Privacy Preserving Deep-Learning Using Intel SGX

Ayesha Samreen, Devendra Sai.M

***Abstract*— With the increasing demand for privacy in deep learning applications, secure computation has become a critical area of research. This project explores a basic implementation of privacy-preserving deep learning using Intel Software Guard Extensions (SGX), a hardware-based Trusted Execution Environment (TEE) that protects data and code integrity during execution. Due to SGX's inherent memory and performance limitations, our focus was on training and testing small-scale models within the enclave. We implemented and evaluated a lightweight neural network on modest datasets to demonstrate the feasibility of end-to-end secure training and inference. Performance were compared between SGX-secured execution and traditional non-enclave environments. Results show that while SGX introduces overhead, it can effectively support privacy-preserving deep learning for constrained models and applications. This work highlights both the potential and limitations of using SGX for secure machine learning**

Keywords: Privacy-preserving machine learning, Intel SGX, secure deep learning, enclave computing, trusted execution environment

I. INTRODUCTION

Deep learning has revolutionized numerous industries, including healthcare, finance, security, and smart infrastructure, by enabling systems to learn complex patterns from large datasets. However, as deep learning becomes increasingly integrated into sensitive applications, privacy and security concerns are growing. Adversaries have demonstrated the ability to exploit machine learning models to extract sensitive information from both training data and model parameters through various attacks such as membership inference, model inversion, and data reconstruction. These risks pose serious challenges, especially when models are deployed in untrusted or cloud environments.

To address these concerns, there is a growing need for privacy-preserving deep learning, where both training and inference processes are protected from unauthorized access or data leakage. Traditional cryptographic techniques like homomorphic encryption and secure multi-party computation offer privacy guarantees but often come at a high computational cost and are difficult to scale.

An alternative and increasingly practical solution lies in Trusted Execution Environments (TEEs)—specifically, Intel Software Guard Extensions (SGX). SGX is a hardware-based security feature available in modern Intel processors, which allows the creation of isolated memory regions called enclaves. These enclaves enable secure execution of code and protection of data from the rest of the system, including privileged software like operating systems and hypervisors. With SGX, sensitive data and model parameters can remain confidential during both training and inference, even in potentially compromised environments.

In this project, we investigate the feasibility of using Intel SGX for privacy-preserving deep learning. Given SGX's strict memory constraints and performance overhead, we focus on a naive, small-scale neural network implementation that fits within enclave limitations. We perform basic model training and testing entirely within the enclave and compare the performance and security trade-offs against standard (non-SGX) execution. Our goal is to demonstrate that while SGX may not yet support large-scale deep learning, it offers a viable foundation for secure learning in lightweight applications.

II. RELATED WORK

With the increasing need to preserve data confidentiality in machine learning applications, several studies have explored privacy-preserving approaches using secure hardware, particularly Intel SGX. These efforts span a wide range of use cases and system architectures, with varying levels of complexity and performance optimization.

In [1], the authors present a distributed machine learning framework for sensitive genomic data, leveraging Intel SGX to enable collaborative learning among untrusted data providers. Their approach implements Secure XGBoost within SGX enclaves, enabling training without exposing raw data or model internals. Additionally, the framework incorporates data-oblivious algorithms to defend against side-channel attacks. While this system supports large-scale collaborative training, it assumes a cluster of SGX-enabled nodes and involves significant engineering effort.

Similarly, the paper [2] proposes a bidirectional trust model where the machine learning model is securely executed within SGX enclaves on the data provider's side. This setup ensures that neither data nor model parameters are exposed, addressing mutual privacy concerns. Notably, this work also ports a Python interpreter into SGX, enhancing flexibility and usability for model developers. However, the framework primarily targets secure inference and relies on complex system integration.

In contrast, [3] explores the client-side deployment of machine learning models. Here, SGX protects model intellectual property and ensures that user queries are processed securely without exposing sensitive input data or the model itself. This approach emphasizes secure inference over the web rather than training, making it particularly relevant for model deployment in consumer applications.

