eNNclave: Offline Inference with Model Confidentiality

Alexander Schlögl
alexander.schloegl@uibk.ac.at
University of Innsbruck
Innsbruck, Austria

Rainer Böhme
rainer.boehme@uibk.ac.at
University of Innsbruck
Innsbruck, Austria

ABSTRACT

Outsourcing machine learning inference creates a confidentiality dilemma: either the client has to trust the server with potentially sensitive input data, or the server has to share his commercially valuable model. Known remedies include homomorphic encryption, multi-party computation, or placing the entire model in a trusted enclave. None of these are suitable for large models. For two relevant use cases, we show that it is possible to keep all confidential model parameters in the last (dense) layers of deep neural networks. This allows us to split the model such that the confidential parts fit into a trusted enclave on the client side. We present the eNNclave toolchain to cut TensorFlow models at any layer, splitting them into public and enclaved layers. This preserves TensorFlow's performance optimizations and hardware support for public layers, while keeping the parameters of the enclaved layers private. Evaluations on several machine learning tasks spanning multiple domains show that fast inference is possible while keeping the sensitive model parameters confidential. Accuracy results are close to the baseline where all layers carry sensitive information and confirm our approach is practical.

CCS CONCEPTS

• **Theory of computation** → *Adversarial learning*; • **Security and privacy** → *Hardware security implementation*; Authentication; • **Computer systems organization** → **Heterogeneous (hybrid) systems**; **Neural networks**.

ACM Reference Format:
Alexander Schlögl and Rainer Böhme. 2020. eNNclave: Offline Inference with Model Confidentiality. In *13th ACM Workshop on Artificial Intelligence and Security (AISec'20), November 13, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3411508.3421376

1 INTRODUCTION

A core problem of outsourcing machine learning (ML) tasks to the cloud is that the protection goals of the server and the client are in direct conflict. The server needs to keep the parameters of the ML model private to retain competitive advantage or sustain business. Clients want to use the server's model to make inferences on sensitive data of their own. Common practice is that clients share their data with the server, who are thereby forced to accept the risks of lost confidentiality.

Publishing the parameters of an ML model in production is not advisable for another reason: it would facilitate a number of attacks studied in the rapidly evolving field of adversarial machine learning [4, 31, 38, 40, 45]. A reasonably effective defense against these threats at inference time is to turn the *white-box* access into a *black-box*: the attacker can query the model as an oracle but learns nothing more than its output, in our case a class label [32].

Previous works have dealt with how to best protect the client's data when it is transmitted to the server, involving homomorphic encryption [13, 14], multiparty computation [27], and trusted processors [16, 29]. The former two incur substantial computation and communication overhead. The latter is constrained by the size of the trusted processor. While these methods may work for small neural networks, they cannot cope with larger (and more complex) deep neural networks (DNNs).

Our approach relates to [16, 29] for using trusted enclaves, but brings the inference to the client. Unlike previous works, we offer a configurable split between public and private parts of the network, and enclave only the private part. Splitting the model this way gives us a novel sweet spot: sensitive parameters in the private layers remain confidential while hardware accelerators, such as GPUs, can handle the (computationally more demanding) public part.[1]

The split crucially depends on a phenomenon observed in *transfer learning*. Transfer learning is a paradigm that proceeds by taking a DNN pre-trained for a given domain, and retraining it for a specialized task [30]. Research suggests that the first layers tend to remain stable feature extractors, whereas the final (often dense) layers, which combine the features for task-specific decisions, change drastically during retraining [39, 53]. For classification problems we can thus split the model into a general *feature extractor* and a task-specific *classifier*.

By choosing a pre-trained and publicly available feature extractor and keeping its parameters fixed during retraining, we can *guarantee* that it does not contain information about the sensitive classifier or the training data. Such publicly known feature extractors are not confidential and do not require protection. Conversely, all sensitive information is contained in the final classifier. To fulfill our goal of moving the black-box offline, it is then enough to protect the parameters of the classifier part of the model.

This paper is organized as follows. In the next section, we motivate our work further by giving two concrete use cases. Section 3 presents the architecture, including our threat model and selected implementation details for TensorFlow and Intel SGX. Evaluation

AISec, November 13 2020, Orlando, USA
© 2020 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-8094-2/20/11…$15.00
https://doi.org/10.1145/3411508.3421376

[1]The release version of the framework can be found on GitHub under https://github.com/alxshine/eNNclave [36]. The experiments can be found under https://github.com/alxshine/eNNclave-experiments [37].

results for the inference time as well as the accuracy cost of confining sensitive parameters follow in Section 4. Section 5 positions the contribution in the context of related work. Section 6 discusses before Section 7 concludes.

2 USE CASES

Executing parts of a DNN in a trusted enclave has many potential applications. To motivate our approach and guide its evaluation, we introduce two exemplary use cases for supervised learning. The first one protects the privacy of subjects in a training dataset against model inversion [11] and set membership attacks [38]. The second use case protects test data and sensitive model parameter in an outsourcing scenario.
Independent of the use case, the term *server* refers to the party who

- has access to labeled training data $\{(x, y)\}^n$,
- trains the model h by minimizing a loss over all training samples $\sum_{i=1}^{n} |h_\theta(x_i) - y_i|$, and
- "owns" the induced parameters θ.

By contrast, the *client*

- observes or collects the test data x',
- applies the model for inference $\hat{y}' = h_\theta(x')$, and
- needs to know the classification output $\hat{y}'$.

Both use cases instantiate client and server in different ways.

2.1 Protecting training data during offline inference

A *company* (= server) wants to authenticate employees with an offline face recognition system [1, 2] installed on all its *laptops* (= clients). While several pre-trained DNNs for face images exist, every such system needs to be retrained to the faces of all employees. This data is exposed to a privacy risk, e. g., if laptops get stolen and an attacker extracts information about employees' faces contained in the local copy of the model parameters θ.

Prior research has shown that an attacker with white-box access to the model h_θ can craft images that circumvent the authentication system [4], or extract (noisy but recognizable versions of) training images [11]. Using the membership inference attack [38], it would also be possible to detect whether a given person is an employee or not. The referenced source has shown that limiting $\hat{y}'$ to only the output label reduces the success rate of this attack to 66 % accuracy down from the original 92 % for a binary classification problem. No defense evaluated in [38] performed better for a fixed model.

The first step in defending all of the above attacks is hiding θ from the attacker. She is only able to query the model as a black-box. Black-boxes are currently created by keeping the model on the server and providing an online oracle for the attacker to query.

To move this black-box offline, the server needs a way to allow the client to efficiently infer $\hat{y}'$ without sharing θ in its entirety. Placing the dense layers and the associated parts of θ inside an enclave means that in the event of a compromised client, the attacker cannot analyze the layers leading to the output label.

2.2 Protecting test data while maintaining model confidentiality

Spam detection relies on ML models [5, 7, 51]. Specialized firms offer ML-based spam and phishing detection for email as a service [8, 21]. The *provider* (= server) treats the classifier h_θ as intellectual property and is not willing to share it with *customers* (= clients). He may even be bound by non-disclosure agreements with his suppliers of training data, and face legal liability if information about the training data leaks through θ. Most deployed solutions perform the classification server-side. Customers, in turn, are reluctant to share all email with the provider to have them classified. This use case demonstrates the full conflict of protection goals between client and server.

With our approach, the server can share h_θ in encrypted form, and let the client make predictions within a trusted enclave. This does not elevate the threat model to more than a black-box scenario. Because the client can efficiently calculate $\hat{y}'$ offline, x' never leaves its domain. The approach eliminates the need for the customer to entrust the provider with sensitive email communication.

Both use cases replace an online service by an offline solution. Without further measures, this deprives the server of the possibility to meter ML queries. We note that business models based on metering are still possible by using known techniques for keeping state in enclaves, such as monotonic counters [20], online signing of commitments to input data, or ledgers [22]. This engineering task is tangential to the present work and not further considered here.

