of 1536MB and 2560MB, respectively.

The CPU chart (fig. 7) is similar to fig. 3. The execution times of Cheetah are significantly lower compared to SCI_{HE}. More specifically, the execution time is between 30.4% and 39.4% lower for comparable CPU allocations. The smallest speedup occurred with 90% CPU, and the highest occurred with a CPU allowance of 50%. The fact that Cheetah outperforms SCI_{HE} is plausible, as Cheetah was released after SCI_{HE} and is supposed to improve upon its predecessor. However, unlike SCI_{HE}, Cheetah fails to finish inference at 10% CPU allocation. At the second to last layer of the NN, Cheetah consistently freezes and will not move on, even after hours.

Looking into the details, the relative execution time for different layer types in SqueezeNet are shown in figs. 8 and 9, when restricting memory and CPU, respectively. When comparing these two figures with the corresponding SCI_{HE} plots (figs. 4 and 5), we can see that the share of `Truncation` and `Convolutional` layers shrunk heavily. In addition, Figure 6 shows that Cheetah running SqueezeNet is being bottlenecked by memory when there is less

Figure 10: Execution time of the Cheetah client running ResNet50 with different memory sizes, on a single core

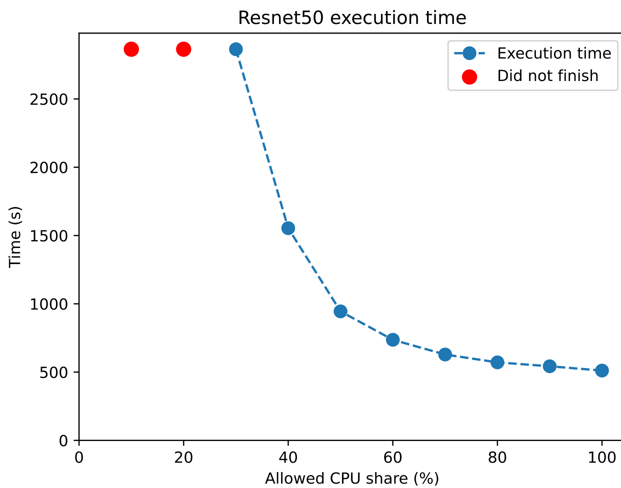

Figure 11: Execution time of the Cheetah client running ResNet50 with different CPU allowances, on a single core

than 1536MB available. In the two cases where this holds, we can see that the non-linear `MaxPool` and `ReLU` layers take over a larger share of the execution time. The share of convolutional layers decreases. This means that page swaps and slow memory accesses especially impact non-linear layers. For runs with more than 1536MB of memory, there are no major differences in the distribution.

With respect to the CPU, while convolutional layers still see a decrease in the share of execution time as the CPU allowance goes down, the difference compared to SCI_{HE} in fig. 5 is much less pronounced. Cheetah's absolute execution times are always lower than SCI_{HE}, but the difference gets very small as the CPU allowance grows. This is true for all layer types, except for convolutional layers. Those are still a few times faster than the SCI_{HE} implementation.

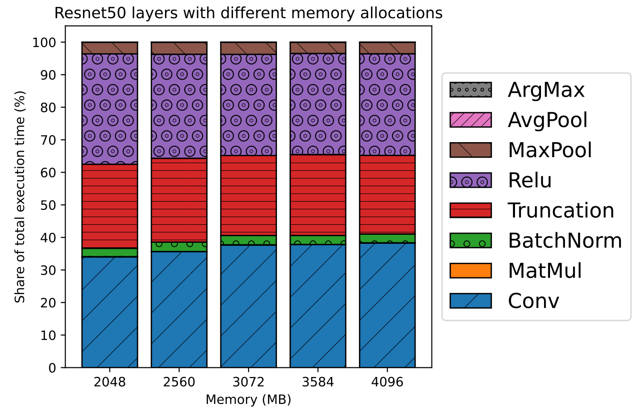

Figure 12: Cheetah client execution time distribution with different memory sizes while executing ResNet50