In this project, we take a more fundamental approach: implementing and evaluating a basic deep learning neural network entirely within a single SGX enclave. Our focus

is on understanding the feasibility and limitations of secure training and inference in resource-constrained environments.

III. Hardware/Software

A. Hardware Requirements

The hardware used for this project includes a laptop equipped with Intel processors that support Software Guard Extensions (SGX). Specifically, the CPU must have SGX capability enabled in the BIOS settings, which is commonly available in Intel Core i5, i7, or i9 newer generations. The system used for this work was a laptop with an Intel Core i7 processor 10th generation and SGX enabled in firmware.

B. Software Requirements

On the software side, the development and execution environment was based on Ubuntu 20.04 LTS (64-bit), chosen for its robust compatibility with Intel SGX tools. Essential software packages include the Intel SGX Software Development Kit (SDK), which provides the necessary libraries and tools to build and run enclave applications. The Intel SGX driver is required at the kernel level to enable SGX instructions and enclave memory management. Platform Software (PSW) components facilitate communication between the host application and the secure enclave during execution. For security validation, the Intel SGX Quote Generation Library supports remote attestation processes. Additionally, the Data Center Attestation Primitives (DCAP) package is included to enable flexible attestation in cloud or data-center scenarios without reliance on Intel Attestation Service (IAS). Development tools such as gcc, make, and cmake were used for compilation, along with Darknet (a deep learning library in C to support Intel SGX) to build, train, and test the neural network inside the enclave.

IV. Implementation/Methodology

A. Deep Learning Model Architecture

In our project, we implemented a simple Convolutional Neural Network (CNN) tailored for execution within an Intel SGX enclave. We utilized the Darknet library, an open-source deep learning framework written in C, which aligns well with SGX's support for C and C++ languages. Our CNN architecture consists of two convolutional layers, each followed by a max-pooling layer to reduce spatial dimensions. The convolutional features are then passed through a fully connected layer, and the final output is produced by a softmax activation, enabling classification. This lightweight design is intentional to fit within SGX's memory constraints while demonstrating basic privacy-preserving training and inference. Figure 1 illustrates the block diagram of the network architecture, showing the flow from convolutional operations through pooling and dense layers to classification output.

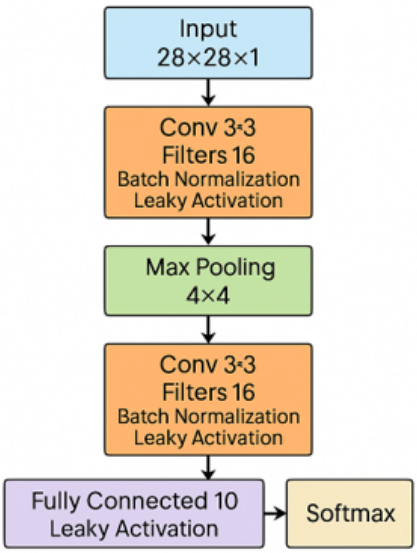

Fig. 1. Block diagram of the convolutional neural network model architecture.

B. Enclave Definition Language (EDL)

The Enclave Definition Language (EDL) file plays a central role in defining the trusted and untrusted interfaces for our SGX enclave. It specifies the **trusted calls (ecalls)** that the host application can invoke inside the enclave, as well as the **untrusted calls (ocalls)** that the enclave uses to communicate with the outside environment.

The trusted calls implemented include:

- enclave_init_ra and enclave_ra_close to initiate and terminate a remote attestation (RA) session, ensuring secure enclave setup.
- verify_att_result_mac to verify the integrity of the attestation quote received.
- put_secret_data to provision sensitive shared secrets securely inside the enclave.
- ecall_build_network to load the neural network configuration and pre-trained weights.
- ecall_train_network and ecall_test_network to execute training epochs and inference, respectively within the enclave.
- ecall_thread_enter_enclave_waiting to handle threading and synchronization for intra-enclave parallelism.

