Towards Auditable Properties of Machine Learning Systems

by

Prach Chantasantitam

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2026

© Prach Chantasantitam 2026

Author's Declaration

This thesis consists of material all of which I authored or co-authored: see Statement of Contributions included in the thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Statement of Contributions

This thesis consists of works supervised by Prof. N. Asokan. The contributions of co-authors of the two works are as follows:

PAL*M: Property Attestation for Large Generative Models

This work, presented in Chapter 3, is in collaboration with Adam Caulfield and Vasisht Duddu from the University of Waterloo, and Lachlan J. Gunn from Aalto University.

- All authors contributed to the idea and design of PAL*M, specifically regarding the move to confidential virtual machines with GPU support and the solution to handle large datasets to improve Laminator.
- Prach implemented PAL*M, set up, and ran the experiments, with Adam providing expert advice.
- All authors contributed to the writing of the paper.

EVOLVE: Enabling Verifiable Output Lineage for Virtualized ML Workload Execution

This work, presented in Chapter 4, is in collaboration with Adam Caulfield from the University of Waterloo.

- Prach and Adam came up with the idea and design EVOLVE.
- Prach implemented EVOLVE and ran the experiments, with Adam providing expert advice.
- All authors contributed to the writing of the paper.

This thesis is a revised and extended version of both papers.

Abstract

Machine learning (ML) property attestations are a set of mechanisms through which model providers (*provers*) can attest to properties about their machine learning models and datasets to *verifiers*, providing auditability and demonstrating compliance with regulatory or internal policies or standards. However, existing approaches do not simultaneously achieve efficiency, scalability, versatility, and security. Systems based on *trusted execution environments* (TEEs) have been shown to be more efficient and scalable; however, prior work is limited to CPU-only computation, making it inefficient to run large generative models. Additionally, TEE-based approaches rely on static attestation, which cannot detect run-time attacks such as loading a malicious model checkpoint or corrupting model weights in memory after the system has been launched and measured, allowing an adversary to manipulate inference outputs at run-time. As such, the limitations of existing TEE-based approaches include: (1) they do not scale to large generative models and datasets, and (2) their sole reliance on static attestation makes them vulnerable to various run-time attacks.

To address the first limitation, I present PAL*M, a property attestation framework for large generative models, illustrated by large language models. PAL*M addresses three key challenges: (1) attesting properties over large datasets accessed via random sampling outside trusted memory, (2) supporting heterogeneous CPU-GPU execution environments, and (3) defining attestable properties for generative model operations. PAL*M leverages *confidential virtual machines* with *security-aware* GPUs and employs *incremental multiset hashing* to efficiently measure dataset integrity across memory-mapped accesses. PAL*M is implemented atop Intel TDX and NVIDIA H100, and formally modeled using the Tamarin Prover symbolic verification tool, confirming that its security guarantees are upheld under the defined threat model.

For the second limitation, I present EVOLVE, a proof-of-execution (PoX) mechanism for confidential virtual machines that enables run-time property attestation of ML operations. EVOLVE has two components: an instrumentation module and a PoX monitor. The instrumentation module preprocesses an ML application to ensure that both memory accesses and indirect calls pass through the PoX monitor. At run-time, the PoX monitor inspects memory accesses to determine whether the correct subset of the application is modifying the output and validates indirect calls to ensure that control flow cannot be redirected into the monitor to alter its state, constructing PoX evidence accordingly. The PoX monitor is implemented as an extension of TD-Shim, and its efficiency and security are evaluated on Intel TDX using classifier model routing as a proof-of-concept application.

Acknowledgements

First and foremost, I would like to thank my advisor, Prof. N. Asokan, for his supervision and support throughout my Master's program. His guidance has been invaluable in helping me learn to think and tackle problems like a researcher. I would also like to thank Dr. Adam Caulfield, who has been a great mentor throughout the program, offering much-needed advice and always making himself available to answer any technical questions I had. I could not have asked for a better mentor. Additionally, I would like to thank my undergraduate advisor, Dr. Chanathip Namprempre, who gave me the push I needed to pursue a Master's degree. Even after I graduated, she continued to provide support and advice. I would not be where I am without you.

I would also like to thank my labmates in the SSG group, who were always available to help and discuss new ideas. I thank my friends, who, even with the time difference between Canada and Thailand, still woke up early or stayed up late to catch up with me and check in on how I was doing. Finally, I would like to thank my family for always being supportive of everything I do. Their love and encouragement have carried me through every challenge along the way.

Table of Contents

Author's Declaration ii

Statement of Contributions iii

Abstract iv

Acknowledgements v

List of Figures x

List of Tables xii

1 Introduction 1

2 Background 4

2.1 Trusted Execution Environments 4

2.2 Attestation Protocols 5

3 PAL*M: Property Attestation for Large Generative Models 7

3.1 Introduction 7

3.2 Background 8

3.2.1 Large Language Model Background 8

3.2.2 Tamarin Prover 9

3.3 Problem Statement . . . 10
3.3.1 System Model . . . 11
3.3.2 Adversary Model . . . 12
3.3.3 Challenges and Requirements . . . 12
3.4 PAL*M Design . . . 13
3.4.1 Overview of PAL*M . . . 13
3.4.2 Dealing with Large Datasets . . . 15
3.4.3 Property Measurements for Large Generative Models . . . 17
3.4.4 From Measurement to Attestation . . . 22
3.4.5 Trusted Authority, Reference Values, Verification . . . 24
3.5 Experiment Setup . . . 26
3.5.1 Datasets, Models, and Metrics . . . 26
3.5.2 Prototype Configuration . . . 27
3.6 Evaluation . . . 28
3.6.1 Performance Analysis . . . 28
3.6.2 Security Analysis . . . 32
3.6.3 Scalability & Versatility . . . 34
3.7 Formal Modeling with Tamarin . . . 35
3.7.1 Security Goals . . . 35
3.7.2 Case 1: In-Memory Public Inputs/Outputs . . . 36
3.7.3 Security Proofs for Case 1 . . . 39
3.7.4 Case 2: In-memory Private Inputs/Outputs . . . 40
3.7.5 Case 3: Memory-mapped Public Inputs/Outputs . . . 42
3.7.6 Case 4: Memory-mapped Private Inputs/Outputs . . . 45
3.7.7 Verification Results . . . 46
3.7.8 Multiset hashing security reduction . . . 46
3.8 Discussion . . . 49

4 EVOLVE: Enabling Verifiable Output Lineage for Virtualized ML Workload Execution **51**

4.1 Introduction . . . 51

4.2 Background . . . 52

4.2.1 Run-time attacks in ML software . . . 52

4.2.2 Proofs of Execution . . . 53

4.2.3 LLVM Compiler . . . 54

4.3 Problem Statement . . . 55

4.3.1 System and Adversary Model . . . 55

4.3.2 Challenges . . . 56

4.4 EVOLVE Design . . . 57

4.4.1 Overview of EVOLVE . . . 57

4.4.2 Instrumentation . . . 59

4.4.3 Monitor . . . 61

4.4.4 EVOLVE-based PoX Protocol . . . 64

4.5 Implementation & Evaluation . . . 66

4.5.1 Prototype Configuration . . . 66

4.5.2 Metrics . . . 68

4.5.3 Fixed costs . . . 68

4.5.4 Performance Costs . . . 69

4.5.5 Security Analysis . . . 72

4.5.6 Discussion . . . 74

5 Related Work **78**

5.1 Cryptographic Proofs of ML operations . . . 78

5.2 TEE-based ML attestation . . . 78

5.3 PoX Architectures . . . 79

6 Conclusion **81**

References 83

Glossary 100

List of Figures

2.1 Traditional remote attestation protocol: a prover (Prv) demonstrates its software integrity to a verifier (Vrf). 5

3.1 Desired property attestation for ML, in which Prv interacts with an Initiator (Inr). Vrf requests evidence from Inr, and may consult a Trusted Authority (Tru) to obtain reference values of a desired property. 11

3.2 High-level overview of PAL*M-enabled property attestation. Inr interacts with untrusted components of Prv, which saves inputs to disk. PAL*M reads and measures inputs, runs the operations (including GPU), and measures all CPU and GPU outputs. All measurements are extended to `REPORTDATA`, which is used to as input to obtain `TDREPORT` from TDX Module. Finally, a `QUOTE` is generated by invoking the QE and returned to Inr. 14

3.3 Depiction of PAL*M 's handling of *in-memory* (left) and *memory-mapped* (right) dataset. In the in-memory approach, the entire dataset is loaded and measured once at load time. In the memory-mapped approach, individual records are measured when accessed using a multiset hash to ensure consistent measurements regardless of sampling order. 16

3.4 PAL*M Property Attestation Protocol between Inr and Prv. Inr sends request to Prv containing untrusted VMM, QE, trusted TDX Module, and attested PAL*M TD. Prv responds after performing the requested operation, measuring relevant inputs and ouputs, and generating both a `TDREPORT` and `QUOTE`. 23

3.5 `MSH` collision game. 47

3.6 Game hops to prove Theorem D.1. 48

4.1 PoX built atop traditional remote attestation: a prover (Prv) demonstrates integrity of its software and execution properties to a verifier (Vrf). 54

4.2 System overview of EVOLVE: an ML App with multiple operations is first instrumented before it is loaded and measured into EVOLVE-enabled TD. Upon receiving a request for a particular operation, EVOLVE's monitor tracks whether only the requested operation alters the reported output data. 58

4.3 EVOLVE instrumentor: extends the LLVM pipeline to replace each `store` and `call` IR with a call to the EVOLVE monitor. 60

4.4 EVOLVE monitor internals: shows the process for extracting labels from the instrumented ELF binary to determine the location of memory regions, the FFI layer exposed to the payload, and the process for validating memory write and indirect call instructions. In the example, `Function1` performs a memory write, which is routed through the FFI layer to the monitor for the memory write check. 62

4.5 EVOLVE-enabled PoX protocol between Vrf and Prv, comprising TDX and EVOLVE TD. The protocol assumes App has already been instrumented. At load time, the TDX Module measures the instrumented binary and launches the TD. The monitor initializes its registers, all section regions and, sets EXEC = 1. At request time, Vrf sends a operation choice, the monitor configures the authorized operation, mediates memory writes and indirect calls, and generates a hardware-attested `QUOTE`. Vrf verifies the quote, including validating that EXEC = 1 65

4.6 Monitor call time as the number of operation sections increases (mean ± stdev over 30 runs, run 1 excluded as warm-up). Each bar represents the average of 100 `ffi_mem_write` calls per run. 72

List of Tables

3.1 Notation Summary 18

3.2 Operations and property attestations in PAL*M 22

3.3 Performance for proofs of dataset attribute distribution and preprocessing using BookCorpus (Full) and BookCorpus (1/100). 28

3.4 Performance for proof of training with GPT-2 on preprocessed BookCorpus. 29

3.5 Performance for proof of model optimization (fine-tuning and quantization). 30

3.6 Performance for proof of evaluation based on MMLU and BLEU score with WMT14(DE-EN) dataset. 30

3.7 Performance of proof of inference for one query-response and proof of session inference for 50 query-responses. 31

3.8 Records per workers (R/W), compute time per record (C/R), and raw input measurement time per record (I/R) for memory-mapped operations. I/R remains roughly constant across operations, while C/R varies significantly, indicating that raw measurement overhead is higher when C/R is low. . . . 32

3.9 Statistics for lemma verification: Max Steps denotes the highest steps for any lemma; Total Steps is the sum of steps across all lemmas; Peak Mem. is the maximum RAM usage; Time is the total verification run-time. 47

4.1 Summary of models used in evaluation. We train a small and large configuration of DT and MLP classifier models using the Breast Cancer Wisconsin Diagnostic (BCWD) [174] and ISOLET datasets [34]. The DT configuration corresponds to the final trained model structure (number of trees, nodes, leaves, and depth), while the MLP configuration corresponds to the final network architecture (layer sizes and depths). Finally, the last column shows the memory size of the model. 67

4.2 End-to-end performance costs of EVOLVE for model routing scenario (mean ± stdev, $n = 30$). 69

4.3 Run-time overhead of EVOLVE monitor. Mean ± stdev over 30 runs (excluded the first run). 71

Chapter 1

Introduction

Machine learning (ML) models are widely deployed across domains such as healthcare, finance, and autonomous systems [169, 64, 175]. Emerging regulations (e.g., EU's AI Act [4]) require models meet certain *properties* related to accuracy, training procedures, data provenance, and inference. Providing proof of such properties may be challenging for confidential datasets or models.

ML property attestation enables a *model provider* to demonstrate properties about their model or datasets to a *verifier* [40]. For practical deployment, it must be efficient, scalable, versatile, and secure. While approaches based on zero-knowledge proofs [52, 1, 94, 145, 160, 65] or repurposing of inference attacks [177, 44] have been proposed, they do not meet all these requirements. Even with recent advances in proof efficiency [160, 138], approaches based on zero-knowledge proofs (ZKP) still incur substantial overhead that grows with model size, making them impractical for modern large language models (LLM). In contrast, approaches based on trusted execution environments (TEE) [77, 79, 7, 11], first proposed in Laminator [41], have been shown to meet these requirements for classifiers. Laminator was built on Intel Software Guard eXtensions (SGX) [77] central processing unit (CPU) to produce attestations of simple ML properties such as proof of training, proof of inference, proof of evaluation, and proof of dataset distribution.