3 ARCHITECTURE

This section describes the threat model, high-level architecture, and selected details of our proof-of-concept implementation.

3.1 Threat model

To analyze the conflicting protection goals, we define two attackers:

Attacker 1. is a malicious client who wants to learn the secret part of θ faster than with repeated black-box access. More formally, we split θ into a confidential part θ_e and a public part $\theta_{\bar{e}}$. Creating and sustaining an offline black-box requires that θ_e and any intermediate values derived from it stay inside the trusted enclave at all times.[2] Attacker 1's objective is to break the confidentiality of this black-box. She will compromise availability or integrity only as a means to this end.

The attacker knows all information about the public part of the model and its parameters $\theta_{\bar{e}}$. She owns the system and controls the entire environment outside the trusted enclave, including input data and all persistent memory. But she cannot read or modify the enclave state other than through the specified entry points. Side channels (cache, timing, etc.) are out of scope for this paper, and we assume Attacker 1 cannot exploit them. Moreover, Attacker 1 is computationally bounded. She cannot brute-force keys or impersonate the server or the vendor of the trusted processor if a public key infrastructure (PKI) is used.

[2]Recall that with this information attackers can craft adversarial samples [4], perform model inversion [11, 38, 40], or steal the model [45].

We recall the limitations of trusted enclaves as computers with limited and volatile memory. This means that any persistent state needs to be stored outside the trusted enclave, and can be inspected or modified if it is not encrypted and integrity protected. Without additional precautions, the trusted enclave is prone to state rollback attacks.

Attacker 2. is a malicious server who wants to abuse test data x'. In our design, all communication between client and server happens before the test samples are observed (see Fig. 2 below). As the client makes inferences offline, the server never receives x'. The client controls the entire environment of the trusted enclave, making it impossible to leak x' using a malicious enclave. Therefore, Attacker 2 is warded off by the design of eNNclave, but mentioned here for completeness.

The system designer's objective is to keep the inference as efficient and accurate as possible, with respect to an insecure alternative that requires trust between client and server, while defending against both attackers.

3.2 High-level pipeline

Fig. 1 illustrates how we split a DNN into a public and a private part. The feature extractor (which here consists of 5 convolutional blocks) remains public, while the task-specific layers are enclaved.

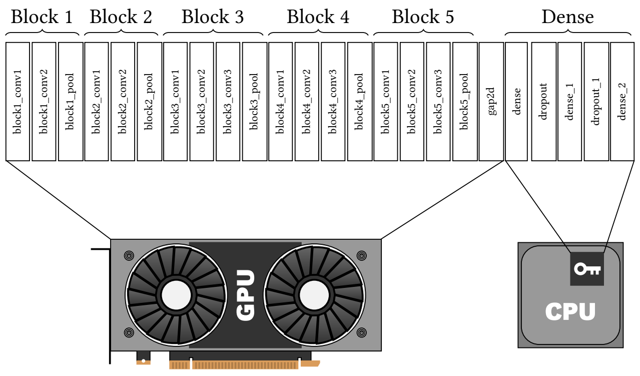

Figure 1: Split of a VGG-16 model into public and enclaved parts. The convolutional layers (Block 1–5) are taken from a pre-trained model with public parameters.

The input to our toolchain is a TensorFlow model given in HDF5 format [15], created for instance by the standard save function of the popular TensorFlow Keras library. These files contain the model architecture with labeled layers, and all parameters. We have implemented a tool to load the original model, separate the last k layers (k supplied as command line argument), and generate enclave code in C according to the architecture information for those layers. The resulting enclave code is then compiled as an Intel SGX enclave. At this step, the parameters of the enclaved layers are extracted for the secret provisioning described below. In our proof-of-concept, we directly encrypt θ_e to disk instead of going through the entire secret provisioning process.

We reimplemented all TensorFlow operations used in our test networks as functions in C. The input model is parsed, and a master `forward` function containing the required calls for each layer is generated. Code for loading the encrypted parameters from disk is also inserted into the `forward` function. Appendix B contains annotated example code resulting from this process.

Users can interact with the generated enclave using a Python-to-C adapter we built. It provides Python bindings, but is itself written in C using the standard Python/C API [10]. Enclave instantiation, teardown, and calling the `forward` function can be done through its interface. This adapter also handles all internal state required for enclave interaction.

Additionally, we built a custom TensorFlow layer to abstract the enclave interaction during inference. This layer has no parameters as those are stored securely in the enclave. Our tool generates a new HDF5 file with the information for the public layers and replacing the last k layers with the `EnclaveLayer`. This file is ready to be sent to the client along with the enclave.

At runtime, the public part of the network is executed in regular TensorFlow until it reaches the `EnclaveLayer`. This layer then extracts the data from the TensorFlow tensors and passes it to the Python-to-C adapter, which forwards it to a previously instantiated enclave. After the enclave has completed the forward pass, $\hat{y}'$ is returned to the `EnclaveLayer`, which wraps it in a TensorFlow tensor and returns it to the calling Python code again.

3.3 Implementation details regarding efficiency

Intel SGX reserves 128 MB of RAM for the entire trusted execution environment, not all of which is available to the programmer. Our tests indicate a maximum of 120 MB of physical memory can be allocated. The available memory is also shared between all enclaves. Ignoring code size, the (roughly) 120 MB of available memory allow for 31 million 32-bit floats, shared between θ_e and intermediate results. Comparing that to the models we used, the VGG-19 model has 41 million parameters, and current models grow larger still.

To reduce the memory footprint of θ_e, we gradually load and decrypt parameters from disk as they are required for the forward pass. As the number of parameters for each layer is known during enclave code generation, we hard-code load calls into the enclave. A single heap buffer is reused for the parameters of all layers. In total, this allows us to reduce the memory footprint for θ_e to the parameter size of the largest enclaved layer.

Due to data dependencies in matrix multiplication and convolution, we cannot use a single buffer for intermediate results. We alternate between two buffers, whose size is determined by the two largest output shapes of consecutive layers. Both optimizations combined reduce our memory footprint to the parameters of the largest layer, plus twice the size of the largest intermediate output.

It remains to handle the result of the public part of the DNN. Per Intel SGX specification, arrays passed to the enclave in function calls are copied to enclave memory on entry. This increases the memory footprint by the input size of the first enclaved layer.

While SGX on Linux can increase memory through paging [18], virtual memory is not automatically increased as needed and new pages need to be explicitly requested. Paging also causes significant performance impacts, requiring roughly 40 k cycles per 4 kB page [41]. Naive usage of paging can also introduce new side-channels into the code, thus increasing the attack surface for θ.

Due to the negative security and performance impacts, we decided against the support of paging and focused on reducing the memory footprint of our enclave instead.

Our proof-of-concept using Intel SGX is limited to feed-forward DNNs. These DNNs have no state, and the only data structures we need to handle are parameter and input matrices. TensorFlow stores these matrices in memory as flattened arrays of 32-bit floats, so they can be forwarded to the enclave without transformation. We did not optimize the dense matrix multiplication and convolution for cache coalescing. While the performance loss for matrix multiplications seems negligible, the convolutions are visibly slower than in optimized TensorFlow CPU code, presumably due to higher cache pressure.

Enclave instantiation takes between 0.74–0.80 seconds, seemingly independent of how much computation is performed inside the enclave. This time is determined by the Intel framework. We have not explored ways to reduce it.

3.4 Implementation details regarding security

Fig. 2 shows the setup and runtime procedure of eNNclave. It involves four parties: client C, server S, the client's trusted processor P, and the vendor of the processor V. We use the common trust assumptions: S trusts P, and both S and C trust V.