The untrusted calls allow the enclave to interact with the host environment and include:

- ocall_print_string for sending debug and status messages to the host console.
- ocall_start_measuring_training and ocall_end_measuring_training to bracket timing around training phases, facilitating external performance logging.
- ocall_spawn_threads to request the host create additional threads to support parallel computation inside the enclave.

```
enclave {
    from "sgx_tkey_exchange.edl" import *;

    include "sgx_key_exchange.h"
    include "sgx_trts.h"
    trusted {
        public sgx_status_t enclave_init_ra(int b_pse,
                                            [out] sgx_ra_context_t *p_context);
        public sgx_status_t enclave_ra_close(sgx_ra_context_t context);
        public sgx_status_t verify_att_result_mac(sgx_ra_context_t context,
                                                  [in,size=message_size] uint8_t* message,
                                                  size_t message_size,
                                                  [in,size=mac_size] uint8_t* mac,
                                                  size_t mac_size);
        public sgx_status_t put_secret_data(sgx_ra_context_t context,
                                            [in,size=secret_size] uint8_t* p_secret,
                                            uint32_t secret_size,
                                            [in,count=16] uint8_t* gcm_mac);
    ▌   public void ecall_train_network([in, count=size_train_file] char *train_file, int size_tr
ain_file, int num_threads);
        public void ecall_test_network([in, count=size_test_file] char *test_file, int size_test_
file, int num_threads);
        public void ecall_thread_enter_enclave_waiting(int thread_id);
        public void ecall_build_network([in, count=len_string] char *file_string, size_t len_stri
ng, [in, count=size_weights] char *weights, size_t size_weights);

 };
```

Fig. 2. EDL Enclave calls

- ocall_push_weights to stream updated model weights out of the enclave for secure sealing or storage.

Figure 2 and 3 depicts the the trusted enclave code (EDL defined ecalls) and untrusted host code (EDL defined ocalls) architecture.

```
    untrusted {
        void ocall_print_string([in, string] const char *str);

        void ocall_start_measuring_training(int sub_time_index, int repetitions);
        void ocall_end_measuring_training(int sub_time_index, int repetitions);

        void ocall_spawn_threads(int n);

        void ocall_push_weights([in, size=size, count=nmemb] const char *ptr, size_t size, size_t
 nmemb);
    };
};
```

Fig. 3. EDL Host calls

C. Training and Encryption within SGX

At the host application side, the first step involves creating the enclave and verifying its integrity using Intel SGX's remote attestation. The initialize_enclave() as shown in Fig 4 function is responsible for creating (or re-opening) the SGX enclave using the sgx_create_enclave API, which allocates and configures a hardware-protected memory region dedicated to our trusted code. It loads the enclave.signed.so binary—containing the enclave code and metadata signed by Intel—into the secure memory. On successful launch, the function returns an enclave ID (global_eid) for subsequent ECALLs and writes an enclave.token file to optimize future startups.

If enclave creation fails (i.e., the return value is negative), the application aborts initialization. Once the enclave is successfully initialized, the host invokes two primary ECALLs (shown in Fig 5):

1) ecall_build_network parses the CNN configuration file and allocates the model structure securely within the enclave. If pretrained weights are provided, it loads them into enclave memory.
2) ecall_train_network accepts the training data and executes the training loop inside the enclave. It also logs training time and finally stores the resulting model weights.

After each ECALL, the SGX return value (sgx_ret) is printed to indicate success or failure of the secure call.