However, Laminator's main limitations are twofold: first, it does not scale to generative models, such as LLMs, due to the memory constraints of SGX and the lack of native support for graphics processing units (GPU). While moving to a larger TEE such as confidential virtual machines (CVM) would be a natural step, datasets used for LLMs training are still often too large to fit into memory even for CVMs (e.g., the Pile dataset is over 800 GB [50]). Thus, popular LLM frameworks (e.g., Hugging Face Dataset [69]) often rely

on *memory mapping*, which results in passing data between TEE memory and untrusted storage [66]. Additionally, datasets are often accessed via *random sampling* [68] for LLMs operations such as pretraining and fine-tuning. In the case of sampling from untrusted storage, a straightforward approach (e.g., a hash chain over accessed records) would be unsuitable, as the resulting measurement will differ due to the order in which records are accessed. Moreover, generative models involve operations (e.g., pretraining, fine-tuning, and chat sessions) and CPU–GPU interaction (e.g., for performance) that are unaddressed by Laminator.

As such, simply porting Laminator to CVMs alone does not solve these challenges. To enable ML property attestations for generative models, we must address three key issues: (1) handling large datasets outside of TEE memory sampled in a *random* fashion, (2) supporting CPU–GPU computing environments, and (3) defining property attestations for common operations related to generative models.

Second, because of the static nature of TEE attestation, run-time attacks can corrupt property measurements. Run-time attacks are prevalent in well-known ML libraries (e.g., PyTorch [137], TensorFlow [166], those arising from `pickle` [33, 105, 179]) and are demonstrated by recently disclosed vulnerabilities [115, 114, 113, 112, 111, 117, 116]. An adversary may corrupt or substitute model weights after the system has been measured [116, 117], or manipulate the execution path to produce outputs inconsistent with the attested model [113, 112, 114]. Another example is corruption of *model routing* mechanisms, which dynamically select a model from a set of collocated models to execute at run-time. Recent work shows that such routing decisions can be adversarially manipulated or rerouted without modifying the underlying models or code [153, 93, 162, 85]. Consequently, static TEE attestation does not capture what actually executes at run-time; hence, there is a need to incorporate dynamic behavior measurements into property attestations.

While *proof of execution (PoX)* [22, 120, 110, 140] extend traditional remote attestation techniques to attest to characteristics about the execution of an attested software, the current approaches for operation-level PoX are tailored to embedded systems and cannot be directly applied to CVMs. Extending PoX to CVMs running ML software requires addressing the following challenges: (1) enabling intra-CVM isolation between the expected application and a PoX measurer, (2) monitoring access to outputs of the expected application made by all software in the CVM, (3) ensuring exclusive access to execution state to a PoX measurer during run-time.

To address the first limitation, I define *ML property attestations of large generative models* and present PAL*M, the first framework to enable them, using LLMs for illustration. PAL*M employs *incremental multiset hashing* [32] to measure large datasets that

are accessed via memory mapping and random sampling from outside protected memory. PAL*M is built on PyTorch [136] and the Hugging Face Datasets library [69], and supports CPU–GPU computing environments. Furthermore, I show that PAL*M meets all desired requirements, including efficiency for common operations, using real-world datasets/models, and security, by formally modeling and verifying PAL*M's property attestation protocol using Tamarin Prover [103]. The prototype runs atop Intel Trust Domain eXtensions (TDX) [79] and NVIDIA H100 [29], presented in Chapter 3.

For the second limitation, I present EVOLVE, a mechanism to enable PoX in CVMs, allowing run-time property attestation of ML operations. EVOLVE has two components: an instrumentation module and a PoX monitor. The instrumentation module preprocesses ML applications by replacing memory-write and indirect-call instructions with requests to the PoX monitor. At run-time, the PoX monitor inspects accesses to the memory region containing the operation outputs to determine whether the anticipated subset of the ML application is performing the access, and validates indirect calls to prevent control flow from being redirected to internal monitor functions. The PoX monitor maintains internal metadata that tracks the trustworthiness of the output based on the code that modifies it at run-time. This metadata is then used as an additional input when constructing run-time property attestation evidence. As a proof of concept, I use classifier model routing to illustrate how EVOLVE can effectively detect which model is executed at run-time. The prototype is built as an extension to TD-Shim [36], and its efficiency and security are evaluated on Intel TDX, as presented in Chapter 4. We conclude Chapter 4 by outlining future work towards formal verification of EVOLVE, adapting to kernel payloads, and enabling support for large generative models.

Chapter 2

Background

This chapter provides background common to all contributions in this thesis. Each subsequent chapter also includes a chapter-specific background section covering material relevant only to that contribution.

2.1 Trusted Execution Environments

TEEs are a set of architectural components that provide a secure execution environment that is isolated from the "rich" host environment. They guarantee that software executing outside of the TEE cannot directly read or write to data inside the TEE's trusted boundary. The exact details of the trusted boundary vary based on the particular approach. For example, Intel SGX [77] creates TEE in the user-space *enclaves* and Arm TrustZone [11] creates a TEE containing all system resources: e.g., separate trusted operating system (OS) and applications with dedicated peripherals.

Recent technologies have also introduced CVMs, which enable entire virtual machines (VM) to execute within the trust boundary of a TEE (e.g., AMD SEV-SNP [7], Intel TDX [79], and Arm Realms [92]). An advantage provided by CVMs is that they enable executing unmodified and full-feature operating systems without requiring TEE-specific rewrites. Additionally, TEE-aware GPUs have recently emerged (e.g., with NVIDIA H100 GPU). As such, CVM providers have released drivers to support integration with TEE-aware GPUs. This has enabled the potential for heavy computations (e.g., ML training) within a TEE. Both CPUs and GPUs with TEEs typically support *remote attestation* to prove trustworthiness to verifiers.

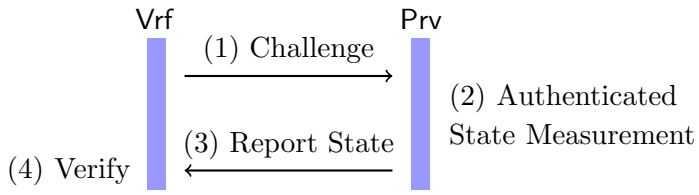


Figure 2.1: Traditional remote attestation protocol: a prover (Prv) demonstrates its software integrity to a verifier (Vrf).

Intel TDX [79] enables isolated CVMs in the form of trust domains (TD). Upon creation of a TD, invoked by the untrusted virtual machine manager (VMM), the TDX module computes a measurement of TD components during TD creation, which can be used later for attestation. To attest TD's state, the TD invokes the TDX module to generate a `TDREPORT`, which is generated by the CPU computing a message authenticated code (MAC) over measurements of the TD image, the TDX module image, and any `REPORTDATA` extended by the TD at runtime. The resulting `TDREPORT` and MAC are passed by the TDX Module to the quoting enclave (QE), a special-purpose Intel SGX enclave [80]. The QE uses contents of `TDREPORT` to generate a SGX-style `QUOTE` using a provisioning certification key [75]. Before generating this `QUOTE`, the QE first validates the MAC, ensuring integrity is maintained when passed by the untrusted VMM.

2.2 Attestation Protocols

Remote attestation is a protocol in which a verifier (Vrf) challenges a prover (Prv) to demonstrate that they are installed with the correct configurations [121]. The steps for a traditional remote attestation protocol are visualized in Figure 2.1. The authenticated integrity measurement in step (2) is computed by Prv's root of trust (RoT) and is typically a MAC or digital signature.

Thus, one requirement of remote attestation is that Prv's RoT can:

- securely store a private key, such that it is only accessible to itself;
- securely use a private key, such that key material does not reside in registers or external memory during or after usage.

These requirements imply some level of hardware assistance. Hence, TEEs can naturally enable remote attestation with their properties for isolated data and execution. For Intel TDX, step (2) involves computing `TDREPORT` and `QUOTE`.

Property attestation [146] is where Vrf is interested in *properties* of Prv's configuration rather than a specific configuration itself. Vrf determines whether Prv's configuration (reported through authenticated integrity measurement) matches a particular property by obtaining *trusted reference values* corresponding to the property of interest; for example, from a certification, a secondary attestation (or set) [41], or a *trusted authority* (Tru) [76, 126]. If the state measurement from Prv is among the trusted reference values obtained, Vrf can verify that the property holds on Prv. Vrf must obtain reference values from a trusted channel rather than to recompute them. Every property of interest is associated with a *measurer* on Prv used to measure the state of the property.

The notion of property attestation has been extended recently to cover ML data and processes [40, 41, 146]. In particular, Laminator [41] leverages Intel SGX [77] to attest properties of training data, model, and training process. This works by executing ML operations inside an SGX enclave, measuring the inputs and outputs of the target operation (e.g., training, inference, or dataset statistics), and using SGX's quote mechanism to return these measurements to the verifier that issued the attestation challenge. Such attestations can then be used for verifiable ML property cards for transparency (e.g., model card [107], datasheet [53, 135], and inference card [41]).

Chapter 3

PAL*M: Property Attestation for Large Generative Models

3.1 Introduction

This chapter presents PAL*M, a framework enabling property attestation for large generative models. PAL*M employs incremental multiset hashing [32] to enable compact secure integrity tracking of large, randomly-accessed, datasets, and TEE-aware GPUs to efficiently and securely ensure integrity of heterogeneous operations. Finally, we define property attestations for a comprehensive set of operations (e.g., fine-tuning, quantization, LLM chat sessions) that do not require revealing confidential datasets or models to Vrf. We prototype and evaluate PAL*M using Intel TDX and NVIDIA H100.

In summary, we claim the following contributions: we

1. propose use of incremental multiset hash functions to construct representative measurements of datasets in external storage that are randomly sampled at run-time (Section 3.4.2);
2. define how to measure properties of generative model operations, incorporating GPU attestation evidence without exposing confidential details (Section 3.4.3);
3. specify a property attestation protocol that shows how such measurements and outputs can be combined to prove data/models were produced by a PAL*M-equipped CPU–GPU configuration (Section 3.4.4);
4. implement a fully functional and open-source [8] prototype of PAL*M built on commercially available CVM and TEE-aware GPU – Intel TDX and NVIDIA H100 (Section 3.5); and

5. show that PAL*M meets desired requirements, including efficiency for common operations, using real-world datasets/models, and security, by formally modeling and verifying PAL*M's property attestation protocol using Tamarin Prover (Section 3.6 and Section 3.7).

3.2 Background

3.2.1 Large Language Model Background

LLMs are massive neural networks trained on large text corpora to generate and understand natural language. We describe the key aspects and operations of LLMs relevant to this thesis.

Training. LLMs use transformer architectures with billions of parameters [171, 20], and are trained to predict the next tokens given previous ones. During training, the model learns a probability distribution of the next token from a vocabulary to follow a sequence of tokens. This is often called *pretraining*, as it is typically followed by another phase of training for task-specific optimization/tuning.

Tokenizers. Modern tokenizers operate at word, sub-word (e.g., byte pair encoding, WordPiece), or character level to convert natural-language text into numerical representations for use during LLM training and inference.

Fine-tuning. Since training LLMs is resource- and time-intensive, publicly available trained LLMs are generally fine-tuned on a task-specific dataset. Parameter-Efficient Fine-Tuning (PEFT) and Low-Rank Adaptation (LoRA) fine-tune a small subset of model parameters instead of retraining the entire model [100, 62]. Alternatively, *adapter layers* are trainable modules that can be attached to a pretrained LLM, enabling task-specific adaptation without altering the core model parameters. These approaches are more efficient than fine-tuning the entire LLM.

Evaluating LLMs. Benchmarks for LLMs cover a variety of tasks and compute relevant metrics. These tasks span language understanding (e.g., MMLU [60]), math problems, multiple-choice, classification, and open-ended question answering. Each task is designed to probe specific capabilities of the model.

Chat sessions. LLMs are commonly used for chatbots, which allow a user to continuously query about a particular task and get the responses from the model that consider prior conversational context (e.g., OpenAI's ChatGPT). Chat sessions contain multiple user queries and their responses.

Memory Management. Large generative models require significantly more data than traditional ML models, which often do not fit into main memory (e.g., Pile dataset is over 800 GB [50]). Instead of loading the entire data into memory, current LLM libraries (e.g., Hugging Face) support *memory-mapped datasets* [66], which map the contents of a file into a process's virtual memory space, allowing it to be accessed like regular memory, rather than through explicit file input/output (I/O) operations, despite residing on disk. When a dataset is memory-mapped, dataset records are only loaded into memory as needed, significantly reducing memory usage.

Data Transformations and Processing: Before a dataset is used, it typically undergoes a series of transformations, including *preprocessing* steps like normalization, concatenation, truncation, tokenization, and splitting into training, validation, and test sets, as well as additional processing for removing corrupted samples and formatting the data to match the expected input structure of the training framework [67]. During training, *random sampling* is commonly used to reduce bias introduced by data ordering and to improve model generalization [143]. PyTorch's DataLoader and Sampler abstractions provide randomized sampling [136]. Popular training libraries, such as Hugging Face Transformers, support random sampling and enable it for default training pipelines [68].