The following steps serve the goal of ensuring the integrity of $e(h)$. V has equipped P with a unique private key P_{priv} and knows the corresponding public key P_{pub}. S has a way to obtain the authentic P_{pub} for P. S generates enclave code $e(h)$ from h and transmits it to the client C. This $e(h)$ is not encrypted as it does not carry confidential information, and C can extract the model to run its public part outside the enclave. C instantiates $e(h)$ in P to prepare the enclaved part. P derives a key pair e_{pub} and e_{priv} specific to $e(h)$ and itself. P then sends a signature of $e(h)$ and e_{pub} to C who forwards it to S. S verifies the authenticity using P_{pub} published by V, as well as the integrity of $e(h)$. Intel SGX standardizes this procedure as *remote attestation* using a pre-installed *quoting enclave*.

The next phase handles the secret provisioning of the parameters θ_e. With the authentic e_{pub}, S can encrypt θ_e and send it to C. C forwards θ_e to the enclave residing in P during inference. Intel SGX has standardized this process as *secret provisioning*. Our proof-of-concepts uses Intel's sealing functionality to encrypt θ_e with AES/GCM. The key is derived from P and $e(h)$. $\theta_{\bar{e}}$ is not confidential and sent to C in plaintext.

After observing or receiving a test sample x', C calculates the public part of the inference process using h and $\theta_{\bar{e}}$. The resulting intermediate value $x'_{\bar{e}}$ is forwarded to the enclave running in P, where the confidential part of the inference is calculated using a decrypted θ_e. The prediction result $\hat{y}'$ is returned to C.

We acknowledge the risk of side channel attacks found against Intel SGX, which may compromise the common trust assumption placed in P, and discuss them in Section 5.

4 EVALUATION

We set up a series of experiments, detailed in Table 1, to evaluate the impact our approach on performance in terms of inference time and accuracy. Fitting our use cases, we selected tasks from the domains of image recognition and text analysis. For multiple

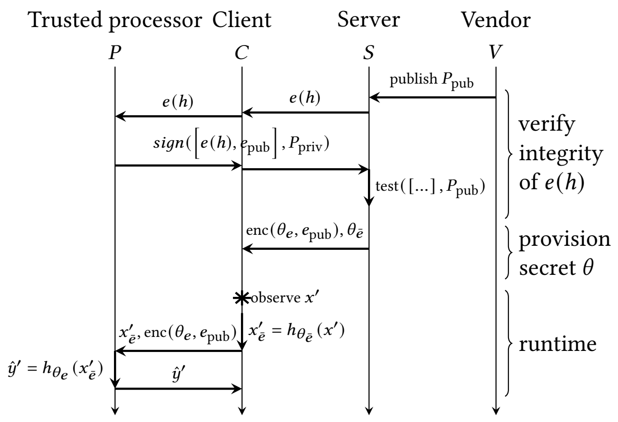
Trusted processor
Client
Server
Vendor
P
C
S
V
publish P_{pub}
$e(h)$
$e(h)$
$sign(\left[e(h), e_{\text{pub}}\right], P_{\text{priv}})$
test$([\ldots], P_{\text{pub}})$
verify integrity of $e(h)$
enc$(\theta_e, e_{\text{pub}}), \theta_{\bar{e}}$
provision secret θ
observe x'
$x'_{\bar{e}}$, enc$(\theta_e, e_{\text{pub}})$
$x'_{\bar{e}} = h_{\theta_{\bar{e}}}(x')$
$\hat{y}' = h_{\theta_e}(x'_{\bar{e}})$
$\hat{y}'$
runtime

Figure 2: Setup and runtime steps required to calculate $h_\theta(x')$ with confidential θ_e. S must verify that an unmodified $e(h)$ is running in P before it can send θ_e without risking loss of confidentiality. The first two phase only happen once. No connection between C and S is required thereafter.

DNN architectures, we measure the inference time as a function of the split. For one plausible split, we compare the accuracy in a transfer learning scenario with free and fixed parameters $\theta_{\bar{e}}$ of the non-enclaved part. As the focus of this work is security and not ML, we refrained from tuning our hyper-parameters to match published performance benchmarks for the tasks and domains. Nevertheless, all evaluated models produce sufficiently high baseline accuracy to rule out spurious performance due to trivial classifier realizations.

4.1 Evaluation setup

4.1.1 Tasks and data. For the image recognition task, we use the indoor scene classification[3] dataset MIT67 [34] and the VGG-16 convolutional architecture (configuration D in [39]). The DNN was pretrained on the training set of the ImageNet ILSVRC challenge [35]. In our accuracy evaluation, we also use a second set of pre-trained parameters specifically for indoor scene classification based on the Places365-Standard database [53]. The input data is resized to the VGG-16 input size of $224 \times 224 \times 3$ RGB images. After the resizing, the feature space of both source sets and the target set are identical. The ILSVRC dataset contains images from multiple contexts, and is not specific to indoor scene recognition. While the Places database is focused on scene recognition like our target task, it also contains outdoor scenes. Our example scenario is thus a domain adaption scenario. As we have labeled data in both the source and the target domain, this corresponds to *inductive transfer learning* as referred to by Pan et al. [30].

Text analysis is evaluated using a synthetic transfer learning problem using Amazon review data [47]. We pre-train a 5-class classifier with the review text and the associated ratings of 150 000 book reviews. The ratings on a scale of 1–5 stars serve as labels. The custom task in this scenario is an unbiased subset of 200 CD and vinyl record reviews per rating, to which we are transferring

[3]Unfortunately, our efforts to use a face recognition task, inspired by the use case, failed due to insurmountable difficulties in compiling the dataset needed to replicate the results published in [33].

the model. A vocabulary of the 20 000 most common words is built from the source dataset. Words are replaced by their indices in this vocabulary. Words not in the vocabulary are replaced with 0. The vocabularized review texts are padded or shortened to 500 tokens to build the training set. Both the source and target datasets were vectorized using the embedding learnt on the source dataset. We again have slightly different source and target tasks, with labelled data for both, making this another domain adaption task in inductive transfer learning.

More precise information about the source and target datasets is contained in Table 2.

4.1.2 Models. As stated earlier, we used the VGG-16 architecture for the indoor scene recognition task. To further investigate the performance limits of our current implementation we also used a dataset of flower images [26] and a VGG-19 model (configuration E in [39]). This was purely for performance analysis and no transfer learning was done. For the larger model, we reduced the number of parameters in the largest dense layer because it requires more memory than is available in the standard configuration.[4] We used the VGG models because they provide very good accuracy on large and complex datasets by connecting only convolutional, pooling, and dense layers in a simple feed-forward manner.

For the text analysis feature extractor tasks, we utilized a simple CNN consisting of an embedding layer, followed by 4 blocks of one separable 1D convolution and a 1D max-pooling each. The task classification was done by 3 dense layers interleaved with dropout layers.[5]

Table 1: Overview of evaluation models and datasets.

Task	Samples				Extractor		Classif.		
Dataset	train	test	c	bias	l	p	l	p	feat.
Image recognition (use case 1)									
MIT67	5360	1340	67	1.0	19	14 M	5	5 M	150 k
Flowers	3485	865	5	1.0	22	20 M	4	21 M	150 k
Text analysis on Amazon reviews (surrogate for use case 2)									
Books	600 k	150 k	5	1.0	9	752 k	6	1 M	500
CDs	800	200	5	1.2[a]	9	752 k	6	1 M	500

a: Bias caused by random split into training and test set.

Table 1 gives an overview of our evaluation tasks, where c refers to the number of classes, l refers to the number of layers, and p refers to the number of parameters. We report a bias metric calculated by the fraction of the most frequent class in the test dataset divided by the expected number of occurrences under equal priors. A value of 1 means no bias. This check rules out that accuracy metrics are inflated by assignments to a dominant class.

4.2 Performance

To evaluate the inference performance of eNNclave, we measured the required time for predicting a single sample. This allows us to compare the instantiation time required for the trusted enclave execution and TensorFlow, respectively. We break down the time required for inferring on h_{θ_e} and $h_{\theta_{\bar{e}}}$.