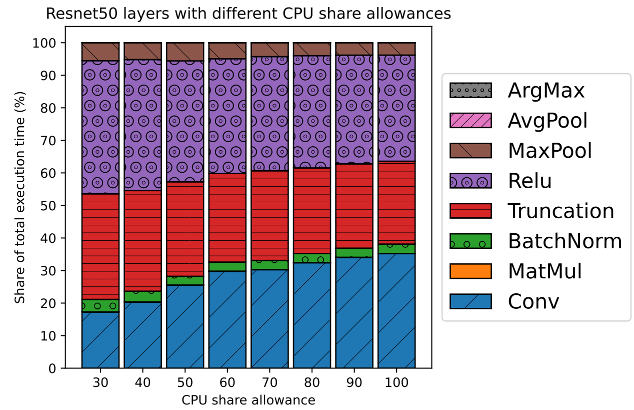

Figure 13: Cheetah client execution time distribution for different CPU allowances while executing ResNet50

4.2.2 ResNet50. Inference with ResNet50 is significantly more resource-intensive and time-consuming than SqueezeNet. While for SqueezeNet, Cheetah was able to finish inference on a machine with 512MB memory, for ResNet50, any amount of memory lower than 2048MB will see the process killed by the Linux kernel because both memory and swap space have been exhausted. From 2560MB of memory, there are no significant performance improvements (see fig. 10). The issue of the inference freezing consistently happens for both 10% and 20% of CPU allowance with ResNet50 (see fig. 11), instead of just for 10% for SqueezeNet. For the successful runs, the curve resembles a hyperbole again, just like for SqueezeNet.

The breakdown into layer types of ResNet50 shows some differences from the previously discussed SqueezeNet. Since ResNet50 has higher memory requirements to run at all, it sees fewer page swaps. Thus, memory size hardly impacts the time distribution, as seen in fig. 12. With respect to the CPU, the non-linear layers again see a growth in share as the CPU allowance decreases (see fig. 13).

4.2.3 DenseNet121. In terms of execution time, DenseNet121 is similar to ResNet50. It also does not finish for memory sizes smaller or equal to 1536MB and sees a bit of a drop in execution time after

Figure 14: Execution time of the Cheetah client running DenseNet121 with different memory sizes, on a single core

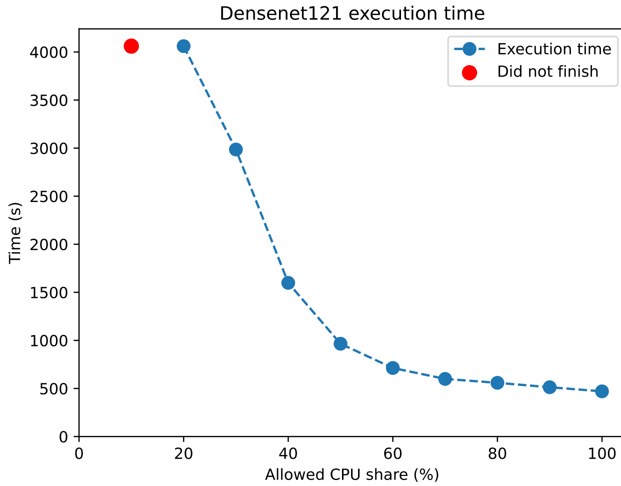

Figure 15: Execution time of the Cheetah client running DenseNet121 with different CPU allowances, running on a single core

2048MB (see fig. 14). Unlike ResNet50, DenseNet121 does finish inference with a 20% CPU share. Other than that, the CPU curve (see fig. 15) is similar to the earlier ones.

The DenseNet121 layer distribution shows similar behavior as SqueezeNet and ResNet50. As little page swapping happens for the inference runs that finished successfully, the distributions for different memory allocations are very similar to one another (see fig. 16). The share of linear layers decreases as CPU allowance goes down, and the share of non-linear layers increases. In particular, ReLU layers have a considerable share in DenseNet121 (see fig. 17).