3.2.2 Tamarin Prover

Tamarin [103] is a symbolic protocol verification tool. Protocols and adversaries are modeled as multiset rewrite rules with preconditions and postconditions (called *facts*). Security goals are specified as *lemmas*, which Tamarin verifies by matching rule preconditions and postconditions.

```
Rule 0: Example Rule
rule Example_Rule:
  [ Fr(~A), In(B)]
  --[Action(~A,B)]->
  [State(~A,B), Out(h(<~A,B>)]
```

Consider the example in Rule 0; *Example_Rule* has two preconditions: *Fr(∼A)* and *In(B)*. It can only be evaluated if a *fresh* symbol A exists (signaled to Tamarin using *Fr(∼)*), and if a symbol B was received from an $\mathcal{A}$dv-controlled channel (e.g., the network).

This rule creates two postconditions: *State(∼A, B)*, that can be used as a precondition in other rules, and *Out(h(<∼ A, B >))* which sends $h(<\sim A, B>)$ to an $\mathcal{A}$dv-controlled channel. This example uses the *hashing* built-in, which provides a representation of a

cryptographic hash function *h()*, to output the hash of A and B concatenated. We use *h()* in our model, along with *signing* built-in, which provides digital signature functions *sign()* and *verify()*.

An *action fact* records that a rule was evaluated. In Rule 0, the action fact *Action(*$\sim A$*, B)* records that this rule was evaluated with symbols A and B. Lemmas in Tamarin must be specified using *symbols* and *action facts*, not precondition or postcondition facts.

```
Lemma 0: Example Lemma
lemma Example_Lemma:
  "All A B #i.
  Action(A,B) @i ==> not (Ex #j. K(A) @ #j)
```

Lemma 0 is an example lemma *Example_Lemma* using the symbols: A, B, and $\#i$. The notation $\#i$ tells Tamarin that the symbol i is a time variable. This lemma states: given *Action(A,B)* occurs at time $\#i$, then there does not exist a time $\#j$ in which $\mathcal{A}$dv learns A. $\mathcal{A}$dv's knowledge is expressed using the built-in fact $K()$.

Assuming Rule 0 is the only rule, Tamarin evaluates explores all possible executions, controlling inputs via *In()* and outputs via *Out()* to attempt to learn A. This lemma is *verified* no execution exists in which A is learned.

3.3 Problem Statement

Our goal is to design a property attestation framework for generative models to prove to Vrf that a claimed property is (a) correctly measured and (b) not false or manipulated.

Unlike typical remote attestation, which is an interactive protocol between Vrf and Prv, ML property attestation must be non-interactive with Vrf. We propose the protocol depicted in Figure 3.1. Prv interacts with an *initiator* (Inr), who sends a request to Prv for a particular operation, along with other request-specific data (e.g., inputs or a challenge when freshness is required). Upon request, Inr makes them available for Vrf, who may consult Tru for property reference values to verify the evidence.[1]

[1] Vrf may not require consulting Tru for reference values for each property attestation response, and may use other means (e.g., previous attestations). We discuss further in Section 3.4.5.

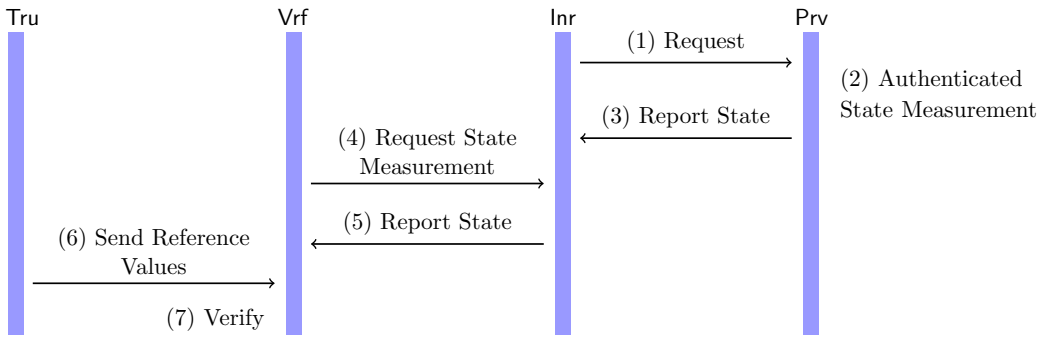


Figure 3.1: Desired property attestation for ML, in which Prv interacts with an Initiator (Inr). Vrf requests evidence from Inr, and may consult a Trusted Authority (Tru) to obtain reference values of a desired property.

3.3.1 System Model

We consider Prv that trains, evaluates, or deploys a model $\mathcal{M}$. We consider software that is already executing within CVMs, as ML companies today are exploring use of CVMs [18, 9, 129], and major cloud service providers (e.g., Google Cloud and Microsoft Azure) support TEE-aware GPUs [55, 104]. We consider Intel TDX and NVIDIA H100 in this work, but our framework can be extended to any CVM or TEE-aware GPU (see Section 3.8).

Property attestations have the following components:

- ***Operation (op)***: an ML-related computational task over inputs ($\mathcal{I}$) to produce outputs ($\mathcal{O}$).
- ***Property***: an expression that describes the semantics of some meaningful relationship between *op*, $\mathcal{I}$, and $\mathcal{O}$.
- ***Attestation evidence***: a verifiable claim that aims to prove whether a property holds true.

We assume an attestation mechanism on Prv executes *op* and produces attestation evidence. Obtaining attestation evidence requires (1) measuring *op*, $\mathcal{I}$, $\mathcal{O}$ and (2) reporting those measurements by producing a cryptographic token (e.g., a digital signature) binding them. For the latter, a signing key is used that must be accessible only to the attestation mechanism.

3.3.2 Adversary Model

We assume an adversary ($\mathcal{A}$dv), in line with the common Intel TDX adversary model [5], who has control over the host machine. As a consequence, $\mathcal{A}$dv can access and manipulate any memory or device that is not within the trusted boundaries (e.g., other VMs, disk, I/O peripherals). The TDX module and TDX-aware CPU are trusted to be implemented correctly and provide their described functionality for inter-TD memory isolation, isolation from host VMM, and hardware-protected measurement and reporting (e.g., via a `QUOTE`) of TD images. Since TD creation is managed by $\mathcal{A}$dv-controlled VMM, TDs are not inherently trusted. Hence, TDX supports attestation via `QUOTE` (see Section 2.1).

We assume NVIDIA H100 GPU hardware is trusted, which measures and attests to its configuration (e.g., boot ROM, firmware, in confidential computing mode). By default, the interface between a GPU and the TD is not trusted; the GPU and all its memory are brought under the TD's protection only when the GPU is directly assigned by the VMM [80]. This includes mapping its communication buffers into TD shared memory. TD must obtain an attestation token from the GPU to determine that the assigned device has the anticipated configuration.

At run-time, $\mathcal{A}$dv may attempt to obtain confidential data or interfere with the execution of *op* in a TD. $\mathcal{A}$dv also may attempt to forge `QUOTE`s, either by learning the signing key or by tampering with relevant code/data without detection. We assume a Dolev-Yao $\mathcal{A}$dv as it relates to the network (i.e., $\mathcal{A}$dv carries the message). Therefore, $\mathcal{A}$dv may attempt to inject or modify messages sent/received by Prv. We consider denial-of-service attacks out of scope.

Side channels are prevalent and can be exploited to leak keys [99, 173, 141, 90, 106, 150]. To mitigate these attacks, additional mechanisms (e.g., constant-time software) can be added; as several works have studied side-channel mitigations [81, 82, 37, 170, 42], we consider these defenses as orthogonal to the focus of this paper. We also consider physical attacks on hardware (e.g., interposing memory buses [31] and or physically swapping GPUs) as out of scope.

3.3.3 Challenges and Requirements

Prior works [41, 58, 159, 140, 148] demonstrate methods to attest ML models using TEEs, but require operations to stay within the one TEE. For example, Laminator [41] requires that all computations stay inside an SGX enclave to ensure integrity. However, this is not possible with reasonable performance of LLM operations. For example, large datasets

may require datasets to reside in external storage during operation and require accelerators – e.g., GPUs or neural processing units (NPU) – for reasonable performance (recall Section 3.2.1). Although it is possible to outsource computations from a TEE to a standard GPU [168, 84], this adds significant overhead and would not preserve integrity of property measurements without additional measures. Furthermore, no prior work defines *properties* of generative models (e.g., from operations like fine-tuning, quantization, and chat sessions). Therefore, it is unclear how to perform property attestation that is relevant for Vrf while protecting Prv's confidential assets.

Due to these challenges, prior work cannot apply to properties of generative models, requiring another approach to address the following concerns:

1. accounting for random sampling of large datasets from (untrusted) external storage devices;
2. handling operations that depend on both CPU and GPU with reasonable computational overhead;
3. defining how to measure properties of generative models in a way that is relevant for Vrf and does not reveal Prv's confidential assets.

Based on these challenges, we consider the following requirements for designing PAL*M:

[**R1**] *Efficient*: low-overhead compared to confidential computing in the same environment;
[**R2**] *Secure*: a malicious Prv cannot generate false attestations.
[**R3**] *Scalable*: supports large numbers of provers/verifiers;
[**R4**] *Versatile*: supports a variety of properties; requires minimal effort for custom property measurement; can be ported to any CVM/TEE-based GPU configuration.

3.4 PAL*M Design

3.4.1 Overview of PAL*M

PAL*M enables Prv to attest properties of generative models through (1) design of a TD that effectively measures the operation corresponding to a particular property and (2) use of attestation support from Intel TDX and NVIDIA H100 to construct authenticated evidence of these measurements. Figure 3.2 shows a high-level overview of Prv's workflow with PAL*M and the interaction with Inr.

In ❶, Inr requests a property attestation by specifying: (a) the requested operation (*op*), (b) an attestation challenge (*Chal*), and any inputs that should be used (e.g., datasets, mod-

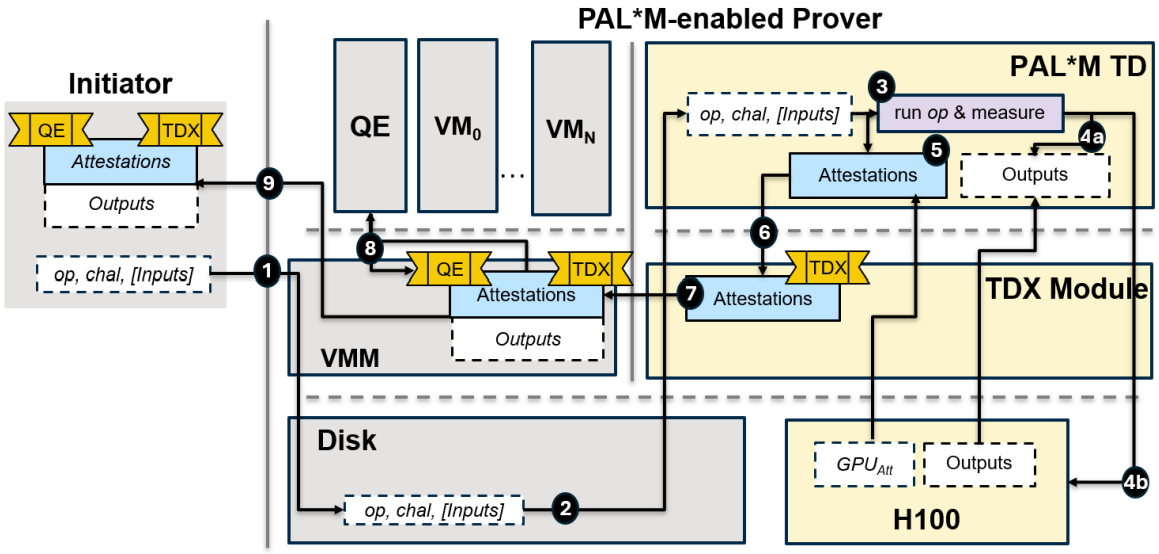


Figure 3.2: High-level overview of PAL*M-enabled property attestation. Inr interacts with untrusted components of Prv, which saves inputs to disk. PAL*M reads and measures inputs, runs the operations (including GPU), and measures all CPU and GPU outputs. All measurements are extended to `REPORTDATA`, which is used to as input to obtain `TDREPORT` from TDX Module. Finally, a `QUOTE` is generated by invoking the QE and returned to Inr.

els, configuration parameters, inference query). Upon receipt, Prv's VMM saves all data to disk. In ❷, inputs are fetched by PAL*M into TD memory according to the dataset access strategy (discussed further in Section 3.4.2). Based on *op*, the corresponding function will start to execute in ❸. While executing, outputs will be obtained by a combination of CPU execution ❹a and GPU execution ❹b. Operation outputs and attestation measurements are obtained ❺. When a GPU is used, the attestation measurements include a GPU-attestation token (GPU_{att}).

After outputs are obtained, PAL*M TD leverages TDX mechanisms to convert property attestation measurements into authenticated evidence. As a first step, it invokes the TDX module in ❻ to generate an authenticated `TDREPORT` containing PAL*M attestation measurements. To complete the quote process, `TDREPORT` is passed to the VMM in ❼, which invokes the QE in ❽ to convert `TDREPORT` into a `QUOTE`. Finally, `QUOTE` is returned to Inr in ❾. Inr will provide `QUOTE` and outputs to requesting Vrf, who consults Tru (e.g., Intel [76] and NVIDIA [126]) to perform evidence verification. Protocol details are described further in Section 3.4.4.