Our experiments were run on an Intel i7-9700 3 GHz 8-core CPU with 64 GB RAM running Ubuntu Linux 18.04 with the Linux SGX SDK [18] and drivers [19] installed. While we do have an Nvidia RTX 2070 GPU installed on the machine, we did not use it for our evaluation as the inference times for a single sample were consistently faster on the CPU. Table 4 (in the appendix) shows a detailed comparison of CPU and GPU execution times. Transfer learning tasks were evaluated with fixed feature extractor weights and measured on the target dataset.

All reported measurements are averages of the runtime of 20 independent inferences on the same input. The standard deviation for the enclave execution time was below 3 % for all models and all splits, indicating that the number of measurements is commensurate to the measurement error. We confirmed that later iterations are not faster than early ones, thereby ruling out that pre-loaded caches affect our measurements. We validated correctness by checking that all splits for all models return the same result as the pure TensorFlow inference.

Our results are shown in Figs. 3 to 5 and Table 5 in the appendix. The bar charts show the total time per inference on a logarithmic vertical axis for every possible split of the DNN. As layers are enclaved from the last to the first, the leftmost bar (at tick 0) represents the baseline without eNNclave, and the rightmost bar shows the (not recommended) case of a fully enclaved network. Ticks in between denominate the number of layers *from the back* that are executed as a trusted enclave.

As to the breakdowns, *TensorFlow time* is the time required to compute $h_{\theta_{\bar{e}}}(x')$. *Enclave execution* is the time required to compute $h_{\theta_e}(x'_{\bar{e}})$. The total enclave time is the sum of the enclave execution and the *enclave instantiation* time, which can make up a large part of the total runtime for small h_{θ_e}. All enclave times refer to the section marked *runtime* in Fig. 2. We do not measure the enclave generation and secret provisioning because it happens only once.

As visible in Fig. 3, enclaving the dense layers (ticks 1–5) has a smaller performance impact than enclaving dense as well as convolutional layers. The computation time is either matched or surmounted by the time required to set up the enclave, depending on the number of enclaved layers. The layer at tick 3 has the largest number of parameters making up 80 % of the dense classifier, which causes a visible increase in the required computation time. The layers at ticks 2 and 4 are dropout layers. These layers are only active during training, so we do not generate code for them and it is evident that the bars do not change. The layers at ticks 6 and 7 are pooling layers, which are fast to compute. The convolutional layers have a much larger impact on performance. This is due to the larger input sizes and a higher number of computations required for convolutions in general. The difference between convolutions in TensorFlow and convolutions in the enclave can be attributed to the hardware limitations of the trusted enclave, as well as less optimized code. As shown in the figure, when convolutional layers are enclaved, the instantiation time becomes less dominant (but keep in mind the log scale).

At the suggested cutoff with 5 enclaved layers (dashed line), the performance impact of inference with parameter confidentiality

[4]We reduced the number of parameters of the dense layers from 4096 to 800. Only the reduction of the first layer is required to make the model small enough for eNNclave.
[5]See the code for more details. It is included in the experimental git repository [37].

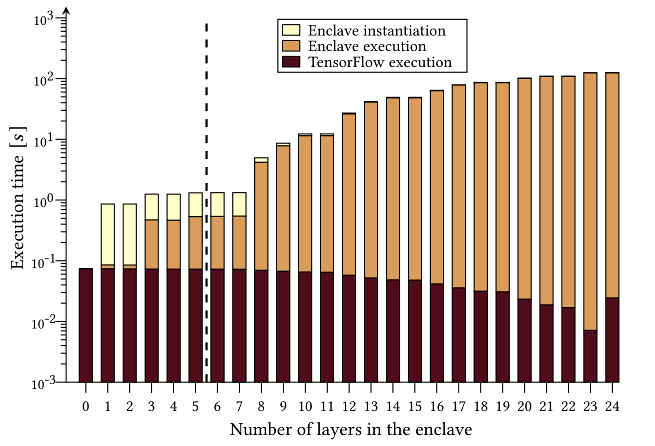

Figure 3: Performance evaluation for our MIT67 indoor scene classification model based on VGG-16 as a function of eNNclave's split point. The leftmost bar (tick 0) is the baseline *without* eNNclave. Tick 24 is fully enclaved. The dashed line shows the boundary between the task-specific classifier and the general feature extractor, the recommended split. Lower is better (faster). Mind the logarithmic scale.

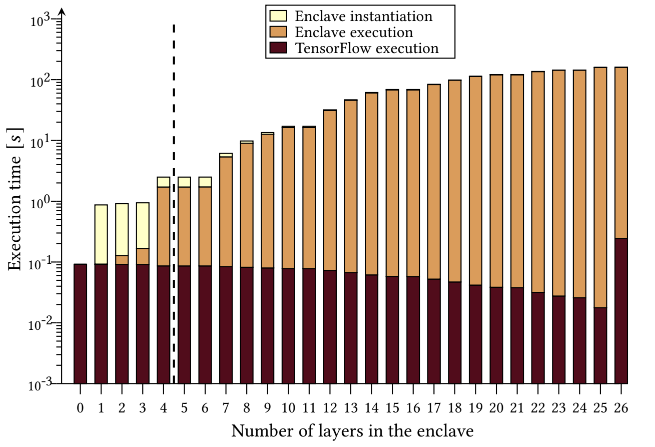

Figure 4: Performance evaluation of our flower classification model based on VGG-19 as a function of the split point.

is factor 17. However, with an inference time of 1.3 seconds, this may be affordable for occasional inferences on single images, for instance in the use case of employee face recognition on laptops. If one can amortize the instantiation time over multiple inferences, the relative overhead decreases.

The VGG-19 model on the flowers dataset in Fig. 4 confirms the observation from the smaller model. The dense layer at tick 4 has the highest number of parameters and causes the largest performance loss. Executing the other dense layers inside the trusted enclave requires less time than setting it up. Note that this model does not use any dropout layers. All layers are active during inference.

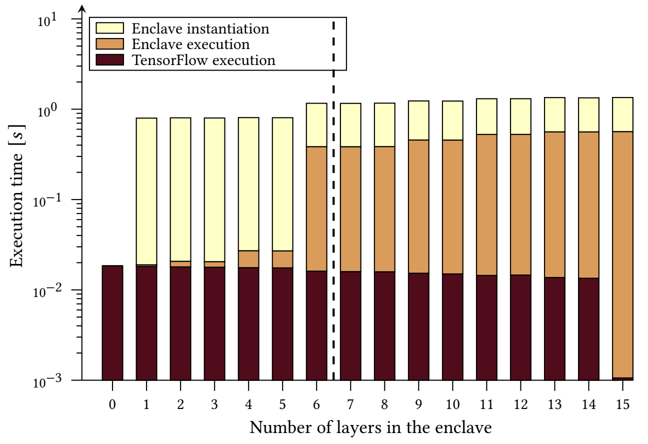

Figure 5: Performance evaluation of our text analysis task using a convolutional DNN as a function of the split point.

As the VGG-19 DNN is deeper than VGG-16, executing the entire model inside the trusted enclave takes more time on inputs of the same size.

Our text analysis model in Fig. 5 exhibits the largest increase in execution time for the first dense layer (tick 6). This layer has the largest input size and holds almost all parameters of the task-specific classifier. For all other dense layers, the instantiation takes an order of magnitude longer than execution.

Observe that the 1D convolutions in text analysis have a much smaller performance impact than the 2D convolutions in image recognition. This is due to smaller inputs and the need to iterate over one dimension less, which greatly increases the efficiency of memory access. Note that the tokenization of input texts is not part of the network and thus not included in the measurements. We also did not port `tensorflow.embedding` layers to the enclave. It is always run in TensorFlow, but requires very little time due to its relatively small input size.