5 DISCUSSION

The results presented in the previous section show that Cheetah has an execution time that is between 14.7% and 43.5% lower than

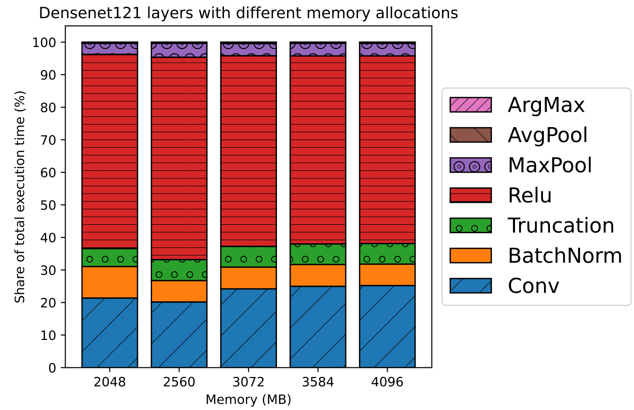

Figure 16: Cheetah client execution time distribution with different memory sizes while executing DenseNet121

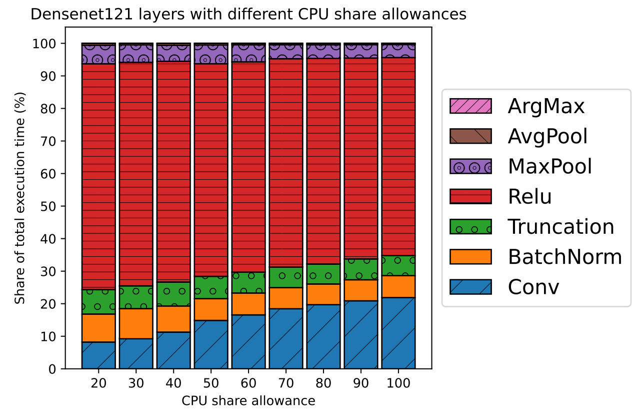

Figure 17: Cheetah client execution time distribution for different CPU allowances while executing DenseNet121

SCI_{HE} for different memory configurations. For different CPU configurations, this range is narrower: Cheetah was between 30.4% and 39.4% faster than SCI_{HE}. Cheetah also has lower minimum hardware requirements to run inference at all as it ran ResNet50 and DenseNet121 with 2GiB of memory, whereas SCI_{HE} could not finish evaluating these NNs with even 4GiB.

Another important aspect that has become clear is that non-linear layers take a larger hit to their execution time than linear layers when decreasing the CPU capacity. Nonlinear layers like ReLU and MaxPool are easy to evaluate if the numbers are available in plaintext, but for encrypted or secret-shared numbers, evaluating these layers requires sophisticated and resource-intensive protocols. In contrast, linear computations (required e.g. for convolutional or average pooling layers) can be performed directly on homomorphically encrypted or additively secret-shared numbers, thus making such layers less resource-intensive than nonlinear layers.

Taking a least squares approximation (see fig. 18) shows that, on average, the percentage of time executing the linear convolutional layers decreases by 3.6 for every 10% decrease in CPU share allowance for SCI_{HE} executing SqueezeNet. This decrease is less pronounced for Cheetah: here, a drop of 1.2, 2.6, and 1.8 percentage