By design, PAL*M generates `QUOTE`s that attest to properties of generative model op-

erations, while addressing key challenges:

1. PAL*M leverages *incremental multiset hash functions* [32] to ensure large datasets are accurately reflected in attestation evidence despite residing in untrusted external storage devices and being randomly sampled;
2. PAL*M ensures measurements of both CPU and GPU configurations are captured in attestation evidence for properties that depend on CPU and GPU.

Building upon these capabilities, we show how to define, measure, and attest to properties of the following generative model operations (from Section 3.2.1):

1. dataset transformations and distributional statistics;
2. model pretraining, optimization (e.g., fine-tuning, quantization), and evaluation;
3. model interactions (e.g., inference or chat sessions).

3.4.2 Dealing with Large Datasets

Given the large size of LLM training datasets, ML software frameworks use two common approaches for efficient dataset handling: (1) loading the entire dataset *in-memory* or (2) accessing the dataset outside of TD protected memory with *memory mapping*. Although in-memory has an advantage for integrity measurements, frameworks such as Hugging Face adopt and default to using memory-mapped datasets to minimize the memory footprint of large datasets that cannot fit entirely into memory [66].

For accurate attestation, PAL*M must measure the dataset (or its items) upon use, since $\mathcal{A}$dv could tamper with data in external storage at any time (recall Section 3.3.2). Figure 3.3 shows how PAL*M handles this for *in-memory* and *memory-mapped* datasets.

Case 1: In-memory Dataset. For integrity measurement, this is the ideal approach when sufficient TD memory is available. Since the TD memory is protected, data within cannot be tampered. As such, the dataset can be measured once during loading, as long as it stays in memory for the entire operation. As shown in Figure 3.3 (left), our framework loads the entire dataset into memory, measures it, and uses it with the operation thereafter. Since one hash operation is required, the overhead is minimal (discussed further in Section 3.6).

Case 2: Memory-mapped Dataset. In contrast to the in-memory case, the dataset resides on the disk, and a virtual address mapping is established in memory (see Figure 3.3: right). As records are sampled, the OS reads just the requested items from disk into memory on demand. Since the dataset stays on disk, it is vulnerable to tampering at

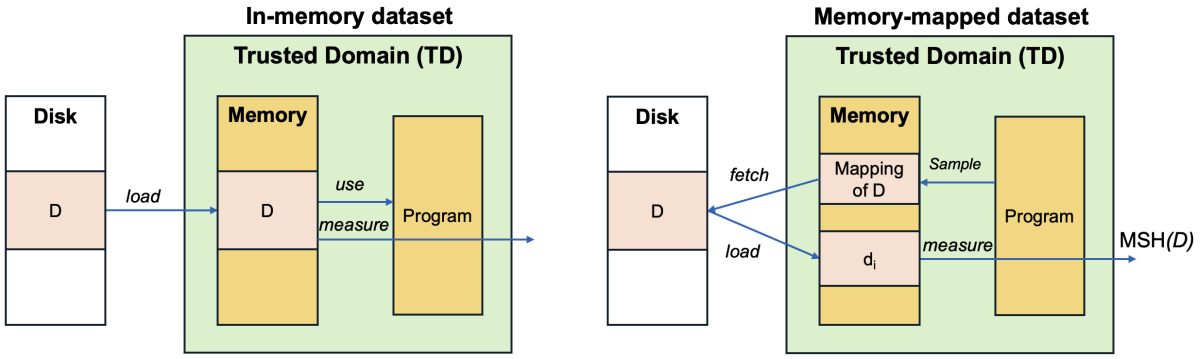


Figure 3.3: Depiction of PAL*M 's handling of *in-memory* (left) and *memory-mapped* (right) dataset. In the in-memory approach, the entire dataset is loaded and measured once at load time. In the memory-mapped approach, individual records are measured when accessed using a multiset hash to ensure consistent measurements regardless of sampling order.

run-time, resulting in inconsistencies between what is measured and what is used (i.e., TOCTOU attacks [17]).

To account for this, we measure each record at the time it is sampled, accumulating a measurement of the dataset in use. Since training algorithms often *randomly sample* records, a straightforward approach (e.g., a hash chain over accessed records) would be unsuitable, as different sampling orders would result in different measurements. Based on this insight, we employ an *incremental multiset hash* (`MSH`) [32].

`MSH` is a cryptographic primitive that produces a fixed-size hash over an unordered collection. It is suitable for settings in which the order of elements is irrelevant, but integrity and compactness are required. By accumulating an `MSH` over the sampled records, PAL*M accurately measures the dataset that is used at run-time. Additionally, this approach has low memory overhead since one multiset hash value is maintained.

However, hashing records alone is insufficient if $\mathcal{A}$dv can reorder them at run-time. Such reordering may affect model characteristics (e.g., fairness [49]) while the resulting `MSH` would be unchanged, since it is insensitive to sampling order. Hence, instead of accumulating a record (d_i) directly, we accumulate the concatenation of each record with its index ($d_i|i$). This approach makes it possible to detect any reordering by $\mathcal{A}$dv at run time, as the resulting `MSH` will differ due to an index mismatch with the requested index.

We use the `Mset-Mu-Hash` multiplicative multiset hash construction (from Theorem 2 in [32]) Let B denote a set of bit vectors of length m, M a multiset of elements of B, and

M_b be the number of times $b \in B$ is in the multiset M. Let q be a large prime power and consider computations in the field $GF(q)$. Then, let $\mathtt{h}$ be a poly-random function performing $\mathtt{h} : B \to GF(q)$. For `Mset-Mu-Hash`, let $\mathtt{h} : B \to \mathbb{Z}_n^l$, $n = 2^{\sqrt{m}}$, $l = \sqrt{m}$. The multiplicative multiset hash $\mathtt{MSH}(M)$ is defined as:

$$\mathtt{MSH}(M) = \prod_{b \in B} M_b \mathtt{h}(b) \mod n$$

`Mset-Mu-Hash` is multiset-collision resistant under the discrete logarithm assumption in the random oracle model (Theorem 2 and Appendix C of [32]), meaning it is computationally infeasible to find two distinct multisets that produce the same hash value. Compared to straightforward dataset hashing, this approach incurs additional overhead due to iterative hash and modular operations based on the hardness of discrete logarithm problem, requiring a large prime. However, it ensures no other countermeasures are required to account for random sampling, as any permutation of the full set of dataset read operations will result in the same final `MSH`. We note that we chose to use the `Mset-Mu-Hash` construction due to its simplicity and widely available library support, but any `MSH` construction could be used. For example, `MSH` construction based on elliptic curves [98] can be used for more efficiency gains, but have other limitations (further details in Section 3.8). Even with `Mset-Mu-Hash`, overhead of `MSH` is negligible compared to computation-heavy ML tasks (shown in Section 3.6).

It is possible that expected dataset measurement from Tru is not `MSH`. Consequently, when memory-mapped datasets are in use, PAL*M must also enable Vrf to trace back to original dataset measurements.

3.4.3 Property Measurements for Large Generative Models

Recall from Section 3.3.1 that obtaining property attestation evidence requires (1) obtaining measurements of $\{op, \mathcal{I}, \mathcal{O}\}$ and (2) reporting those measurements by producing an authentication token. In this section, we discuss the former.

We categorize properties measured by PAL*M as one of the following based on *op*: dataset property, model property, model interaction property. Based on this, we focus on the following components of property measurement (subset of those from Section 3.3.1):

- ***Input*** ($\mathcal{I}$): inputs (e.g., datasets, models, queries, or other parameters) used by *op*.
- ***Output*** ($\mathcal{O}$): outputs (e.g., prompt response, model, or dataset) produced by *op*.

Table 3.1: Notation Summary

Symbol	Definition
$\mathcal{I}$, $\mathcal{O}$, Att	Set of inputs, outputs, and attestation measurements
P_{dist}	Distributional property
`MSH`	Multiset hash function
`h`	Standard cryptographic hash function
$\texttt{h}_{\mathcal{D}}$	Dataset hash (either `MSH` or `h` based on memory management strategy from Sec. 3.4.2)
GPU_{att}	Attestation token from TEE GPU
$\mathcal{D}$	Dataset
$\mathcal{D}_{tr}$	Training dataset
$\mathcal{D}_{pre}$	Preprocessed dataset
$\mathcal{M}_{ar}$	Model architecture
$\mathcal{T}$	Training configuration
$\mathcal{M}_{tr}$	Pretrained model
$\mathcal{M}_{tok}$	Tokenizer
id_{opt}	Optimization type
$\mathcal{M}_{adp}$	Adapter model
$\mathcal{D}_{opt}$	Optimization dataset
$[\cdot]^*$	Optional components
$\mathcal{D}_{te}$	Test dataset
$metric$	Evaluation metric
q_i, r_i	The i-th inference query and response
$\mathcal{H}$	Session history (i.e., set of all query-response pairs $[(q_0, r_0), \ldots, (q_{i-1}, r_{i-1})]$) that are used as additional context to produce q_i for r_i

- ***Property***: an expression that describes semantics (either natural language or mathematical expression) of the relationship between $\{op, \mathcal{I}, \mathcal{O}\}$.
- ***Attestation Measurement (Att)***: Measurements that reflect the property under the adversary model (from Section 3.3.2).

A summary of frequent notation in this section is in Table 3.1.

GPU Attestation. For all operations that require GPU, Att must include GPU_{att} to reflect the complete environment (i.e., GPU driver, firmware/VBIOS, and other configurations) in which the computations were performed. Therefore, GPU_{att} is included as an optional input (denoted with $[\cdot]^*$) for all operations described in this section. Beyond obtaining GPU_{att}, a secure channel must be established prior to any operation. This is achieved through Security Protocol and Data Model (SPDM) session key exchange during the TD instantiation [127, 56], which cryptographically binds all subsequent run-time interactions to the attested GPU instance. The session establishment is also included in the CVM's attestation (i.e., the `QUOTE`). Together, GPU_{att} and SPDM session ensure that

the GPU in use is a legitimate confidential GPU and that it remains so throughout the operation.

Dataset Properties. We discuss dataset properties based on the following operations: dataset preprocessing, calculation of dataset attribute distribution, and dataset measurement binding.

Dataset preprocessing (from Section 3.2.1) may occur prior to training or inference. We define attestation measurement for preprocessing in Definition 1. For preprocessing, $\mathcal{I}$ contains an input dataset ($\mathcal{D}$) and $\mathcal{O}$ contains an output preprocessed dataset ($\mathcal{D}_{pre}$). A property measurement of preprocessing asserts that $\mathcal{D}_{pre}$ was the result of computing a preprocessing algorithm (PreProc) on $\mathcal{D}$. Finally, *Att* contains measurements of $\mathcal{D}$ and $\mathcal{D}_{pre}$ using the dataset hash function ($\mathtt{h}_{\mathcal{D}}$): either a standard cryptographic hash function ($\mathtt{h}$) or a $\mathtt{MSH}$, based on whether $\mathcal{D}$ and $\mathcal{D}_{pre}$ are in-memory or memory mapped, respectively.

Definition 1: Preprocessing
Input ($\mathcal{I}$): $\mathcal{D}$
Output ($\mathcal{O}$): $\mathcal{D}_{pre}$
Property: $\mathcal{D}_{pre} = \text{PreProc}(\mathcal{D})$
Att. Measurement (*Att*): $\mathtt{h}_{\mathcal{D}}(\mathcal{D})$, $\mathtt{h}_{\mathcal{D}}(\mathcal{D}_{pre})$, $[GPU_{att}]^*$

Dataset attribute distribution calculations [40] reflect a distribution of an attribute across a dataset (e.g., word lengths/frequency for LLMs). We define attestation measurement for calculation of attribute distribution in Definition 2. For this operation, $\mathcal{I}$ contains $\mathcal{D}$, and $\mathcal{O}$ contains an attribute distribution (A_{dist}). *Att*, containing $\mathtt{h}(A_{dist})$ and $\mathtt{h}_{\mathcal{D}}(\mathcal{D})$, asserts the property that A_{dist} was obtained by an algorithm (Dist) that computes $\mathcal{D}$'s attribute distribution.

Definition 2: Attribute Distribution
Input ($\mathcal{I}$): $\mathcal{D}$
Output ($\mathcal{O}$): A_{dist}
Property: $A_{dist} = \text{Dist}(\mathcal{D})$
Att. Measurement (*Att*): $\mathtt{h}(A_{dist})$, $\mathtt{h}_{\mathcal{D}}(\mathcal{D})$, $[GPU_{att}]^*$

It is possible that reference values used by Vrf are produced by $\mathtt{h}$ while an operation requires memory-mapping, resulting in a measurement based on $\mathtt{MSH}$. To account for this scenario, PAL*M supports measurement binding defined in Definition 3.

For measurement binding, $\mathcal{I}$ contains $\mathcal{D}$, and $\mathcal{O}$ contains the measurements of $\mathcal{D}$ that should be bound: $\mathtt{h}(\mathcal{D})$ and $\mathtt{MSH}(\mathcal{D})$. A property that $\mathtt{h}(\mathcal{D})$ and $\mathtt{MSH}(\mathcal{D})$ were produced from the same dataset is asserted through *Att* containing $\mathtt{h}(\mathtt{h}(\mathcal{D})||\mathtt{MSH}(\mathcal{D}))$.