At the suggested cutoff with 6 enclaved layers (dashed line), the price to pay for parameter confidentiality is factor 65. While this sounds prohibitive at the first glance, the corporate email filter in our use case will probably make many inferences at a time and hence amortize the enclave instantiation penalty more easily, which reduces the overall impact to factor 23. For perspective, confidentiality of corporate emails (and safety from spam and malware) may be worth a 10-fold hardware investment and twice the latency.

To use the TensorFlow adapter, the input layer is never enclaved. This is visible at the rightmost ticks in Figs. 3 to 5. As the time required for executing $h_{\theta_{\tilde{e}}}$ also depends on input and output size, splits with a single public layer can ironically require more time than models with multiple public layers. However, this effect is negligible when compared to the total inference time.

Running neural networks fully inside trusted enclaves is suitable for small models, but breaks down performance as the input size and number of parameters increase. For models of the size used in image recognition, eNNclave's ability to cut between any two

Table 2: Accuracy of transfer learning with and without fixing the non-enclaved layers. The difference between flexible and fixed is the *cost of confining sensitive parameters to the dense layers* during retraining.

Domain	Original model					Retrained to custom task						
	Samples				**Accuracy**	Samples				**Accuracy**		
Dataset	train	test	C	Bias	(flexible)	train	test	C	Bias	flexible	fixed	diff.
Image recognition												
ImageNet	1.2 M	150 k	1000	n/a	74.4 % [a]	5360	1340	67	1.0	59.7 %	57.1 %	2.6 %
Places	1.8 M	329 k	365	1.012	55.2 %	5360	1340	67	1.0	62.9 %	60.1 %	2.8 %
Text analysis												
Books	480 k	120 k	5	1.004	52.5 %	800	200	5	1.2	48.5 %	47.0 %	1.5 %

a: Table 3 in [39]: top-1 validation error for configuration D (VGG-16)

layers allows the operator to choose the application-specific tradeoff between security and performance. This was not possible before.

4.3 Accuracy impact

Choosing eNNclave's cutoff layer in our transfer learning scenario entails a tradeoff between inference speed and accuracy. The measurements presented in the last section suggest that inference times remain practical when enclaving the dense layers only. In this section we evaluate the resulting accuracy loss if a classifier is retrained to a task with $\theta_{\bar{e}}$ fixed. For the common practice of using public feature extractors [30], this approach guarantees that no information about the retraining set is leaked to the adversary. Our baseline case are transfer learning models retrained on exactly the same data, but with all parameters flexible. Table 2 shows the results.

For the image recognition task using the VGG-16 architecture, we evaluate the case of enclaving five layers. We observe accuracy losses of 2.6 %-pts (from an initial 59.7 % task accuracy) and 2.8 %-pts (from 62.9 %) for a pre-trained network we learned from the ImageNet ILSVRC 2014 challenge [35] and public weights specifically trained for indoor scene recognition [52], respectively.[6]

For comparison, we find an accuracy loss of 1.5 %-pts, down from an in initial 48.5 % task-specific classification accuracy, for our synthetic text analysis task using the vanilla convolutional DNN architecture. Here we enclaved the last six layers, noting that the scenario is a bit artificial because the model is small enough to fully fit into the enclave with negligible performance impact. Nevertheless, it confirms the finding of relatively small performance losses for another domain. It also corroborates the tenor in the transfer learning literature on the relevance of the dense layers for task-specific classification accuracy [39, 52].

Overall, it depends on the criticality of the application to decide if accuracy losses of this magnitude are acceptable. In principle, one can extend the speed–accuracy tradeoff to a triangle relationship. This would involve accepting some confidentiality loss by retraining with more flexible layers than enclaved during inference. However, since quantifying the information leakage from non-enclaved convolutional layers is non-trivial, we defer the exploration of this dimension to future work. This could encompass a security-aware composition of training sets.

[6]The initial training iterated over 2000 epochs. Retraining took 200 epochs with 3 batches per epoch. Further meta-parameters and test/training split seeds can be found in our experiment repository [37].

5 RELATED WORK

The eNNclave approach contributes to and builds on several streams of literature, which we summarize one by one.

Trusted enclaves for machine learning. Prior art using trusted enclaves for machine learning include Ohrimenko et al. [29] and Kunkel et al. [24]. These works enclave the entire model and thus are limited to models fitting into the hardware or require expensive and possibly insecure paging. None of the two source reports results for models of input size larger than 32×32 pixels. Our input sizes are 50 times as large. Tramèr et al. [44] propose using the trusted enclave as controller for computations performed on an untrusted GPU. Inputs and intermediate results are encrypted using an additive cipher, and the results are verified statistically using Freivalds' algorithm [12]. This method scales to large models, but it cannot be used to protect the model parameters in an offline scenario.

Hanzlik et al. [16] propose enclaving individual model layers. This approach circumvents the size limitation of enclaves, but incurs constant performance overhead during the transition between trusted and untrusted execution, as the input and output data of each layer must be transferred between environments. More importantly, the approach is vulnerable to differential attacks on individual layers, which give rise to divide-and-conquer algorithms for model stealing. We are not aware of a security analysis showing the limits of such attacks. Our work avoids this risk as data does not leave the trusted environment between layers.

Fig. 6 illustrates the different ways eNNclave and prior art use the trusted procession in ML.

Cryptography and distributed computing. Cryptographic defensive measures have been proposed to protect test data in online scenarios. Graepel et al. [14] use homomorphic encryption to train and apply polynomial classifiers on small amounts of encrypted data. CryptoNets by Gilad-Bachrach et al. [13] apply this method to neural networks. However, it requires changes to the model and greatly increases inference time. Known methods using homomorphic encryption for neural networks do not protect the model

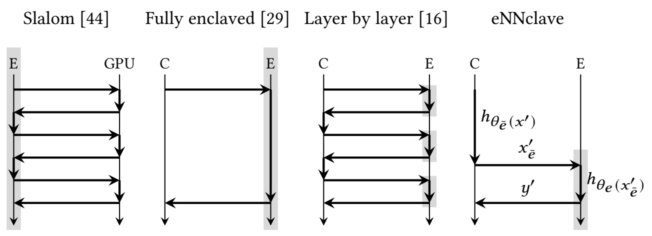

Figure 6: Use of trusted processors for inference on neural networks: eNNclave and prior art.

parameters, cannot handle large models, and do not support all types of activation functions.

Multiparty computation (MPC) has been used by Mohassel and Zhang [27] to distribute data and models for regressions and neural networks. A similar approach could in principle be used to split data and model between client and server, achieving test data and model confidentiality. However, multiparty computation requires online communication, substantial bandwidth, latency overhead, and distributed trust assumptions.

Focusing on other aspects than security, Dean et al. [9] and Teerapittayanon et al. [43] propose distributing neural networks over multiple nodes. In principle, this could prevent a single malicious node from learning x' or θ completely. But it incurs the cost of online communication. As neither of the two methods has been designed with security in mind, the model confidentiality and confidentiality of test data have not been evaluated.

Most prior work we are aware of in the security domain deals with protecting client data on the server. While some methods could be adapted to protect the server's model on the client's machine, they were not designed for such a scenario and cannot be used truly offline. Table 3 synthesizes a comparison of related work to the proposed eNNclave proof-of-concept.

Table 3: Comparison of approaches to solve confidentiality problems in machine learning.