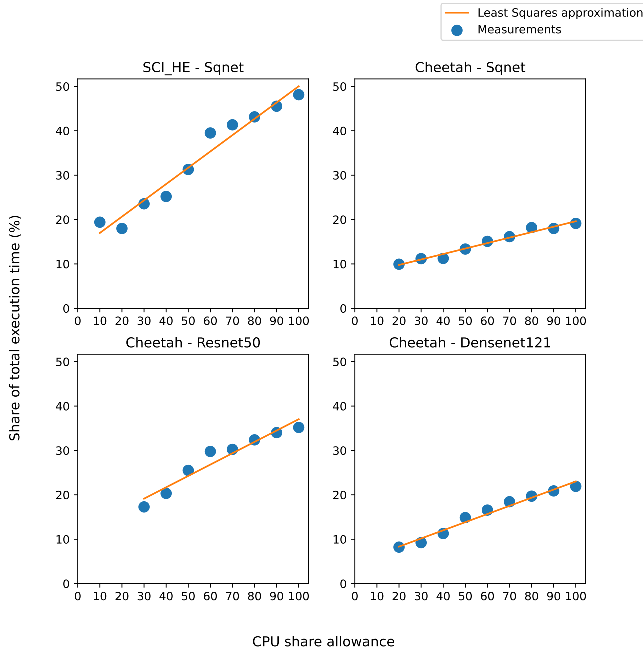

Figure 18: Share of execution time for convolutional layers

points are observed for SqueezeNet, ResNet50, and DenseNet121, respectively. Interestingly, the gradients for the three different networks running in Cheetah are quite different. This may be attributed to the fact that some NNs have a larger share of convolutional layers; the percentage of time executing these layers will decrease more drastically than if there were fewer convolutional layers. SqueezeNet has fewer convolutional layers than DenseNet121, which in turn has fewer convolutional layers than ResNet50, and this is in line with the different sensitivity of their share of convolutional layers in execution time to resource limitations.

As already mentioned, Cheetah has lower memory requirements across all three tested NNs than SCI_{HE}. When running Cheetah, a certain amount of memory is necessary to complete inference, and there is always a threshold amount where getting a larger amount of memory does not offer any significant performance benefits anymore. When increasing memory past this point, the distribution of work across the different layer types does not change anymore either. If page swaps are necessary, this takes its largest toll on the non-linear layers, just like restricting the CPU does. The amount of required memory for a given NN is dependent on the types and sizes of layers in the network. After finishing the execution of a given layer, both Cheetah and SCI_{HE} release the memory used for the given layer and just store the results. These are then used as input for the next layer. The depth of the network, therefore, has no effect on the ability to complete inference. DenseNet121 using SCI_{HE}, for example, consistently crashes at the execution of the second convolutional layer. The first convolutional layer runs without errors, but the second one doubles the number of input channels from 64 to 128 and increases the filter size from 1x1 to 3x3. This requires extra computing and extra storage of the already computed elements in the layer. This is too much to handle for the

Table 1: The performance of a range of devices and their closest tested equivalent in the paper

Device	Geekbench Score	RAM	Closest tested equivalent
Raspberry Pi 3 Model B (2018)	108[6]	1GB	1GB RAM, 10% CPU
Raspberry Pi 4 Model B (4GB) (2019)	296[7]	4GB	4GB RAM, 30% CPU
Samsung Chromebook Pro (2017)	432[8]	4GB	4GB RAM, 50% CPU
Samsung Galaxy A23 5G (2022)	654[9]	4GB	4GB RAM, 70% CPU

limited 4096MB of memory. The reason this issue is present for SCI_{HE} but not for Cheetah, is that Cheetah improved the evaluation of convolutional layers. The different way of encoding the data and the use of silent OT really makes the difference here.

5.1 Real devices

It is interesting to see how the results of our experiments can be transferred to approximate SNNI performance on real devices. For this purpose, benchmarking can be used, as benchmarks can give a rough indication of the relative performance of different devices [18]. We use Geekbench V5, a widely available benchmark that runs on many different devices. As our experiments were carried out using one CPU core, we use the single-core score of Geekbench. The machine that we used for testing has a Geekbench score of 949[5]. The performance of some other devices commonly found in edge computing settings, such as IoT devices and budget smartphones, and their closest tested equivalent in this paper are shown in table 1.

From the table, we can deduce that a Raspberry Pi 3-B would likely struggle to run any of the tested NNs. The most popular variant of its successor [8], the Raspberry Pi 4-B 4GB would be able to perform SNNI with all tested NNs using Cheetah, albeit at a rather slow rate. For a common Chromebook (the Samsung Chromebook Pro) and a modern mid-range phone (the Samsung Galaxy A23 5G), SNNI using Cheetah and the three tested NNs would be somewhat faster, but would still take several minutes.