Definition 3: Measurement Binding
Input ($\mathcal{I}$): $\mathcal{D}$,
Output ($\mathcal{O}$): $\mathtt{h}(\mathcal{D})$, $\mathtt{MSH}(\mathcal{D})$
Property: $\mathtt{h}(\mathcal{D})$ and $\mathtt{MSH}(\mathcal{D})$ were produced from $\mathcal{D}$
Att. Measurement (Att): $\mathtt{h}(\mathtt{h}(\mathcal{D})||\mathtt{MSH}(\mathcal{D}))$, $[GPU_{att}]^*$

Model Properties. We present model properties related to training and evaluation. Among model training operations, we consider model training itself and post-training model optimizations.

Training Properties. First, we consider training by building on prior work [41]. We define attestation measurements for training in Definition 4. For training, $\mathcal{I}$ includes the specified model architecture ($\mathcal{M}_{ar}$), training dataset ($\mathcal{D}_{tr}$), training configuration ($\mathcal{T}$), which specifies hyperparameters, and a tokenizer ($\mathcal{M}_{tok}$). Then, $\mathcal{O}$ contains the trained model ($\mathcal{M}_{tr}$). *Att* contains measurements of ($\mathcal{T}$, $\mathcal{M}_{ar}$, $\mathcal{M}_{tr}$, $\mathcal{M}_{tok}$) using $\mathtt{h}$ and $\mathcal{D}$ using $\mathtt{h}_{\mathcal{D}}$, asserting a property describing that $\mathcal{M}_{tr}$ was obtained by a training algorithm (Train) taking $\mathcal{D}_{tr}$, $\mathcal{T}$, $\mathcal{M}_{ar}$, $\mathcal{M}_{tok}$.

Definition 4: Training
Input ($\mathcal{I}$): $\mathcal{M}_{ar}$, $\mathcal{D}_{tr}$, $\mathcal{T}$, $\mathcal{M}_{tok}$
Output ($\mathcal{O}$): $\mathcal{M}_{tr}$
Property: $\mathcal{M}_{tr} = \text{Train}(\mathcal{D}_{tr}, \mathcal{T}, \mathcal{M}_{ar})$
Att. Measurement (Att): $\mathtt{h}(\mathcal{M}_{tr})$, $\mathtt{h}(\mathcal{M}_{ar})$, $\mathtt{h}_{\mathcal{D}}(\mathcal{D}_{tr})$, $\mathtt{h}(\mathcal{T})$, $\mathtt{h}(\mathcal{M}_{tok})$, $[GPU_{att}]^*$

Definition 5 defines attestation measurements for post-training model optimizations (e.g., fine-tuning, quantization, pruning). For model optimization, $\mathcal{I}$ contains the following required inputs: $\mathcal{M}$, $\mathcal{T}$, $\mathcal{M}_{tok}$, and the optimization type (id_{opt}). Optional components of $\mathcal{I}$ include a base adapter layer ($\mathcal{M}_{adp}$) and additional datasets ($\mathcal{D}_{opt}$). For this operation, $\mathcal{O}$ contains the optimized model ($\mathcal{M}_{opt}$). *Att* contains all components of $\mathcal{I}$ and $\mathcal{O}$ using $\mathtt{h}$ and $\mathtt{h}_{\mathcal{D}}$, reflecting its assertion that:

- Opt is selected from a set of known optimization functions ($\mathcal{F}_{opt}$) based on id_{opt};
- $\mathcal{M}_{opt}$ is the result of Opt taking all components of $\mathcal{I}$.

Evaluation Properties. As described in Section 3.2.1, generative models have various benchmarks to produce metrics reflecting a model's effectiveness according to a particular criteria. Attestation measurement for model evaluation is defined in Definition 6. For model evaluation, $\mathcal{I}$ includes $\mathcal{M}$, $\mathcal{M}_{tok}$, and a test dataset ($\mathcal{D}_{te}$), while $\mathcal{O}$ contains the evaluation *metric*. Measurements of $\mathcal{M}$ and $\mathcal{M}_{tok}$ using $\mathtt{h}$ and $\mathcal{D}_{te}$ using $\mathtt{h}_{\mathcal{D}}$ in *Att* assert

Definition 5: Weight Optimizations
Input ($\mathcal{I}$): $\mathcal{M}$, $\mathcal{M}_{tok}$, $\mathcal{T}$, id_{opt}, $[\mathcal{M}_{adp}, \mathcal{D}_{opt}]^*$
Outputs ($\mathcal{O}$): $\mathcal{M}_{opt}$
Property: Opt $= \mathcal{F}_{opts}[id_{opt}]$
$\mathcal{M}_{opt} = \text{Opt}(\mathcal{M}, \mathcal{M}_{tok}, \mathcal{T}, [\mathcal{M}_{adp}, \mathcal{D}_{opt}]^*)$
Att. Measurement (*Att*)**:** $\texttt{h}(\mathcal{M}_{opt})$, $\texttt{h}(\mathcal{M})$, $\texttt{h}(\mathcal{M}_{tok})$, $\texttt{h}(\mathcal{T})$, $\texttt{h}(id_{opt})$, $[GPU_{att}, \texttt{h}(\mathcal{M}_{adp}), \texttt{h}_{\mathcal{D}}(\mathcal{D}_{opt})]^*$

a property about the evaluation: that *metric* was produced by an evaluation operation (Eval) taking $\mathcal{M}$, $\mathcal{M}_{tok}$, and $\mathcal{D}_{te}$.

Definition 6: Evaluation
Input ($\mathcal{I}$): $\mathcal{M}$, $\mathcal{M}_{tok}$, $\mathcal{D}_{te}$
Output ($\mathcal{O}$): *metric*
Property: $metric \leftarrow \text{Eval}(\mathcal{M}, \mathcal{M}_{tok}, \mathcal{D}_{te})$
Att. Measurement (*Att*): $\texttt{h}(\mathcal{M}), \texttt{h}(\mathcal{M}_{tok})$, $\texttt{h}_{\mathcal{D}}(\mathcal{D}_{te})$, $\texttt{h}(metric)$, $[GPU_{att}]^*$

Inference-time Properties. We consider two cases of inference-time properties which includes interactions between client and model: one prompt-response inference and a *chat session* of multiple prompt-response inference pairs.

Attestation measurements for a single inference are defined in Definition 7. An inference query (q) is one component of $\mathcal{I}$ along with $\mathcal{M}$ and $\mathcal{M}_{tok}$, while the $\mathcal{O}$ contains the inference response (r). Measurements of q, $\mathcal{M}$, $\mathcal{M}_{tok}$, and r using $\texttt{h}$ assert the property that q was obtained by passing r first through $\mathcal{M}_{tok}$ and then $\mathcal{M}$.

Definition 7: Single Inference
Input ($\mathcal{I}$)**:** q, $\mathcal{M}$, $\mathcal{M}_{tok}$
Output ($\mathcal{O}$)**:** r
Property: $r = \mathcal{M}(\mathcal{M}_{tok}(q))$
Att. Measurement (*Att*)**:** $\texttt{h}(q)$, $\texttt{h}(\mathcal{M})$, $\texttt{h}(\mathcal{M}_{tok})$, $\texttt{h}(r)$, $[GPU_{att}]^*$

Attestation measurements reflecting the influence of a chat session on a query-response (i.e., a *session inference*) are defined in Definition 8. For a session inference operation, $\mathcal{I}$ contains the same data as the single inference operation, while $\mathcal{O}$ contains r and the chat session history ($\mathcal{H}$): a sequential list of all (q_i, r_i) pairs of the current chat session, including (q, r). Measurements of r, $\mathcal{H}$, q, $\mathcal{M}_{tok}$, and $\mathcal{M}$ using $\texttt{h}$ assert the property that

Table 3.2: Operations and property attestations in PAL*M

Operation (*op*)	**Property Attestation**
Preprocessing	Proof of preprocessing
Attribute distribution	Proof of attribute distribution
Measurement binding	Proof of binding
Training	Proof of training
Weight optimization	Proof of optimization
Fine-tuning	Proof of fine-tuning
Quantization	Proof of quantization
Evaluation	Proof of evaluation
Inference	Proof of inference
Chat session inference	Proof of session inference

r was obtained from passing q prepended with $\mathcal{H}$ without (q, r) through $\mathcal{M}_{tok}$, followed by passing that result through $\mathcal{M}$.

Definition 8: Session Inference
Input ($\mathcal{I}$): q, $\mathcal{M}_{tok}$, $\mathcal{M}$
Output ($\mathcal{O}$): r, $\mathcal{H}$
Property: $r = \mathcal{M}(\mathcal{M}_{tok}(\mathcal{H} \setminus [q, r] \mid\mid q))$
Att. Measurement (Att): $\mathtt{h}(q)$, $\mathtt{h}(\mathcal{M}_{tok})$, $\mathtt{h}(\mathcal{M})$, $\mathtt{h}(r)$, $\mathtt{h}(\mathcal{H})$, $[GPU_{att}]^*$

3.4.4 From Measurement to Attestation

For each *op*, the corresponding components of *Att*, as defined in Section 3.4.3, are used to construct property attestation evidence. We refer to each property attestation as *proof of* $< op >$. Table 3.2 shows operations and resulting property attestations. In this section, we describe how PAL*M interacts with other system components and leverages Intel TDX features to respond to Inr requests with property attestation evidence.

Figure 3.4 shows the property attestation protocol enabled by PAL*M. Here, we describe the interactions between Inr and Prv containing a VMM (including QE), TDX module, and PAL*M-based TD. We note there are several possible interactions between Inr and Vrf

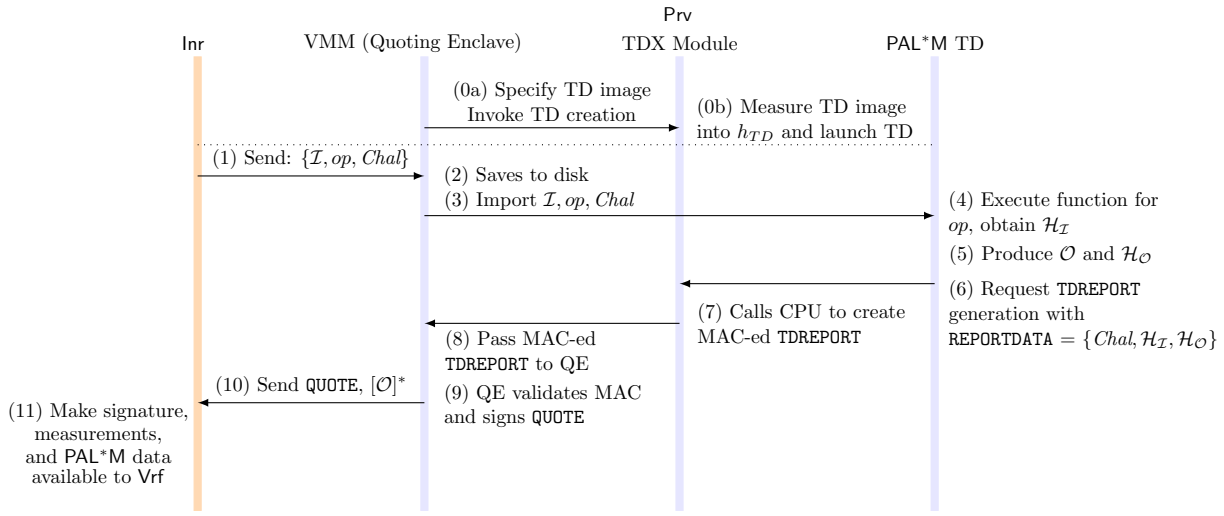


Figure 3.4: PAL*M Property Attestation Protocol between Inr and Prv. Inr sends request to Prv containing untrusted VMM, QE, trusted TDX Module, and attested PAL*M TD. Prv responds after performing the requested operation, measuring relevant inputs and ouputs, and generating both a TDREPORT and QUOTE.

(e.g., Inr publishes attestations to a *transparency log* [58, 159]). We discuss implications of alternative strategies in Section 3.8.

In an offline phase, Prv initializes and measures the PAL*M TD. In step (0a), VMM specifies the TD image that should be used. In step (0b), the Intel TDX architecture and TDX module measure the TD image into h_{TD} and launch it.

After initialization, the online protocol can begin. In step (1), Inr sends a property attestation request to Prv containing op, an attestation challenge ($Chal$), and a set of input assets ($\mathcal{I}$). For large input assets (e.g., datasets or models), Inr's request may specify repositories from which Prv should download them. Or, Inr's request may specify that Prv use a dataset/model owned by Prv. Regardless of these specificities, Prv saves all data from the request to disk in step (2) upon receipt. In step (3), the PAL*M TD imports request data and all data required for op into its memory. Datasets are imported depending on whether Prv is using a memory-mapped or an in-memory dataset (from Section 3.4.2).