	Confidentiality		Offline	Hardware	**Versatility**	
Approach	data	parameters	inference	accel.	size	activ.
Slalom [44]	●	○	○	●	●	●
Fully enclaved [29]	●	●	●	○	○	●
Layer by layer [16]	●	◐	●	○	●	●
Homom. crypto [13, 14]	●	●	○	○	○	○
MPC [27]	●	●	○	○	○	○
Split models [9, 43]	◐	◐	○	●	●	●
eNNclave *(proposed here)*	●	●	●	◐	●	●

Attacks against machine learning at inference time. The attacks reviewed here contribute to the motivation of our work and its threat model. Tramèr et al. [45] first showed how to extract model parameters from black-box oracles using targeted queries. When the model is public, information about the training process can be extracted, as shown by Wang et al. [50]. Fredrikson et al. [11] leverage model characteristics to extract samples from the training set using prediction confidences in white-box scenarios. Shokri et al. [38] and Song et al. [40] propose membership inference attacks using surrogate models and prediction confidences. Surrogate models were also used by Papernot et al. [31] to craft adversarial samples for the target model. Most of these attacks take advantage of white-box access, and mitigations require a black-box setting.

Security of trusted enclaves. Our work inherits weaknesses of imperfect enclave implementations. Multiple works have shown that microarchitectural side channels [3] threaten the confidentiality and integrity of Intel SGX. The Foreshadow attack, presented by van Bulck et al. [48], is a variant of Meltdown [25] that works against trusted enclaves. Similarly, Chen et al. [6] adapted Spectre [23] to trusted enclaves. Both attacks could compromise sealing or attestation keys, but have since then been mitigated with microcode updates following a responsible disclosure process.

Murdock et al. [28] use undocumented voltage regulation instructions in Intel processors to inject faults into enclave execution, which can theoretically enable differential fault analysis [46] of the AES keys used for sealing. The interface enabling this attack has since been disabled with BIOS updates and microcode updates reflect this status in SGX quotes. A similar attack for ARM Trustzone on Android devices has been proposed by Tang et al. [42].

The very recent load value injection attack [49] does not affect our enclave code specifically as it only has a single level of indirection and does not dereference any loaded value. Nevertheless, we plan to update the enclave code generation to avoid return instructions, which are weak points for several reasons.

Despite mitigation efforts on many layers, it seems that all current variants of trusted enclaves are vulnerable to side channels attacks. While these general issues of trusted enclaves prevail, our approach is practical for scenarios where attackers do not possess the capabilities to launch such sophisticated attacks.

6 DISCUSSION

This work proposes a new point in the design space for secure inference on larger neural networks in untrusted domains. We take advantage of a known phenomenon in transfer learning to achieve model and test data confidentiality. The use of trusted enclaves makes our work practical, as demonstrated with experimental evidence. Our implied trust assumptions are shared by all existing solutions using trusted computing. The risk of imperfect isolation may subside as the technology matures and better trusted processors proliferate.

Several extensions of our proof-of-concept stand to reason. The current implementation supports feed-forward DNNs only. The inclusion of skip and recursion layers requires a more thorough study of information leakages if such connections cross the boundary of the enclaved part of the network. More complex connections between layers, like in the ResNet architectures [17], complicate the memory management for intermediate results inside the enclave. Performance can be improved by using cache-adjusted data layouts and SIMD instructions.

Concerning the high-level pipeline, the model architecture is currently not confidential as the code is not encrypted. Using Intel's protected code loader would enable this, however at the cost of increased vulnerability to attacks like LVI [49]. Conversely, the ability to inspect the architecture of the black-box oracle can also be seen as a feature which helps hold the server accountable.

For our current evaluation, we use a simple transfer learning approach with a complete split between feature extractor and task-specific classifier. This guarantees us that the feature extractor does not leak any information about the training set, and enclaving the classifier fully protects the task-specific θ. At the cost of increased confidentiality risk, more granular transfer learning approaches can be taken to improve the classification accuracy of the full model. As the training happens before the model is passed to our pipeline, any conceivable training approach is possible. Similarly, the model can be split at any desired location. Both retraining and split location influences the amount of information leakage, for which we have no precise way of measurement at the moment. A reevaluation of current black-box attacks [31, 45] with partially known models is a possible avenue for future work.

Looking ahead, it might be worth exploring if there are useful applications for an adapted eNNclave approach that hides the parameters of the central layers of autoencoders. This breaks our current guarantees gained from using pre-trained feature extractors and requires more study of the information leakage properties. Similar to using it for autoencoders, eNNclave could enable placing black-boxes on untrustworthy nodes in heterogeneous architectures for edge, fog, and cloud computing [9, 43].

7 CONCLUSION

We have presented eNNclave, a new way to solve the confidentiality conflict for outsourced machine learning. This work is novel in its goal to allow fast offline inference with confidential parameters, while not drastically increasing the attack surface for model stealing attacks. Client-side offline inference keeps test data confidential, and the use of trusted enclaves protects the model parameters. Our toolchain allows the user to cut a given TensorFlow model between any two layers and generates enclave code for the last layers. This flexibility enables new tradeoffs between inference speed, model confidentiality, and accuracy. For example, our proof-of-concept implementation makes inferences on a 24-layer deep neural network with 150 k input dimensions in just over a second by executing 5 dense layers in the enclave. We demonstrate that sensitive information about the training set can be contained in these enclaved layers at an accuracy loss as small as 3 percentage points on an unbiased 67-class image recognition task.

While our measurements refer to Intel SGX, the generated enclave code is not specific to any platform. This somewhat mitigates concerns of vulnerabilities in current SGX-enabled processors. It seems possible to adapt eNNclave to any future trusted execution environment, hopefully ones that deserve this name.

Acknowledgments

We thank Michael Fröwis and Daniel Woods for feedback on earlier drafts, and the participants of the 2020 Obergurgl Winter School for useful discussion. This work was funded by Archimedes Privatstiftung, Innsbruck, Austria.

REFERENCES

[1] Amazon. 2016. Amazon reknognition. https://aws.amazon.com/rekognition/. Accessed on 2020-04-15.

[2] Kairos AR. 2012. Kairos face recognition. https://kairos.com/. Accessed on 2020-04-15.

[3] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. 2019. A systematic evaluation of transient execution attacks and defenses. In *USENIX Security Symposium*. USENIX, 249–266.

[4] Nicholas Carlini and Daniel Wagner. 2017. Towards evaluating the robustness of neural networks. In *IEEE Symposium on Security and Privacy (S&P)*. 39–57.

[5] Chao Chen, Jun Zhang, Yi Xie, Yang Xiang, Wanlei Zhou, Mohammad Mehedi Hassan, Abdulhameed AlElaiwi, and Majed Alrubaian. 2015. A performance evaluation of machine learning-based streaming spam tweets detection. *IEEE Transactions on Computational Social Systems* 2, 3 (2015), 65–76.

[6] Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten H. Lai. 2019. SgxPectre Attacks: Stealing Intel secrets from SGX enclaves via speculative execution. In *IEEE European Symposium on Security and Privacy (EuroS&P)*. 142–157.

[7] Michael Crawford, Taghi M. Khoshgoftaar, Joseph D. Prusa, Aaron N. Richter, and Hamzah Al Najada. 2015. Survey of review spam detection using machine learning techniques. *Journal of Big Data* 2 (2015), 23. Issue 1.

[8] Darktrace. 2013. Darktrace email security. https://www.darktrace.com/en/technology/#email-security. Accessed on 2020-04-15.

[9] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Marc'aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, Quoc V. Le, and Andrew Y. Ng. 2012. Large scale distributed deep networks. In *Advances in Neural Information Processing Systems*. Vol. 25. Curran Associates, 1223–1231.

[10] Python Software Foundation. 2020. Python C API. https://docs.python.org/3.7/c-api/index.html. Accessed on 2020-09-10.

[11] Matt Fredrikson, Somesh Jha, and Thomas Ristenpart. 2015. Model inversion attacks that exploit confidence information and basic countermeasures. In *ACM Conference on Computer and Communications Security (CCS)*. ACM, 1322–1333.

[12] Rusins Freivalds. 1977. Probabilistic machines can use less running time. In *IFIP Congress*, Bruce Gilchrist (Ed.). 839–842.

[13] 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 *International Conference on Machine Learning (ICML)*, Vol. 48. 201–210.