5.2 Recommendations

Based on our findings, we can identify important recommendations for future work on SNNI. First, more research is needed to make SNNI less resource-hungry specifically on the client side. Efforts to make SNNI overall less resource-hungry have been made and should continue. But in addition, and this is a novel observation, also techniques to shift work from the client to the server should be considered. SNNI approaches were so far evaluated in symmetric environments (where client and server had similar capacity), leading

[5]https://browser.geekbench.com/v5/cpu/21358797
[6]https://browser.geekbench.com/v5/cpu/21356250
[7]https://browser.geekbench.com/v5/cpu/21356296
[8]https://browser.geekbench.com/v5/cpu/21306330
[9]https://browser.geekbench.com/v5/cpu/21122210

to an implicit incentive to balance the work between client and server. By focusing on asymmetric situations (where the client is weaker), work should be split unevenly between client and server. This has ramifications on the cryptographic protocols to select, as they are characterized by different levels of asymmetry between the parties. Also, there should be more focus on improving non-linear layers, as they are more critical on resource-constrained clients.

Second, also the choice of NN should account for resource-constrained clients. We have seen that, when dealing with memory-constrained clients, it is better to use deep but narrow NNs instead of wide and shallow NNs. The size of non-linear layers is especially critical. If we have the choice between multiple NNs offering similar accuracy, the NN with smaller non-linear layers is to be preferred.

Third, it is important to recognize that an SNNI service needs to serve clients with different capacity. This may make it reasonable to determine dynamically on a per-client basis which protocols to use, which NN to use, and how to set other hyperparameters (e.g., bitlength). How to do this is a completely new research direction.

Finally, further work could extend our experiments, e.g., to other NNs, other metrics beyond wall-clock time, other benchmarking methods, and a more detailed statistical evaluation.

6 RELATED WORK

In this section, we review previous efforts on SNNI in general and SNNI in resource-constrained environments.

Work on SNNI. Early attempts to solve the SNNI problem date back to 2006-2007, when Barni, Orlandi, and Piva suggested using homomorphic encryption for this purpose [4, 26]. These early protocols incurred a large performance overhead while also leaking sensitive information. It took a decade until the potential for practical SNNI could be demonstrated in the seminal paper of Gilad-Bachrach et al. introducing the CryptoNets approach [7]. With a carefully crafted NN of 5 layers, CryptoNets could perform secure inference in a couple of minutes, with an accuracy of 99% on the MNIST image classification task.

Instead of homomorphic encryption, other researchers suggested applying existing secure multi-party protocols to solve the SNNI problem. DeepSecure was probably the first such approach, using the existing Garbled Circuits MPC protocol [33]. To mitigate the huge performance overhead, DeepSecure also used non-cryptographic methods, such as model pruning, to reduce the amount of computation. The XONN approach also used Garbled Circuits, and to reduce the performance overhead, it restricted the NN to only allow the numbers 1 and −1 [30]. The secure evaluation of such restricted NNs using the Garbled Circuits protocol is much faster. However, it is challenging to maintain high accuracy with such restricted NNs.

Further progress was mainly driven by the observation that different protocols are appropriate for securely evaluating different types of layers. Several researchers experimented with different combinations of protocols and devised faster and faster SNNI approaches. Early approaches based on this idea include SecureML [25], MiniONN [20], Chameleon [31], and Gazelle [15].

In the last couple of years, researchers have devised more specialized and sophisticated cryptographic protocols for evaluating typical NN layer types. This way, the performance overhead of SNNI has decreased significantly. Typical examples of such approaches include Delphi [24], CrypTFlow [16], SiRnn [28], as well as the already described CrypTFlow2 and Cheetah.

SNNI in resource-constrained environments. A common problem of the cryptography-based SNNI approaches described above is their high performance overhead. Thus, running SNNI on large NNs in a resource-constrained environment is challenging.

Some researchers suggested using other, less resource-hungry methods. For example, model splitting was suggested [27, 35]. In model splitting, the client evaluates the first k layers of the NN and sends the output of layer k to the server, which completes the inference and returns the output to the client. Model splitting offers some data protection: the server does not learn the raw input data, and the client only learns the parameter values of the first k layers, but nothing about the remaining layers. Also, model splitting does not require computation- and communication-intensive cryptographic protocols. However, the data protection offered by model splitting is actually very weak: unless k is large (which would leak much information to the client), the server can reconstruct the input quite well [27].