Upon completion of the training phase, and prior to exporting model parameters, the learned weights are encrypted using AES (Rijndael standard) as depicted in Fig 6. The encryption function inside the enclave converts raw model weights into ciphertext while also generating an authentication tag. This ensures both confidentiality and integrity. Finally, the ocall_push_weights function is invoked to securely send the encrypted weights from enclave memory to the host.

```
if (initialize_enclave(&global_eid, "enclave.token", "enclave.signed.so") < 0)
{
    printf("\n failed to initialize enclave");
    if (log_file)
        fclose(log_file);
    return 1;
}
printf("\n enclave initilaized");
```

Fig. 4. Enclave initialization at host side

```
        sgx_status_t status = ecall_build_network(global_eid, cfg, cfg_length+1, N
ULL, 0);
        printf("\n sgx-status after building %#08x\n", status);
        status = ecall_train_network(global_eid, train, length+1, number_of_additi
onal_threads);
        printf("\n sgx-status after training %#08x\n", status);
```

Fig. 5. Enclave call from host side for training

```
sgx_aes_gcm_128bit_tag_t tag;
sgx_status_t st = sgx_rijndael128GCM_encrypt(
    &enclave_aes_key,
    (uint8_t*)file_bytes, offset,
    cipher,
    iv, sizeof(iv),
     nullptr, 0,
    &tag
);
if (st != SGX_SUCCESS) {

     printf("\n Encrypt failed: 0x%x\n", st);
    // handle error…
    free(cipher);
    free(file_bytes);
    return;
}
printf("\n Encryption Successful");
```

Fig. 6. AES encryption routine for weights before leaving the enclave

D. Decryption and Testing within SGX

During the testing phase, the encrypted model weights are securely passed back into the enclave. The enclave invokes its AES decryption routine as pictured in Fig 8, which transforms the ciphertext back into the original plaintext weights. Simultaneously, the authentication tag is verified to ensure the data has not been tampered with during storage or transit. If the tag is valid, the decrypted weights are accepted and ready for inference.

Once the decrypted weights are available, ecall_build_network is called, as listed in Fig 7, again

to construct the CNN architecture with the provided configuration file and decrypted weights. A print statement on the host side confirms whether the model initialization is successful.

Following model setup, ecall_test_network is executed to evaluate the trained model on test data inside the enclave. The output accuracy or predictions can then be optionally exported to the host for further analysis or display.

```
    sgx_status_t status = ecall_build_network(global_eid, cfg, cfg_length + 1,
weights, weights_length + 1);
    printf("\n sgx-status after building %#08x\n", status);
    status = ecall_test_network(global_eid, test, length + 1, number_of_additi
nal_threads);
    printf("\n sgx-status after testing %#08x\n", status);
}
```

Fig. 7. ECALL flow for building the network and testing it during inference.

```
sgx_status_t st = sgx_rijndael128GCM_decrypt(
    &enclave_aes_key,
    cipher,      pt_len,
    (uint8_t*)file_bytes + 8,
    iv, iv_len,
    NULL, 0,
    tag
);
if (st != SGX_SUCCESS) {
    printf("\n Decrypt failed: 0x%x\n", st);
    return;
}
printf("\n Decrypt successful");
```

Fig. 8. AES decryption routine with authentication tag validation inside enclave.

V. Experimental Setup & Results

To evaluate the feasibility of secure deep learning training and inference using Intel SGX, we conducted a series of experiments on a laptop equipped with SGX-compatible hardware and running Ubuntu 20.04 LTS. The system was configured with the Intel SGX SDK, Platform Software (PSW), SGX driver, and the Data Center Attestation Primitives (DCAP) components to enable full remote attestation and enclave execution support.

As the first step of execution, we performed remote attestation to verify the authenticity and integrity of the enclave. This process ensures that the enclave has not been tampered with and is running on a genuine Intel SGX-enabled processor. The successful attestation confirms the secure launch of the enclave and its eligibility for provisioning secrets. Figure 9 illustrates the output from this stage.

Due to memory and performance limitations of the SGX secure enclave, we used a flattened dataset as input for training. A simple convolutional neural network (CNN) with two convolution layers, a max-pooling layer, and a fully connected softmax output layer was used. The network was implemented using the Darknet library, which is written in C and easily portable into the enclave environment.