PAL*M TD reads op to execute in step (4). During execution, it obtains a set of input measurements ($\mathcal{H}_\mathcal{I}$). After completing the operation, PAL*M will obtain a set of outputs ($\mathcal{O}$) and corresponding measurements ($\mathcal{H}_\mathcal{O}$) in step (5). $\mathcal{H}_\mathcal{I}$ and $\mathcal{H}_\mathcal{O}$ correspond to components of Att for each op described in Section 3.4.3.

Next, PAL*M prepares the request for TDREPORT generation in step (6). To do so, it assigns REPORTDATA input used in TDREPORT generation (recall Section 2.1) as the concatenation of the following data: $\{op, Chal, \mathcal{H}_{\mathcal{I}}, \mathcal{H}_{\mathcal{O}}\}$. Upon receiving REPORTDATA, the TDX module will create a TDREPORT in step (7) by invoking the CPU. During this step, the CPU will compute a MAC over h_{TD}, measurements of the TDX module, and all data extended into REPORTDATA by PAL*M. In step (8), the MAC-ed TDREPORT is passed to VMM, who passes it to QE in step (9). Upon receipt, QE:

- verifies the MAC on TDREPORT is valid, to ensure VMM did not tamper with TDREPORT it received from the TDX Module;
- uses TDREPORT to produce the QUOTE containing all measurements from TDREPORT plus a signature (sig).

In step (10), Prv sends QUOTE and optionally $\mathcal{O}$ (if it is not confidential) to Inr, who sends them to Vrf in step (11).

Upon receiving QUOTE and (optionally $\mathcal{O}$) from Inr, Vrf parses the response to perform verification. First, it checks sig to determine whether QUOTE came from Prv. Next, it checks $Chal$ to determine the freshness of the response. The type of $Chal$ that is effective depends on the operation (discussed further in Section 3.8).

Assuming both checks pass, Vrf continues to inspect the measurements of the TD image, TDX module image, and GPU_{att} (by obtaining reference values from Intel [76] and NVIDIA [126]). Assuming this passes, Vrf continues to inspecting $\mathcal{H}_{\mathcal{I}}$, $\mathcal{H}_{\mathcal{O}}$, and $\mathcal{O}$. If $\mathcal{O}$ was sent to Vrf, it checks whether $\mathcal{H}_{\mathcal{O}}$ corresponds to $\mathcal{O}$. Finally, Vrf inspects $\mathcal{H}_{\mathcal{I}}$ and $\mathcal{H}_{\mathcal{O}}$, consulting reference values from Tru as needed, to determine if the desired property related to op is upheld.

Like prior work [41], property attestations introduced in this work can be used to construct chains of attestations, enabling verifiers to derive composite guarantees spanning the full ML life-cycle. For example, combining a proof of fine-tuning (attesting that $\mathcal{M}_{opt}$ was fine-tuned on $\mathcal{D}_{opt}$ that was preprocessed with $PreProc(\mathcal{D}_{opt})$ algorithm) with proof of inference (attesting that $\mathcal{M}_{opt}$ produces $\mathcal{O}$ with an inference query q).

3.4.5 Trusted Authority, Reference Values, Verification

Role of the Trusted Authority. The role of Tru depends on what is being verified. For hardware and platform state, Intel and NVIDIA naturally act as Tru by publishing reference measurements of firmware and platform state (e.g., through NVIDIA's NRAS [126] and Intel's Trust Authority [76]). For public models and datasets, public registries such

as Hugging Face can act as Tru by hosting reference values that Vrf can independently retrieve and compare with the attested values. For ML properties, Vrf determines whether a property holds by combining the TEE attestation (which confirms that trusted measurement code ran correctly inside the TEE) with reference values from Tru (e.g., expected MSH values for a dataset or model). Neither source alone is sufficient: the TEE attestation provides integrity of the measurement process, while Tru provides the expected values against which the measurement is compared.

Confidential Models and Datasets. When both dataset and model are confidential, no natural Tru exists since Vrf cannot independently derive or check reference values. Two mechanisms can address this. First, attestation evidence (e.g., proof of training) can be published to a transparency log, recording that a confidential model with a certain hash was trained on a dataset with a certain hash. This evidence can then be chained with subsequent attestations, for example, proof of evaluation of the same model on a benchmark (e.g., the model achieves a certain MMLU accuracy), allowing Vrf to trace the model's full history without ever seeing the underlying data or model. Second, a new Tru can act as a traditional auditor or regulator that holds confidential reference values or independently verifies the authenticity of confidential datasets and models, confirming to the Vrf whether attested values match without revealing the data itself.

Non-deterministic Nature of Training. PAL*M's MSH captures the dataset accesses regardless of the sampling method (e.g., batching, order), always producing the same unique MSH for the same dataset. Since reference values are defined per artifact (dataset and model), training nondeterminism does not invalidate commitments to the inputs and outputs. For example, in proof of training, Vrf verifies that attested code processed a dataset with a certain MSH value and produced a model, rather than reproducing the exact sequence of gradient updates. For multi-epoch training, Vrf can derive the expected MSH directly from the single-epoch dataset MSH and the configured number of epochs. Since each epoch processes the same dataset, the effect of multiple epochs is equivalent to repeating the single-epoch MSH composition the corresponding number of times. Consequently, only the single-epoch dataset MSH needs to be published. Note that, because MSH requires every record in the dataset to be processed exactly once, training must use an integer number of complete epochs.

3.5 Experiment Setup

3.5.1 Datasets, Models, and Metrics

We use the following datasets in our experiments:

- BookCorpus [180]: a collection of 74 million unpublished novels. We use the full BookCorpus dataset for proof of training and attribute distribution, while for proof of preprocessing and proof of binding, we use a 1/100 subset to reduce experiment running time. For training, we preprocess the full BookCorpus dataset by tokenizing and concatenating text into sequences of 1024 tokens.
- yahma/alpaca-cleaned [163]: a instruction-following dataset containing over 52,000 instruction-response pairs aimed at improving model alignment. We use this dataset for proof of fine-tuning to provide a dataset variety other than BookCorpus.
- MMLU [60]: a multiple-choice question-and-answer (Q&A) dataset covering 57 fields. It is widely used to assess general-purpose intelligence in language models. We use all 57 fields for our proof of evaluation experiments.
- WMT14(DE - EN) [14]: a test set of WMT14 German-English to evaluate LLM translation capability. This dataset is used to calculate the BLEU score [133] for our proof of evaluation experiment.
- CoQA [142]: a conversational Q&A dataset containing stories with context-dependent Q&A. For proof of inference, we select the first question of the first ten records. For the proof of session inference, we use all ten records and select the first five questions from each, simulating a session with 50 prompts.

For models, we use GPT-2 [139] (124M parameters; 0.46 GB) for proof of training. For fine-tuning, inference, and evaluation, we use three popular instruction-tuned LLMs:

- Llama-3.1-8B [165] (8B parameters, 15 GB),
- Gemma-3-4B [164] (4B parameters, 8 GB), and
- Phi-4-Mini [3] (3.8B parameters; 7.15 GB).

We choose Llama-3.1-8B for its 8 billion parameters, while both Gemma-3-4B and Phi-4-Mini have ≈4 billion parameters, allowing us to show how the overhead of PAL*M varies across different model sizes and architectures. To demonstrate a realistic fine-tuning workflow, we fine-tune all three models at full precision using LoRA [62]. For quantization, we focus on Llama-3.1-8B.

We use the following metrics to capture both the overall overhead from PAL*M and the main sources of overhead:

- **Total Time:** the total execution time spent on the task.
- **Baseline:** the execution time on a TDX-enabled machine without measurement from PAL*M. We omit a non-TDX baseline, as prior work has evaluated the overhead of TDX itself [30].
- **Memory Usage:** total memory usage by the dataset.
- **Raw Measurement Time:** time spent purely hashing inputs and outputs (with `h` or `MSH`).
- **Attestation:** The time spent by Intel TDX and NVIDIA H100 to produce `QUOTE` and GPU_{att}, respectively.
- **Raw Measurement Overhead (%):** The total raw measurement overhead from PAL*M as a percentage of the total time.
- **Attestation Overhead (%):** The total attestation overhead from PAL*M as a percentage of the total time.
- **Total Observed Overhead (%):** The overall observed overhead as a percentage when comparing the total execution time of PAL*M to the baseline.

3.5.2 Prototype Configuration

The prototype of PAL*M's TD is configured with 32 vCPUs and 128 GB of memory, running Ubuntu 24.04.1 LTS with kernel version 6.8.0-86-generic [21]. The host has an Intel Xeon Silver 4514Y CPU, 512 GB of random access memory (RAM), and an NVIDIA H100 NVL 94 GB GPU with confidential compute mode enabled [124]. We assign the GPU directly to the TD using Direct Device Assignment to give the TD exclusive access. We run each experiment five times and report the average. Standard deviation is excluded as it was negligible.

PAL*M is measurement/attestation is implemented in Python. Attestation uses Intel Data Center Attestation Primitives (DCAP) [147] and NVIDIA Remote Attestation Service (NRAS) [126]. PAL*M is built on PyTorch and the Hugging Face Datasets framework, allowing access to dataset and model internals for measurement. We use SHA-256 as the underlying hash function for normal hashing and `MSH`. We use a 2048-bit prime for `MSH`. For attestation, we use Intel DCAP to generate `QUOTE` and NRAS to verify GPU_{att}. We also use the following operation-specific frameworks for the experiments:

- **Unsloth** [38]: an open-source framework for optimized LLM fine-tuning. We use it for proof of fine-tuning experiment.
- **AutoAWQ framework** [59]: implements the Activation-aware Weight Quantization (AWQ) algorithm for quantizing LLMs. We use it for proof of quantization experiment.

Table 3.3: Performance for proofs of dataset attribute distribution and preprocessing using BookCorpus (Full) and BookCorpus (1/100).

Proof of	Attribute Distribution		Preprocessing	
Dataset	BookCorpus (Full)		BookCorpus (1/100)	
Memory Variant	In-memory	Memory-mapped	In-memory	Memory-mapped
Total Time	339.16 min	576.97 min	237.70 s	378.62 s
Baseline	339.09 min	331.73 min	236.66 s	236.12 s
Mem Usage (GB)	5.47	4.33	0.12	0.04
Raw Meas.	3.06 s	392.68 min	0.04 s	236.06 s
Attestation	1.06 s	0.80 s	0.90 s	0.79 s
Raw Meas. Overhead (%)	0.015	68.06	0.06	62.35
Att. Overhead (%)	0.0052	0.0023	0.38	0.21
Total Observed Overhead (%)	0.021	42.50	0.44	37.64

- **lm-evaluation-harness framework** [51]: standardized toolkit that runs LLMs on multiple benchmark tasks through a unified API to produce comparable scores. We use it for proof of evaluation, specifically evaluating the model with MMLU.

3.6 Evaluation

3.6.1 Performance Analysis

Tables 3.3–3.7 show the timing results for the performance evaluations. The cells in gray represent the baseline, PAL*M's overhead is highlighted in blue, and the total times are in orange. We parallelize the dataset lookup across 8 workers for dataset-related attestations requiring manual dataset iteration.

Dataset Attestations. Table 3.3 presents the results for proof of attribute distribution and proof of preprocessing. Both operations process the dataset using 8 parallel workers (i.e., vCPU cores). In the memory-mapped setting, parallel I/O can scale better with the memory-mapped approach, as all processes can directly access the same underlying memory pages without duplication.

As a result, the raw overhead (i.e., MSH overhead per core) differs from the observed end-to-end overhead of the parallelized operation. The latter can be lower due to DataLoader

Table 3.4: Performance for proof of training with GPT-2 on preprocessed BookCorpus.

Dataset	BookCorpus (Full)	
Memory Variant	**In-memory**	**Memory-mapped**
Total Time	401.65 min	425.03 min
Baseline	401.60 min	401.20 min
Mem Usage (GB)	2.74	0.04
Raw Meas.	2.27 s	23.86 min
Attestation	1.11 s	0.96 s
Raw Meas. Overhead (%)	0.009	5.47
Att. Overhead (%)	0.005	0.006
Total Observed Overhead (%)	0.014	5.61

prefetching [136] or higher due to pipeline delays [10, 176]. For example, raw overhead from MSH operations of 62.35–68.06% results in 37.64–42.50% total observed overhead in the end-to-end parallelized operation.

Despite the reduction in this case, the observed overhead remains high because MSH requires modular exponentiation and multiplication over a 2048-bit prime for each record. Since dataset-level operations are inexpensive on per-record basis compared to others (e.g., fine-tuning, evaluation), even modest MSH costs dominate the run-time. We analyze this effect further in Section 3.6.1. Fortunately, these costs can be amortized across future dataset use. For example:

- proof of attribute distribution and preprocessing apply to all future uses of a dataset;
- proof of binding applies to any future dataset MSH.

For the 1/100 subset of BookCorpus dataset used in our evaluation, proof of binding takes 337.5 seconds (3.65 ms per record).

Model Attestations. In contrast to dataset attestations, the measurement overhead of PAL*M for proof of training is low. For the memory-mapped case, the raw measurement overhead is only 5.47% of the total time, whereas for the in-memory case, it is 0.009%; this results in an observed overhead of 5.61% for memory-mapped and 0.014% for in-memory case (Table 3.4). LLM training is computationally expensive; hence, the measurement time introduces a proportionally smaller overhead.

For model optimization, proof of fine-tuning (Table 3.5) also shows minimal overhead

Table 3.5: Performance for proof of model optimization (fine-tuning and quantization).