Other, lightweight solutions to the problem have also been explored. However, these make different assumptions, such as requiring a pre-processing phase or a third (neutral and trustworthy) party [21], or compromise on the secrecy goals of SNNI, such as by sharing the neural network [19].

A different approach is to run the inference entirely on the client device, using hardware-based protection in the form of a trusted execution environment (TEE) [10]. A TEE is a part of the computer's processor and memory that is protected by the hardware even against processes of the highest privilege level (e.g., processes of the operating system's kernel). By running the inference process in a client-side TEE, the client's secrecy goals are fulfilled because the client's data do not leave the client device, and the service provider's secrecy goals are also fulfilled because client-side processes cannot access NN parameter values stored in the TEE. This approach is relatively efficient because cryptographic operations benefit from a hardware implementation and communication is minimized. However, this approach only works if the client device supports TEEs and is powerful enough to perform the evaluation of the NN on its own – assumptions that are often violated for resource-constrained devices and large NNs.

Finally, we would like to mention the approach of [12]. This approach does use expensive cryptographic protocols. The idea is that those cryptographic protocols are executed between two edge servers, so that the computational demand on the client side is minimal. However, for this approach to be secure, the two edge servers must not collude, which is difficult to guarantee in practice. In addition, the approach also assumes the existence of a trusted third party in a setup phase.

Altogether, although SNNI would be clearly needed in resource-constrained environments [6], existing solution approaches are either too resource-hungry, not secure enough, or only work under very specific conditions. Devising efficient, secure, and widely applicable SNNI approaches for resource-constrained environments is thus still an important research challenge. Our work is a first step into this direction.

7 CONCLUSIONS AND FUTURE WORK

In the past, the evaluation of SNNI approaches was limited to setups with strong devices on both client and server side. Since in practice, client devices are often resource-constrained, this paper focused on evaluating the effects of client-side resource limitations on SNNI performance. We carried out controlled experiments using two state-of-the-art SNNI approaches, CrypTFlow2 and Cheetah, and three complex neural networks. We varied the CPU and RAM capacity of the client, and measured the SNNI execution time. Our results demonstrate that client-side resource limitations have indeed profound effects on SNNI performance. Limited resources on the client side can make the inference process much slower or even lead to failure. Our results also give detailed insights into the sensitivity of different combinations of protocol and NN layer type to CPU and RAM limitations. These insights can be exploited in various ways in future research. By identifying performance bottlenecks, our insights can inform the development and tuning of future SNNI approaches to make them more appropriate for resource-constrained clients. These insights should also be taken into account in the selection of NNs for SNNI with resource-constrained clients. Moreover, in scenarios where new clients emerge on the fly, information about client capabilities makes it possible to dynamically select the most appropriate NNs and SNNI protocols.

REFERENCES

[1] Laith Alzubaidi, Jinglan Zhang, Amjad J. Humaidi, Ayad Al-Dujaili, Ye Duan, Omran Al-Shamma, J. Santamaría, Mohammed A. Fadhel, Muthana Al-Amidie, and Laith Farhan. 2021. Review of deep learning: concepts, CNN architectures, challenges, applications, future directions. *Journal of Big Data* 8, 1 (2021).

[2] Dhouha Ayed, Paul-Andrei Dragan, Edith Félix, Zoltán Adám Mann, Eliot Salant, Robert Seidl, Anestis Sidiropoulos, Steve Taylor, and Ricardo Vitorino. 2022. Protecting sensitive data in the cloud-to-edge continuum: The FogProtect approach. In *22nd International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*. IEEE, 279–288.

[3] Ruhi Kiran Bajaj, Rebecca Mary Meiring, and Fernando Beltran. 2023. Co-Design, Development, and Evaluation of a Health Monitoring Tool Using Smartwatch Data: A Proof-of-Concept Study. *Future Internet* 15, 3 (2023), 111. https://doi.org/10.3390/fi15030111

[4] Mauro Barni, Claudio Orlandi, and Alessandro Piva. 2006. A privacy-preserving protocol for neural-network-based computation. In *Proceedings of the 8th Workshop on Multimedia and Security*. 146–151.