Training was performed inside the enclave for 10 epochs. During this phase, the model parsed the configuration, initialized the network, and processed the training data securely.

```
ATTESTATION RESULT RECEIVED - 145 bytes:
{
0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0,
0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0,
0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0,
0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0,
0x0, 0x97, 0x9e, 0xb9, 0x5a, 0xdd, 0x14, 0x17,
0xf2, 0xfa, 0xad, 0xfa, 0xd7, 0x66, 0x73, 0xfd,
0x71, 0x57, 0xf4, 0xe9, 0x8d, 0xee, 0xaf, 0x42,
0x5e, 0xbb, 0x8a, 0x4c, 0xd4, 0x84, 0xdc, 0x83,
0x6a, 0x8, 0x70, 0xd, 0xf2, 0x42, 0x8b, 0x2b,
0xee, 0x42, 0xb0, 0x85, 0xe5, 0xbf, 0x99, 0xc5,
0x22, 0xf8, 0x37, 0xf7, 0xee, 0xb6, 0x2c, 0xd5,
0x8c, 0x37, 0xa2, 0xd2, 0x51, 0xed, 0x45, 0xf9,
0x65, 0xc6, 0xf6, 0xa0, 0xd1, 0xa5, 0x27, 0xa0,
0x9e, 0xf1, 0xb8, 0x59, 0x3e, 0x1b, 0xb3, 0x67,
0x93, 0x8, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0,
0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0,
0x0, 0x14, 0x7e, 0xcf, 0x89, 0xd5, 0x67, 0xe3,
0x4a, 0xcf, 0xc3, 0xc7, 0xe6, 0x7, 0xdb, 0x60,
0xff
}

Secret successfully received from server.
Remote attestation success!
```

Fig. 9. Remote attestation results confirming enclave authenticity and integrity.

The training loss was recorded at each epoch. After the final epoch, the model weights were encrypted using AES encryption (Rijndael standard) before being pushed to the untrusted host memory. This encryption step ensures that the weights remain protected during storage or transmission. The entire training process, including encryption, took approximately 41 seconds. Figure 10 shows a snippet of the training output log.

```
training function entered
network builded
sgx-status after building 00000000

entered the ecall_train_network
done loading training data
data loaded - Matrices sizes:
X= 196 x 10001
y= 10 x 10001
ocall made to begin performance measurement
epoch  1 finished with loss: 2.267116

epoch  2 finished with loss: 2.095558

epoch  3 finished with loss: 1.909830

epoch  4 finished with loss: 1.833145

epoch  5 finished with loss: 1.800307

epoch  6 finished with loss: 1.782458

epoch  7 finished with loss: 1.768328

epoch  8 finished with loss: 1.756526

epoch  9 finished with loss: 1.745324

epoch 10 finished with loss: 1.733372

Training timeavg=41.253120
Encryption Successful
saved weights to weights_from_enclave.weights

sgx-status after training 00000000
```

Fig. 10. Training phase inside enclave: Epoch-wise loss values and timing.

In the inference phase, the encrypted weights were passed back to the enclave. The enclave then performed AES decryption to recover the original weights and verified their integrity using the associated authentication tag. Upon successful verification, the network was reinitialized with the decrypted weights, and testing was carried out on the test

dataset within the enclave. The entire testing pipeline, including decryption, took approximately 6.9 seconds. Figure 11 provides the output from the testing phase.

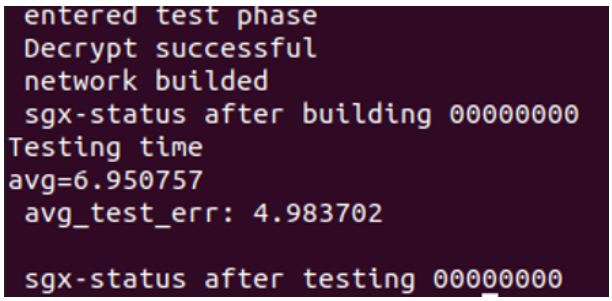

Fig. 11. Testing phase results showing successful decryption and inference.