[14] Thore Graepel, Kristin Lauter, and Michael Naehrig. 2013. ML Confidential: Machine learning on encrypted data. In *Information Security and Cryptology (ICISC)*, Han Dong-Guk Lee Hyang-Sook (Ed.). Springer, 1–21.

[15] HDF Group. 1998. HDF5 file format. https://portal.hdfgroup.org/display/HDF5/HDF5. Accessed on 2020-04-23.

[16] Lucjan Hanzlik, Yang Zhang, Kathrin Grosse, Ahmed Salem, Max Augustin, Michael Backes, and Mario Fritz. 2019. MLCapsule: Guarded offline deployment of machine learning as a service. https://publications.cispa.saarland/2751/.

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

[18] Intel. 2018. Intel software guard extensions (Intel SGX) SDK for Linux OS, Developer Reference. https://download.01.org/intel-sgx/linux-2.2/docs/Intel_SGX_Developer_Reference_Linux_2.2_Open_Source.pdf. Accessed on 2020-04-01.

[19] Intel. 2019. Intel SGX driver for Linux. https://github.com/intel/linux-sgx-driver. Accessed on 2020-04-22.

[20] Intel. 2019. Intel SGX monotonic counters. https://software.intel.com/en-us/dal-developer-guide-features-monotonic-counters. Accessed on 2020-04-15.

[21] Ironscales. 2014. Ironscales email security. https://ironscales.com/anti-phishing-solutions/. Accessed on 2020-04-22.

[22] Gabriel Kaptchuk, Ian Miers, and Matthew Green. 2019. Giving state to the stateless: Augmenting trustworthy computation with ledgers. In *Network and Distributed Systems Security (NDSS)*.

[23] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2019. Spectre Attacks: Exploiting speculative execution. In *IEEE Symposium on Security and Privacy (S&P)*. 1–19.

[24] Roland Kunkel, Do Le Quoc, Franz Gregor, Sergei Arnautov, Pramod Bhatotia, and Christof Fetzer. 2019. Tensorscone: A secure TensorFlow framework using Intel SGX. *arXiv preprint arXiv:1902.04413* (2019).

[25] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. 2018. Meltdown: Reading kernel memory from user space. In *USENIX Security Symposium*. USENIX, 973–990.

[26] Alexander Mamaev. 2018. Kaggle flower classification challenge. https://www.kaggle.com/alxmamaev/flowers-recognition/data. Accessed on 2020-04-21.

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

[28] Kit Murdock, David Oswald, Flavio D Garcia, Jo Van Bulck, Daniel Gruss, and Frank Piessens. 2020. Plundervolt: Software-based fault injection attacks against

Intel SGX. In *IEEE Symposium on Security and Privacy (S&P)*. 859–876.

[29] Olga Ohrimenko, Felix Schuster, Cédric Fournet, Aastha Mehta, Sebastian Nowozin, Kapil Vaswani, and Manuel Costa. 2016. Oblivious multi-party machine learning on trusted processors. In *USENIX Security Symposium*. USENIX, 619–636.

[30] Sinna Jialin Pan and Quian Yang. 2010. A survey on transfer learning. *IEEE Transactions on Knowledge and Data Engineering* 22, 10 (2010), 1345–1359.

[31] Nicolas Papernot, Patrick McDaniel, Ian Goodfellow, Somesh Jha, Berkay Celik, and Ananthram Swami. 2017. Practical black-box attacks against machine learning. In *Asia Conference on Computer and Communications Security (ASIA CCS)*. ACM, 506–519.

[32] Nicolas Papernot, Patrick McDaniel, Arunesh Sinha, and Michael P. Wellman. 2018. SoK: Security and privacy in machine learning. In *IEEE European Symposium on Security and Privacy (EuroS&P)*. 399–414.

[33] Omkar M. Parkhi, Andrea Vedaldi, and Andrew Zisserman. 2015. Deep face recognition. In *British Machine Vision Conference (BMVC)*, Xianghua Xie, Mark W. Jones, and Gary K. L. Tam (Eds.). BMVA, 41.1 – 41.12.

[34] Ariadna Quattoni and Antonio Torralba. 2009. Recognizing indoor scenes. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 413–420.

[35] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. 2015. ImageNet large scale visual recognition challenge. *International Journal of Computer Vision (IJCV)* 115, 3 (2015), 211–252.

[36] Alexander Schlögl. 2020. eNNclave framework. https://github.com/alxshine/eNNclave/tree/aisec. Accessed on 2020-09-10.

[37] Alexander Schlögl. 2020. eNNclave paper experiments. https://github.com/alxshine/eNNclave/tree/aisec. Accessed on 2020-09-10.

[38] Reza Shokri, Marco Stronati, Congzheng Song, and Vitaly Shmatikov. 2017. Membership inference attacks against machine learning models. In *IEEE Symposium on Security and Privacy (S&P)*. 3–18.

[39] Karen Simonyan and Andrew Zisserman. 2015. Very deep convolutional networks for large-scale image recognition. In *International Conference on Learning Representations (ICLR)*.

[40] Liwei Song, Reza Shokri, and Prateek Mittal. 2019. Membership inference attacks against adversarially robust deep learning models. In *IEEE Security and Privacy Workshops (SPW)*. 50–56.

[41] Meysam Taassori, Ali Shafiee, and Rajeev Balasubramonian. 2018. VAULT: Reducing paging overheads in SGX with efficient integrity verification structures. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, 665–678.

[42] Adrian Tang, Simha Sethumadhavan, and Salvatore Stolfo. 2017. CLKSCREW: Exposing the Perils of Security-Oblivious Energy Management. In *USENIX Security Symposium*. USENIX, 1057–1074.

[43] Surat Teerapittayanon, Bradley McDanel, and H.T. Kung. 2017. Distributed deep neural networks over the cloud, the edge and end devices. In *IEEE International Conference on Distributed Computing Systems (ICDCS)*. 328–339.

[44] Florian Tramèr and Dan Boneh. 2019. Slalom: Fast, verifiable and private execution of neural networks in trusted hardware.. In *International Conference on Learning Representations (ICLR)*.

[45] Florian Tramèr, Fan Zhang, Ari Juels, Michael Reiter, and Thomas Ristenpart. 2016. Stealing machine learning models via prediction APIs. In *USENIX Security Symposium*. USENIX, 601–618.

[46] Michael Tunstall and Debdeep Mukhopadhyay. 2009. Differential fault analysis of the advanced encryption standard using a single fault. In *IFIP international workshop on information security theory and practices*, Olivier Markowitch, Angelos Bilas, Jaap-Henk Hoepman, Chris J. Mitchell, and Jean-Jacques Quisquater (Eds.). Springer, 224–233.

[47] Justifying Recommendations using Distantly-Labeled Reviews and Fine-Grained Aspects. 2019. Jianmo Ni and Jiacheng Li and Julian McAuley. In *Empirical Methods in Natural Language Processing (EMLNP)*. ACL.

[48] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. 2018. Foreshadow: Extracting the keys to the intel SGX kingdom with transient out-of-order execution. In *USENIX Security Symposium*. USENIX, 991–1008.

[49] Jo Van Bulck, Daniel Moghimi, Michael Schwarz, Moritz Lipp, Marina Minkin, Daniel Genkin, Yarom Yuval, Berk Sunar, Daniel Gruss, and Frank Piessens. 2020. LVI: Hijacking transient execution through microarchitectural load value injection. In *IEEE Symposium on Security and Privacy (S&P)*.

[50] Binghui Wang and Neil Z. Gong. 2018. Stealing hyperparameters in machine learning. In *IEEE Symposium on Security and Privacy (S&P)*. 36–52.

[51] Tingmin Wu, Shigang Liu, Jun Zhang, and Yang Xiang. 2017. Twitter spam detection based on deep learning. In *Australasian Computer Science Week Multiconference (ACSW '17)*. ACM Press.