[5] Chris M. Bishop. 1994. Neural networks and their applications. *Review of Scientific Instruments* 65, 6 (1994), 1803–1832. https://doi.org/10.1063/1.1144830

[6] Daphnee Chabal, Dolly Sapra, and Zoltán Ádám Mann. 2023. On Achieving Privacy-Preserving State-of-the-Art Edge Intelligence. In *4th AAAI Workshop on Privacy-Preserving Artificial Intelligence (PPAI-23)*.

[7] Ran Gilad-Bachrach, Nathan Dowlin, Kim Laine, Kristin Lauter, Michael Naehrig, and John Wernsing. 2016. CryptoNets: Applying neural networks to encrypted data with high throughput and accuracy. In *33rd International Conference on Machine Learning*. 201–210.

[8] Gareth Halfacree. 2020. Raspberry Pi 4 B: How Much RAM Do You Really Need? https://www.tomshardware.com/news/raspberry-pi-4-how-much-ram

[9] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 770–778.

[10] Jiahui Hou, Huiqi Liu, Yunxin Liu, Yu Wang, Peng-Jun Wan, and Xiang-Yang Li. 2021. Model Protection: Real-time privacy-preserving inference service for model privacy at the edge. *IEEE Transactions on Dependable and Secure Computing* 19, 6 (2021), 4270–4284.

[11] Gao Huang, Zhuang Liu, Laurens Van Der Maaten, and Kilian Q Weinberger. 2017. Densely connected convolutional networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 4700–4708.

[12] Kai Huang, Ximeng Liu, Shaojing Fu, Deke Guo, and Ming Xu. 2019. A lightweight privacy-preserving CNN feature extraction framework for mobile sensing. *IEEE Transactions on Dependable and Secure Computing* 18, 3 (2019), 1441–1455.

[13] Zhicong Huang, Wen-jie Lu, Cheng Hong, and Jiansheng Ding. 2022. Cheetah: Lean and fast secure two-party deep neural network inference. In *31st USENIX Security Symposium*. 809–826.

[14] Forrest N Iandola, Song Han, Matthew W Moskewicz, Khalid Ashraf, William J Dally, and Kurt Keutzer. 2016. SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and < 0.5 MB model size. arXiv preprint arXiv:1602.07360.

[15] Chiraag Juvekar, Vinod Vaikuntanathan, and Anantha Chandrakasan. 2018. GAZELLE: A low latency framework for secure neural network inference. In *27th USENIX Security Symposium (USENIX Security 18)*. 1651–1669.

[16] Nishant Kumar, Mayank Rathee, Nishanth Chandran, Divya Gupta, Aseem Rastogi, and Rahul Sharma. 2020. CrypTFlow: Secure TensorFlow inference. In *IEEE Symposium on Security and Privacy (SP)*. IEEE, 336–353.

[17] Clemens Lachner, Zoltán Adám Mann, and Schahram Dustdar. 2021. Towards understanding the adaptation space of AI-assisted data protection for video analytics at the edge. In *41st International Conference on Distributed Computing Systems Workshops (ICDCSW)*. IEEE, 7–12.

[18] Byron C. Lewis and Albert E. Crews. 1985. The Evolution of Benchmarking as a Computer Performance Evaluation Technique. *MIS Quarterly* 9, 1 (1985), 7.

[19] Jiarui Li, Zhuosheng Zhang, Shucheng Yu, and Jiawei Yuan. 2022. Improved Secure Deep Neural Network Inference Offloading with Privacy-Preserving Scalar Product Evaluation for Edge Computing. *Applied Sciences* 12, 18 (2022), 9010.

[20] Jian Liu, Mika Juuti, Yao Lu, and Nadarajah Asokan. 2017. Oblivious neural network predictions via MiniONN transformations. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 619–631.