To assess the performance overhead introduced by Intel SGX, we performed a comparative evaluation. The same model and dataset were executed on the CPU outside the enclave. For a fair comparison, all preprocessing steps were kept consistent. Table I summarizes the timing results. During training, we observed a 50% increase in execution time when using the enclave. This overhead is attributed to the memory encryption and limited EPC size within SGX. The testing phase exhibited a more significant slowdown, largely due to additional cryptographic operations (decryption and integrity checks) and limited multithreading support inside the enclave.

TABLE I
PERFORMANCE COMPARISON: WITH AND WITHOUT INTEL SGX

Phase	Without SGX (s)	With SGX (s)
Training	27.43	41.07
Testing	0.63	6.95

VI. CONCLUSION AND FUTURE WORK

A. Conclusion

In this work, we explored the feasibility of using Intel Software Guard Extensions (SGX) to provide confidentiality and integrity guarantees for deep learning workflows. We implemented a simple convolutional neural network (CNN) within the SGX enclave using the Darknet library and demonstrated both secure training and inference with encrypted model parameters. The enclave-based execution ensured that sensitive data and learned parameters were protected from external access, even in an untrusted environment.

We evaluated the system through remote attestation and conducted experiments to measure the performance trade-offs of using SGX. Our results show that, while there is a measurable performance overhead—approximately 50% during training and higher during inference—the overhead is acceptable for small-scale applications where privacy is critical. These findings validate that hardware-based trusted execution environments like Intel SGX can serve as practical building blocks for secure deep learning.

B. Future Work

There are several promising directions to extend this project:

- **Support for Larger Models:** Current SGX enclaves are limited by their Enclave Page Cache (EPC) size. Future work can explore model compression, quantization, or enclave paging strategies to enable training of more complex networks.
- **Streaming and Batching:** Implementing support for data streaming and mini-batch processing inside the enclave can help manage larger datasets and improve training efficiency without exceeding memory limits.
- **Side-Channel Defense:** While SGX provides hardware-based isolation, it is still vulnerable to side-channel attacks. Incorporating data-oblivious operations and constant-time execution can further harden the implementation.
- **End-to-End Privacy Pipeline:** Extending this work to build an end-to-end secure pipeline—from data collection to model deployment—would enable practical real-world applications in healthcare, finance, and IoT.
- **Integration with Federated Learning:** Combining SGX with federated learning can enable secure collaborative training across multiple devices or organizations without centralizing raw data.

Through this project, we demonstrate a viable path to bridging the gap between machine learning and data privacy using trusted hardware. With further optimizations and enhancements, this approach holds great promise for secure, decentralized AI systems in privacy-sensitive domains.

VII. ACKNOWLEDGEMENTS

We would like to thank Dr. Akhilesh Tyagi, and Iowa State University for their invaluable support throughout the project.

REFERENCES

[1] M. S. Aldeen, C. Zhao, Z. Chen, L. Fang, and Z. Liu, "Privacy-preserving collaborative learning for genome analysis via secure xgboost," *IEEE Transactions on Dependable and Secure Computing*, vol. 21, no. 6, pp. 5755–5765, 2024.

[2] C. Liu, L. Zhang, Y. Dai, F. Chen, H. Chen, and P. Zhong, "Intel sgx-based trust framework designed for secure machine learning," in *2022 IEEE 6th Conference on Energy Internet and Energy System Integration (EI2)*, 2022, pp. 1621–1627.

[3] D. Ács and A. Coleşa, "Securely exposing machine learning models to web clients using intel sgx," in *2019 IEEE 15th International Conference on Intelligent Computer Communication and Processing (ICCP)*, 2019, pp. 161–168.

[4] P. C. Ling and U. U. Sheikh, "Secure deep learning inference with intel sgx on intel ice lake-sp xeon processor," in *2024 10th International Conference on Smart Computing and Communication (ICSCC)*, 2024, pp. 55–59.

[5] C. Zhao, D. Saifuding, H. Tian, Y. Zhang, and C. Xing, "On the performance of intel sgx," in *2016 13th Web Information Systems and Applications Conference (WISA)*, 2016, pp. 184–187.