Proof of	Fine-tuning						Quantization
Model	Llama-3.1-8B		Gemma-3-4B		Phi-4-Mini		Llama-3.1-8B
Memory Variant	In-mem.	Mem.-mapped	In-mem.	Mem.-mapped	In-mem.	Mem.-mapped	N/A
Total Time	268.81 min	261.74 min	325.61 min	369.56 min	169.93 min	170.48 min	812.92 s
Baseline	268.30 min	258.07 min	325.29 min	361.82 min	169.65 min	168.16 min	773.84 s
Mem Usage (GB)	0.085	0.004	0.087	0.004	0.088	0.004	N/A
Raw Meas.	28.76 s	132.87 s	17.93 s	129.40 s	15.83 s	127.99 s	28.16 s
Attestation	1.11 s	1.44 s	1.00 s	1.20 s	1.00 s	0.84 s	0.92 s
Raw Meas. (%)	0.18	0.85	0.09	0.58	0.16	1.20	4.70
Att. (%)	0.007	0.009	0.005	0.005	0.010	0.008	0.117
Total Observed (%)	0.19	1.40	0.10	2.09	0.17	2.27	4.81

Table 3.6: Performance for proof of evaluation based on MMLU and BLEU score with WMT14(DE-EN) dataset.

Model	Llama-3.1-8B				Gemma-3-4B				Phi-4-Mini			
Memory Variant	In-mem.		Mem.-mapped		In-mem.		Mem.-mapped		In-mem.		Mem.-mapped	
Metric	MMLU	BLEU	MMLU	BLEU	MMLU	BLEU	MMLU	BLEU	MMLU	BLEU	MMLU	BLEU
Total Time	7.93 min	121.71 min	8.78 min	121.55 min	13.27 min	242.23 min	14.41 min	241.53 min	8.47 min	119.03 min	9.39 min	118.65 min
Baseline	7.73 min	121.23 min	7.80 min	120.94 min	13.15 min	241.92 min	13.30 min	241.10 min	8.35 min	118.75 min	8.43 min	118.26 min
Mem Usage (GB)	0.028	0.001	0.013	0.001	0.022	0.001	0.008	0.001	0.025	0.001	0.009	0.001
Raw Meas.	10.13 s	28.16 s	74.59 s	35.57 s	6.33 s	17.32 s	70.72 s	24.55 s	5.66 s	15.65 s	70.06 s	22.50 s
Attestation	1.58 s	1.10 s	0.85 s	1.07 s	1.09 s	1.12 s	0.87 s	1.23 s	1.35 s	1.08 s	0.92 s	1.13 s
Raw Meas. (%)	2.13	0.39	14.16	0.49	0.79	0.12	8.18	0.17	1.11	0.22	12.43	0.31
Att. (%)	0.33	0.015	0.16	0.015	0.13	0.008	0.10	0.009	0.27	0.015	0.16	0.016
Total Observed (%)	2.46	0.40	11.24	0.50	0.93	0.13	7.69	0.17	1.38	0.23	10.28	0.31

for both raw and observed of $\leq$2.27% across all models, plus a substantial reduction in memory usage from 85-87 MB for the in-memory dataset to only 4 MB for the memory-mapped dataset. For proof of quantization, since no dataset is involved, the 4.7% raw measurement overhead comes from measuring models.

As shown in Table 3.6, the observed overhead of proof of evaluation with MMLU is dominated by the measurement overhead; the observed overheads are 0.93-2.46% for the in-memory case and 7.69-11.24% for the memory-mapped case (Table 3.6). For BLEU score, since dataset access is limited (WMT14(DE-EN) contains only 3003 records), the raw overhead of performing MSH in the memory-mapped case is comparable to hashing the dataset in the in-memory case, with both the raw and observed overheads remaining $\leq$ 0.5% across all models. Unlike dataset operations, total observed overhead for model operations is often

Table 3.7: Performance of proof of inference for one query-response and proof of session inference for 50 query-responses.

Model	Llama-3.1-8B		Gemma-3-4B		Phi-4-Mini	
Proof of Inference Type	**Single**	**Session**	**Single**	**Session**	**Single**	**Session**
Total Time	43.31 s	256.19 s	40.55 s	489.69 s	29.94 s	249.37 s
Baseline	14.54 s	227.02 s	22.09 s	471.22 s	13.40 s	232.70 s
Mem Usage (GB)	0.001	0.001	0.001	0.001	0.001	0.001
Raw Meas.	27.17 s	28.25 s	16.27 s	17.47 s	14.59 s	15.66 s
Attestation	0.75 s	0.92 s	0.79 s	0.99 s	0.71 s	1.01 s
Raw Meas. Overhead (%)	64.34	11.03	43.28	3.57	52.16	6.28
Att. Overhead (%)	2.09	0.36	2.24	0.20	3.10	0.41
Total Observed Overhead (%)	66.43	11.39	45.52	3.77	55.26	6.69

higher than raw measurement overhead, likely due to pipeline stalls when transferring data to GPU.

Inference Attestations. Table 3.7 presents the results for both proof of inference and proof of session inference. Proof of inference follows the same pattern observed in dataset attestations, where the overhead appears large (64.34% for `Llama-3.1-8B`, 43.28% for `Gemma-3-4B`, and 52.16% for `Phi-4-Mini`) because the underlying computation is relatively small. This represents an uncommon case in which a proof is generated for only one prompt. On the other hand, proof of session inference simulates a more realistic user-model interaction. In this setting, the measurement overhead is significantly smaller: 11.03% for `Llama-3.1-8B`, 3.57% for `Gemma-3-4B`, and 6.28% for `Phi-4-Mini`.

Measurement overhead of memory-mapped operations. Table 3.8 compares per-record computation time with the `MSH` construction time (i.e., input measurement) for operations that use a memory-mapped dataset. Normalizing per record shows that dataset-related operations are much more lightweight than model-related operations. In contrast, input measurement time per record remains nearly constant (≈2.29–2.57 ms), as it is independent of the operation type. Consequently, relative measurement overhead is higher for operations with lower computational cost, despite the near-constant per-record measurement time. Due to the low overhead for common operations (training, fine-tuning, quantization, evaluation, session inference), PAL*M satisfies **R1**.

Table 3.8: Records per workers (R/W), compute time per record (C/R), and raw input measurement time per record (I/R) for memory-mapped operations. I/R remains roughly constant across operations, while C/R varies significantly, indicating that raw measurement overhead is higher when C/R is low.

Operation	Model	R/W	C/R (ms)	I/R (ms)	Raw Meas. Overhead (%)
Training	GPT2	572.2k	43.07	2.44	5.47
Fine-tuning	Llama-3.1-8B	51.8k	311.1	2.36	0.84
Fine-tuning	Gemma-3-4B	51.8k	376.3	2.38	0.58
Fine-tuning	Phi-4-Mini	51.8k	196.9	2.36	1.20
Eval. (MMLU)	Llama-3.1-8B	28.1k	40.5	2.29	14.15
Eval. (MMLU)	Gemma-3-4B	28.1k	33.8	2.29	8.17
Eval. (MMLU)	Phi-4-Mini	28.1k	27.4	2.29	12.43
Eval. (BLEU)	Llama-3.1-8B	3.0k	2416	2.34	0.45
Eval. (BLEU)	Gemma-3-4B	3.0k	4817.5	2.35	0.14
Eval. (BLEU)	Phi-4-Mini	3.0k	2363.7	2.34	0.30
Attr. Dist.	N/A	9.25M	3.75	2.54	68.06
Preprocessing	N/A	92.5k	4.09	2.57	62.35

3.6.2 Security Analysis

PAL*M aims to uphold integrity of its property attestation evidence in the presence of $\mathcal{A}\mathsf{dv}$, as defined in Section 3.3, who controls the host machine, including the VMM and disk. Based on this, the following security properties are desired:

- **[P1]** integrity of the operation (the set of $\mathcal{H}_{\mathcal{I}}$ and $\mathcal{H}_{\mathcal{O}}$ correspond to the expected $\mathcal{I}, \mathcal{O}$, according to the requested *op*).
- **[P2]** integrity of environment and responses (h_{TD}, *op*, $\mathcal{H}_{\mathcal{I}}$, and $\mathcal{H}_{\mathcal{O}}$ in `QUOTE` correspond to the expected TD image, $\mathcal{I}$, op, and $\mathcal{O}$, respectively);
- **[P3]** authenticity of attestation responses (any `QUOTE` that passes verification was produced through the expected QE, and `TDREPORT` was not forged); and

In order to violate these security properties, $\mathcal{A}\mathsf{dv}$ attempts to (1) tamper with any operation inputs; (2) tamper with, discard, or replay `TDREPORT` or `QUOTE` at any stage of their construction or transmission; or (3) tamper with the configuration of PAL*M TD.

To verify that the property attestation protocol from Section 3.4.4 upholds **[P1-3]**, we use the Tamarin Prover symbolic verification tool [103] to model the protocol depicted in Figure 3.4, to construct a set of security goals that subsume **[P1-3]**, and to verify that the security goals are upheld. Here, we describe the rationale for why they are upheld, with complete details in Section 3.7.

$\mathcal{A}$dv may tamper with operation inputs to violate any of these properties. For example, $\mathcal{A}$dv may manipulate *op* to change the requested function or misconfigure assets in $\mathcal{I}$ used for the requested function. Both cases are detectable by Vrf, since PAL*M does not measure either until they are loaded into TD memory. Therefore, Vrf is assured that the inputs reflected in the QUOTE were used in the process. In the case of a memory-mapped data structure, $\mathcal{A}$dv may manipulate data before it is read into the TD memory. For example, they may aim to repeat, replace, or reorder data records during fine-tuning to poison or bias the resulting model. These attempts can be detected by Vrf, since PAL*M measures a MSH of records and indices as they are sampled. We use Lemmas 1, 3, 6, 8, 9, 10, 11, and 12 to verify the properties are upheld despite these attempts.

$\mathcal{A}$dv may attempt to generate false outputs and corresponding measurements in an attempt to create a faulty TDREPORT before it is passed to the QE. PAL*M relies on the standard TDX attestation guarantees discussed in Chapter 2 that (a) TDREPORT authenticity is protected by MAC under a hardware-protected key inaccessible to software [5] and (b) the QE verifies this MAC prior to the QUOTE generation. Consequently, $\mathcal{A}$dv cannot forge a valid TDREPORT without access to the device key, and QUOTE forgery is infeasible because each QUOTE is signed using a provisioned attestation key whose private material never leaves QE. PAL*M only extends data into REPORTDATA of the TDREPORT. Since material extended into REPORTDATA by PAL*M is independent of keys, and PAL*M does not influence any other step the QUOTE generation pipeline, the integrity/authentication guarantees of Intel TDX attestation remain unchanged. $\mathcal{A}$dv may attempt to replay responses from the same *op* to Vrf, but this is detectable by Vrf due to PAL*M incorporating *Chal* as input in construction of TDREPORT (and subsequently QUOTE). We use Lemmas 1, 4, 5, 6, and 8 to ensure that a verifiable QUOTE was produced after an integrity check on a MAC-ed TDREPORT, and no secret keys are revealed through the protocol.

Since $\mathcal{A}$dv has control over the VMM, they can assign which image is loaded into the TD. Therefore, they could load an image containing a backdoor to assist with lying about datasets/models that are used for a particular operation. However, the TD image is measured by the TDX module before TD creation, and included in the QUOTE obtained by Vrf. Therefore, these attempts are detectable by Vrf. Furthermore, $\mathcal{A}$dv may attempt to assign a GPU without a TEE to the TD. In this case, Vrf detects it due to PAL*M including GPU attestation evidence when the GPU is used in obtaining outputs. In this case, Vrf

would observe GPU_{att}'s absence from the response. We use Lemmas 1 and 2 to show that a QUOTE containing PAL*M configuration (e.g., TD image and GPU) is not accepted by Vrf unless the reporting TD was created with the same configuration. We do not explicitly model the scenario of GPU reassignment in our formal verification: we assume the SPDM key exchange between the GPU and TD occurs before TD launch, as it is impossible to launch a GPU in Confidential Computing (CC)-mode without this [56, 127]. Given this, we assume the GPU–TD pair is a single trust boundary in our model.

Finally, when inputs or outputs are deemed private, $\mathcal{A}$dv may try to corrupt PAL*M TD to reveal them. However, through PAL*M measurer script and construction of TDREPORT, confidential inputs/outputs themselves are never extended into any TDREPORT or any downstream QUOTE that are visible by $\mathcal{A}$dv. We use Lemmas 7 and 13 outlined in Section 3.7 to verify that this is upheld.

Under the assumed threat model, $\mathcal{A}$dv cannot tamper with operation inputs, forge or replay attestation responses, or manipulate the TD configuration without detection by Vrf. Furthermore, the property attestation protocol is formally verified to uphold **[P1-3]** despite these attempts, as confirmed by the Tamarin model detailed in Section 3.7. Therefore, PAL*M satisfies **R2**.

3.6.3 Scalability & Versatility

Scalability. PAL*M satisfies **R3** as it can be applied to extremely large datasets and models, distributed computing by Prv, and enables verification by multiple verifiers.