[52] Bolei Zhou, Agata Lapedriza, Aditya Khosla, Aude Oliva, and Antonio Torralba. 2017. Places: A 10 million image database for scene recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence* (2017).

[53] Bolei Zhou, Agata Lapedriza, Jianxiong Xiao, Antonio Torralba, and Aude Oliva. 2014. Learning deep features for scene recognition using Places database. In *Advances in Neural Information Processing Systems*. Vol. 27. Curran Associates, 487–495.

A ADDITIONAL TABLES

Table 4: Inference times (average of 20 independent inferences, in *s*)

Model	VGG-16	VGG-19	Text DNN
TensorFlow CPU	0.074	0.092	0.018
TensorFlow GPU	1.233	1.236	0.739

Table 5: Performance measurements (average of 20 independent inferences, in *s*)

	Image recognition											
	VGG-16				VGG-19				Text analysis			
#	TF	EE	EI	**Total**	TF	EE	EI	**Total**	TF	EE	EI	**Total**
0	0.074	–	–	**0.074**	0.092	–	–	**0.092**	0.018	–	–	**0.018**
1	0.074	0.011	0.774	**0.859**	0.091	0.001	0.777	**0.870**	0.018	0.001	0.778	**0.797**
2	0.074	0.012	0.774	**0.859**	0.091	0.036	0.782	**0.909**	0.018	0.003	0.782	**0.803**
3	0.073	0.399	0.778	**1.250**	0.091	0.076	0.774	**0.941**	0.018	0.003	0.779	**0.799**
4	0.073	0.393	0.782	**1.248**	0.086	1.627	0.781	**2.495**	0.018	0.010	0.780	**0.807**
5	0.073	0.460	0.780	**1.313**	0.086	1.628	0.780	**2.494**	0.018	0.010	0.778	**0.805**
6	0.073	0.466	0.783	**1.321**	0.086	1.633	0.777	**2.497**	0.016	0.369	0.777	**1.162**
7	0.072	0.472	0.778	**1.322**	0.084	5.280	0.778	**6.141**	0.016	0.369	0.776	**1.160**
8	0.070	4.117	0.774	**4.961**	0.082	8.931	0.779	**9.793**	0.016	0.370	0.781	**1.166**
9	0.067	7.771	0.777	**8.615**	0.080	12.586	0.780	**13.445**	0.015	0.441	0.778	**1.234**
10	0.065	11.418	0.779	**12.263**	0.078	16.230	0.782	**17.090**	0.015	0.440	0.776	**1.231**
11	0.065	11.429	0.777	**12.271**	0.078	16.237	0.782	**17.096**	0.014	0.511	0.780	**1.306**
12	0.058	26.194	0.777	**27.029**	0.073	30.993	0.778	**31.844**	0.015	0.512	0.780	**1.307**
13	0.052	40.938	0.778	**41.768**	0.067	45.758	0.778	**46.603**	0.014	0.548	0.783	**1.345**
14	0.049	48.310	0.777	**49.135**	0.061	60.512	0.777	**61.351**	0.013	0.547	0.776	**1.336**
15	0.048	48.316	0.783	**49.147**	0.058	67.883	0.778	**68.719**	0.001	0.564	0.781	**1.346**
16	0.042	63.261	0.776	**64.079**	0.058	67.886	0.778	**68.721**	–	–	–	**–**
17	0.036	78.229	0.773	**79.038**	0.052	82.844	0.782	**83.679**	–	–	–	**–**
18	0.032	85.695	0.777	**86.504**	0.047	97.809	0.778	**98.634**	–	–	–	**–**
19	0.031	85.737	0.780	**86.547**	0.042	112.754	0.779	**113.575**	–	–	–	**–**
20	0.023	100.852	0.779	**101.655**	0.038	120.228	0.780	**121.046**	–	–	–	**–**
21	0.019	108.433	0.781	**109.233**	0.038	120.247	0.780	**121.065**	–	–	–	**–**
22	0.017	108.485	0.779	**109.281**	0.032	135.380	0.779	**136.191**	–	–	–	**–**
23	0.007	123.807	0.780	**124.594**	0.028	142.960	0.779	**143.767**	–	–	–	**–**
24	0.024	124.527	0.780	**125.332**	0.026	142.991	0.782	**143.799**	–	–	–	**–**
25	–	–	–	**–**	0.018	158.332	0.778	**159.128**	–	–	–	**–**
26	–	–	–	**–**	0.244	159.156	0.778	**160.178**	–	–	–	**–**

TF = TensorFlow, EE = enclave execution, EI = enclave instantiation
Suggested cutoff layers are marked with horizontal rules.

B CODE EXAMPLE

Model definition in Python:

```python
# Public feature extractor
model = Sequential()
model.add(layers.Embedding(
  NUM_WORDS,
  32,
  input_length=SEQUENCE_LENGTH))
model.add(layers.SeparableConv1D(
  filters=64,
  kernel_size=3,
  padding='same',
  activation='relu'))
model.add(layers.MaxPooling1D(pool_size=2))
model.add(layers.SeparableConv1D(
  filters=128,
  kernel_size=3,
  padding='same',
  activation='relu'))
model.add(layers.MaxPooling1D(pool_size=2))
model.add(layers.SeparableConv1D(
  filters=256,
  kernel_size=3,
  padding='same',
  activation='relu'))
model.add(layers.MaxPooling1D(pool_size=2))
model.add(layers.SeparableConv1D(
  filters=256,
  kernel_size=3,
  padding='same',
  activation='relu'))
model.add(layers.MaxPooling1D(pool_size=2))
model.add(layers.Flatten())

# Sensitive classifier
model.add(layers.Dense(600, activation='relu'))
model.add(layers.Dropout(DROPOUT_RATE))
model.add(layers.Dense(150, activation='relu'))
model.add(layers.Dropout(DROPOUT_RATE))
model.add(layers.Dense(150, activation='relu'))
model.add(layers.Dense(1, activation='linear'))
```

Generated enclave code for classifier:

Load sealed parameters → (line 43)

Close parameter file → (line 83)

```c
int enclave_f(float *m, int s, float *ret, int rs) {
  int sts;

  open_parameters();
  float *tmp0 = (float*) malloc(600*sizeof(float));
  if(tmp0 == NULL){
    print_err("\n\nENCLAVE ERROR:Could not allocate tmp0 of
          size 600\n\n\n");
    return 1;
  }

  float *tmp1 = (float*) malloc(600*sizeof(float));
  if(tmp1 == NULL){
    print_err("\n\nENCLAVE ERROR:Could not allocate tmp1 of
          size 600\n\n\n");
    return 1;
  }

  float *params = (float*) malloc(4762200*sizeof(float));
  if(params == NULL){
    print_err("\n\nENCLAVE ERROR:Could not allocate params
         of size 4762200\n\n\n");
    return 1;
  }
  load_parameters(params, 4762200); // load and unseal
       weights and biases
  if ((sts = matutil_multiply(m, 1, 7936, params+0, 7936,
       600, tmp0))) // multiply with weights
    return sts;
  if ((sts = matutil_add(tmp0, 1, 600, params+4761600, 1,
       600, tmp0))) // add biases
    return sts;
  matutil_relu(tmp0, 1, 600); // perform activation

  //Layer dropout skipped as only active during training
```

```c
  load_parameters(params, 151);
  if ((sts = matutil_multiply(tmp0, 1, 150, params+0, 150,
       1, tmp1)))
    return sts;
  if ((sts = matutil_add(tmp1, 1, 1, params+150, 1, 1, tmp1
       )))
    return sts;
  //linear activation requires no action

  for(int i=0; i<rs; ++i)
    ret[i] = tmp1[i]; // copy values to return buffer

  free(tmp0);
  free(tmp1);
  free(params);

  close_parameters();
  return 0;
}
```