[21] Xiaoning Liu, Yifeng Zheng, Xingliang Yuan, and Xun Yi. 2023. Securely Outsourcing Neural Network Inference to the Cloud With Lightweight Techniques. *IEEE Transactions on Dependable and Secure Computing* 20, 1 (2023), 620–636. https://doi.org/10.1109/tdsc.2022.3141391

[22] Zoltán Ádám Mann, Andreas Metzger, Johannes Prade, and Robert Seidl. 2019. Optimized application deployment in the fog. In *17th International Conference on Service-Oriented Computing (ICSOC)*. Springer, 283–298.

[23] Zoltán Ádám Mann, Christian Weinert, Daphnee Chabal, and Joppe W. Bos. 2023. Towards Practical Secure Neural Network Inference: The Journey So Far and the Road Ahead. *Comput. Surveys* (2023), accepted.

[24] Pratyush Mishra, Ryan Lehmkuhl, Akshayaram Srinivasan, Wenting Zheng, and Raluca Ada Popa. 2020. Delphi: A Cryptographic Inference Service for Neural Networks. In *USENIX Security Symposium*. 2505–2522.

[25] Payman Mohassel and Yupeng Zhang. 2017. SecureML: A system for scalable privacy-preserving machine learning. In *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE, 19–38.

[26] Claudio Orlandi, Alessandro Piva, and Mauro Barni. 2007. Oblivious neural network computing via homomorphic encryption. *EURASIP Journal on Information Security* 2007 (2007), 1–11.

[27] Seyed Ali Osia, Ali Shahin Shamsabadi, Sina Sajadmanesh, Ali Taheri, Kleomenis Katevas, Hamid R Rabiee, Nicholas D Lane, and Hamed Haddadi. 2020. A hybrid deep learning architecture for privacy-preserving mobile analytics. *IEEE Internet of Things Journal* 7, 5 (2020), 4505–4518.

[28] Deevashwer Rathee, Mayank Rathee, Rahul Kranti Kiran Goli, Divya Gupta, Rahul Sharma, Nishanth Chandran, and Aseem Rastogi. 2021. SiRnn: A math library for secure RNN inference. In *IEEE Symposium on Security and Privacy (SP)*. IEEE, 1003–1020.

[29] Deevashwer Rathee, Mayank Rathee, Nishant Kumar, Nishanth Chandran, Divya Gupta, Aseem Rastogi, and Rahul Sharma. 2020. CrypTFlow2: Practical 2-party secure inference. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 325–342.

[30] M Sadegh Riazi, Mohammad Samragh, Hao Chen, Kim Laine, Kristin Lauter, and Farinaz Koushanfar. 2019. XONN: XNOR-based oblivious deep neural network inference. In *28th USENIX Security Symposium*. 1501–1518.

[31] M Sadegh Riazi, Christian Weinert, Oleksandr Tkachenko, Ebrahim M Songhori, Thomas Schneider, and Farinaz Koushanfar. 2018. Chameleon: A hybrid secure computation framework for machine learning applications. In *Proceedings of the 2018 Asia Conference on Computer and Communications Security*. 707–721.

[32] Mauro Ribeiro, Katarina Grolinger, and Miriam A.M. Capretz. 2015. MLaaS: Machine Learning as a Service. In *2015 IEEE 14th International Conference on Machine Learning and Applications (ICMLA)*. IEEE.

[33] Bita Darvish Rouhani, M Sadegh Riazi, and Farinaz Koushanfar. 2018. DeepSecure: Scalable provably-secure deep learning. In *55th Annual Design Automation Conference*.

[34] Tjerk Timan and Zoltan Mann. 2021. Data protection in the era of artificial intelligence: trends, existing solutions and recommendations for privacy-preserving technologies. In *The Elements of Big Data Value: Foundations of the Research and Innovation Ecosystem*. Springer, 153–175.

[35] Mengyao Zheng, Dixing Xu, Linshan Jiang, Chaojie Gu, Rui Tan, and Peng Cheng. 2019. Challenges of privacy-preserving machine learning in IoT. In *1st Intl. Workshop on Challenges in Artificial Intelligence and Machine Learning for Internet of Things*. 1–7.