PAL*M's use of MSH allows it to scale to extremely large datasets, beyond what can fit into memory, without sacrificing out-of-order access. We assume model weights can fit into the GPU's RAM (e.g., 96 GB for our prototype); when they do not, standard optimizations (e.g., quantization) reduce memory impact while maintaining performance. While some ML frameworks support memory-mapped model weights [54, 43], accessing them through GPU↔CVM↔disk adds significant overhead. Recent work shows quantization can eliminate the need for weight offloading [157]. When quantization is insufficient or undesirable, MSH can also apply to memory-mapped model weights, and not just datasets.

PAL*M can extend to distributed training [172], where each node returns proof(s) for its dataset subset. Due to MSH 's incremental design, it enables aggregation of component hashes into that of the full dataset. However, verification complexity, node-level corruption, and Byzantine resilience require further study, leaving *proofs of distributed training* as an intriguing direction for future work.

PAL*M uses digital signatures to create each QUOTE containing the property attestation. Therefore, any number of verifiers with access to the attesting device's public key can perform verification.

Versatility. PAL*M satisfies **R4** as it can apply to any CVM and GPU with TEE, and it can be adapted to any type of model architecture or dataset. Although PAL*M prototype uses Intel TDX as a CVM and NVIDIA H100 for TEE-enabled GPU, PAL*M's attestations can be generated by Prv running on any CVM-GPU configuration, given both have TEEs with built-in attestation functionality (e.g., AMD SEV-SNP [7] and NVIDIA Blackwell GPU [122]). Furthermore, PAL*M can be adapted to any type of model architecture or dataset, beyond those used in our experiments, as measurements are specified in software running inside a CVM that invokes a GPU with a TEE. Property attestations can also be easily added by extending PAL*M with additional measurer scripts corresponding to the new property, with no changes required by Inr or Vrf.

3.7 Formal Modeling with Tamarin

To prove the security properties of PAL*M, we model its property attestation protocol (from Figure 3.4) in the Tamarin Prover [103]. In our setting, QE, the TDX module, and PAL*M TDX are trusted. Communication between the TDX module, PAL*M TDX, and GPU occurs over secure channels. This manifests as custom precondition/postcondition facts. Channels that traverse the VMM, disk, or the network are considered untrusted. For example, QE to TDX Module communication passes through the VMM, dataset inputs are read from disk during mem-mapped operation, and each QUOTE is transmitted over the network prior to verification. These interactions are modeled using *In* and *Out* facts.

3.7.1 Security Goals

We formalize properties [**P1-3**] from Section 3.6.2 as the following goals that are provable in our formal model:

[**G1**] **Confidentiality of Secrets**: PAL*M measurements and outputs should not reveal secrets used to generate a QUOTE;

[**G2**] **TD Measurement Integrity:** a QUOTE is only valid if produced from a valid TD image that was indeed installed;

[**G3**] **Quote Authenticity:** any accepted QUOTE is signed by QE using its private key;

[G4] **Quote Integrity:** any accepted `QUOTE` was produced after performing a MAC integrity check on a `TDREPORT`;

[G5] **Output Integrity**: any accepted `QUOTE` uses $\mathcal{O}$ that was produced via *op* execution over specified inputs $\mathcal{I}$; and

[G6] **Input Measurement Integrity:** any accepted `QUOTE` was produced using $\mathcal{I}$ that was loaded into TD-protected memory.

Properties from Goals:

$$[\mathbf{P1}] \leftarrow [\mathbf{G1}] \wedge [\mathbf{G5}] \wedge [\mathbf{G6}]$$
$$[\mathbf{P2}] \leftarrow [\mathbf{G1}] \wedge [\mathbf{G2}] \wedge [\mathbf{G5}] \wedge [\mathbf{G6}]$$
$$[\mathbf{P3}] \leftarrow [\mathbf{G1}] \wedge [\mathbf{G3}] \wedge [\mathbf{G4}]$$

[**P1**] concerns operation integrity. It is established by [**G1**], [**G5**], and [**G6**]. [**G1**] ensures MAC/signature keys are not leaked by the protocol. [**G5**] and [**G6**] ensure that output/input measurements correspond to the operation.

[**P2**] concerns integrity of the environment and responses. It is established by [**G1**], [**G2**], [**G5**], and [**G6**]. [**G2**], [**G5**], and [**G6**] ensure response integrity, as it includes h_{TD}, $\mathcal{H}_{\mathcal{O}}$, and $\mathcal{H}_{\mathcal{I}}$.

[**P3**] concerns response authenticity. It is established by [**G1**], [**G3**], and [**G4**]. [**G1**] again ensures key secrecy, while [**G3**] and [**G4**] ensure any verifiable `QUOTE` was signed by the expected QE and MAC-ed by the expected TDX machine, respectively.

We use a Tamarin model to evaluate [**G1-6**] under four protocol scenarios, varying by dataset memory model and privacy of operation outputs:

- in-memory datasets to produce public inputs/outputs (**Case 1**);
- in-memory datasets to produce private inputs/outputs (**Case 2**);
- memory-mapped datasets to produce public inputs/outputs (**Case 3**);
- memory-mapped datasets to produce private inputs/outputs (**Case 4**).

3.7.2 Case 1: In-Memory Public Inputs/Outputs

```
// Case 1: In-memory and public inputs/outputs
Rule 1: TD Initialization
rule Init_TDX_TD:
  [ Fr(~k), Fr(~qsk), In(tdImage) ]
  --[ InitTD(tdImage), SecretKey(~k), SecretKey(~qsk), InitialInstall(tdImage) ]->
```

```
  [ !Key(~k), !QESignKey(~qsk), !QEVerKey(pk(~qsk)), Out(pk(~qsk)), !TDIsRunning(
    tdImage) ]

Rule 2: Get Inputs
rule Get_Operation_Inputs:
  [ In(op), In(I), In(chal), !TDIsRunning(hTD)]
  --[ InputMessage(op, I, chal) ]->
  [ OperationInputs(op, I, chal, hTD)]

Rule 3: Operation Execution (In-memory)
rule Execute_Operation:
  [ OperationInputs(op, I, chal, hTD) ]
  --[ RunOperation(op, I, chal, runOp(op, I)) ]->
  [ CompleteOperation(op, I, chal, runOp(op, I), hTD) ]

Rule 4: TD Report Generation
rule TD_Produce_Report:
    let
      O = runOp(op, I)
      mac = h(<h(I), h(op), h(chal), h(O), hTD, k>)
    in
    [ !Key(k), CompleteOperation(op, I, chal, runOp(op, I), hTD) ]
  --[ TDProducesReport(op, I, chal, runOp(op, I), hTD, k) ]->
    [ Out(<O, h(I), h(op), h(chal), h(O), hTD, mac>), ReportReady(op, I, chal, runOp(op
    , I), hTD, k) ]

Rule 5: Quote Generation
rule QE_Produce_Quote:
    let
      mac = h(<hI, hOp, hChal, hO, hTD, k>)
      sig = sign(<hI, hOp, hChal, hO, hTD>, qsk)
      O = runOp(op, I)
    in
    [ !QESignKey(qsk), In(<O, hI, hOp, hChal, hO, hTD, mac>), ReportReady(op, I, chal,
    O, hTD, k) ]
  --[ MacCheck(hI, hOp, hChal, hO, hTD, mac, k),
      QEQuotes(hI, hOp, hChal, hO, hTD, pk(qsk)) ]->
    [ Out(<O, sig>), SignatureReady(op, I, chal, O, hTD, sig) ]

Rule 6: Verification
rule Vrf_Verifies:
    [ !QEVerKey(pkQ),
      SignatureReady(op, I, chal, O, hTD, sig),
      In(<O, sig>)
    ]
```

```
--[ VrfAccepts(op, I, chal, O, hTD, verify(sig, <h(I), h(op), h(chal), h(O), hTD>,
  pkQ), pkQ) ]->
   []
```

We divide the protocol for Case 1 into three parts: (i) TD initialization, (ii) operation execution, and (iii) `TDREPORT` generation, `QUOTE` signing, and verification.

TD Initialization. We model steps (1)—(2) of the TD initialization phase from Figure 3.4 using Rule 1. This rule models TD creation (with GPU assigned and GPU_{att} obtained), provisioning cryptographic keys, and the initial state measurement.

This rule takes two fresh secret values, an internal MAC key ($\sim k$) and a QE signing key ($\sim qsk$), and a TD image as inputs, producing persistent facts for the resulting TD state: the MAC key, the QE signing/verification key pair, and a running TD loaded with that image. The QE public key is output via the untrusted channel to $\mathcal{A}$dv, while private keys are logged via *SecretKey* action facts. The *InitTD* and *InitialInstall* action facts capture the CPU measurement and extension of the TDX's initial software state.

Operation Execution. We model steps (1)–(5) after TDX initialization from Figure 3.4 with two rules: (i) fetch operation inputs (e.g., *op*, *Chal*, and $\mathcal{I}$) in Rule 2 and (ii) operation execution (Rule 3).

Rule 2 consumes an *op*, $\mathcal{I}$, and *Chal* from the network alongside a running TDX fact, binding them together into an *OperationInputs* fact that scopes execution to a live TDX. Rule 3 consumes these inputs and applies the abstract function *runOp(op, I)* which computes the operation output. Then, we log the execution via the *RunOperation* action fact and forward all relevant values as a *CompleteOperation* fact for the next steps.

`TDREPORT/QUOTE` **Generation and Verification**. This part models steps (6)–(11) of Figure 3.4. Rule 4 models `TDREPORT` generation. Then, Rule 5 models transferring `TDREPORT` to QE for signing via (untrusted) VMM. Lastly, the Rule 6 models making signed `QUOTE` available to Vrf for verification.

In Rule 4, TDX computes a MAC over the hashes of $\mathcal{I}$, *op*, *Chal*, $\mathcal{O}$, h_{TD}, using its MAC key k. These hashes, $\mathcal{O}$ and the MAC are passed via *Out()*, reflecting transmission via the VMM to QE. A *ReportReady* fact is also produced to maintain the internal binding between the operation's values and the TDX's secret key.

For `QUOTE` generation (Rule 5), QE receives `TDREPORT` via *In()* and recomputes the MAC to verify it before signing. Upon successful verification, QE signs the hashes of the $\mathcal{I}$, *op*, *Chal*, $\mathcal{O}$, and h_{TD} using qsk, and send *Out()* the signed `QUOTE` and $\mathcal{O}$.

Lastly, for Rule 6, Vrf consumes `QUOTE` from the network alongside the known QE public key, checking the signature against the expected hash values. The *VrfAccepts* action fact

records the outcome, binding Vrf's acceptance to the specific $\mathcal{I}$, *op*, *Chal*, $\mathcal{O}$, h_{TD}, and public key.

3.7.3 Security Proofs for Case 1

```
// Case 1: In-memory and public inputs/outputs
Lemma 1: Key Secrecy
lemma key_secrecy [reuse]:
  "All k #i. SecretKey(k) @ #i ==> not (Ex #j. K(k) @ #j)"

Lemma 2: TD Measurement Integrity
lemma td_measurement_integrity:
  "All op I chal O tdImage vk #n.
      VrfAccepts(op, I, chal, O, tdImage, true, vk) @ #n
    ==> Ex #i. InitialInstall(tdImage) @ #i & #i < #n"

Lemma 3: Output came from Operation and Input
lemma O_came_from_op_and_I:
  "All op I chal O hTD pkQ #i.
     VrfAccepts(op, I, chal, O, hTD, true, pkQ) @ #i
     ==> O = runOp(op, I)"

Lemma 4: Quote Authenticity
lemma quote_authenticity [reuse]:
  "All op I chal O hTD vk #n.
     VrfAccepts(op, I, chal, O, hTD, true, vk) @ #n
   ==> Ex #m. QEQuotes(h(I), h(op), h(chal), h(O), hTD, vk) @ #m & #m < #n"

Lemma 5: Quote Only If MAC is correct
lemma quote_only_if_mac:
  "All hI hOp hChal hO hTD mac k #i.
      MacCheck(hI, hOp, hChal, hO, hTD, mac, k) @ #i
    ==> mac = h(<hI, hOp, hChal, hO, hTD, k>)"

Lemma 6: Input Measurement Ingetrity
lemma input_measurement_integrity:
  "All op I chal O hTD vk #n.
      VrfAccepts(op, I, chal, O, hTD, true, vk) @ #n
    ==> Ex #j. InputMessage(op, I, chal) @ #j & #j < #n"
```

Lemmas 1– 6 are used to prove the protocol model detailed in Appendix 3.7.2 uphold security goals [**G1-6**] from Section 3.7.1:

[G1] **Confidentiality of Secrets**: Lemma 1 ensures that any key marked as secret at initialization is never derivable by $\mathcal{A}$dv;

[G2] **TDX Measurement Integrity:** Lemma 2 ensures that the Vrf only accepts an attestation if the TDX image referenced in the attestation was legitimately installed prior to verification, binding accepted attestations to a known, measured software state. Notably, the integrity guarantee is scoped to the TDX image measurement. Whether the expected TDX image was used requires consulting reference values, as described in Section 3.4.5. Note that due to SPDM occurring to establish a secure channel with GPU at the time of TDX creation [127, 56], these takeaways extend to the TDX+GPU environment;

[G3] **Quote Authenticity:** Lemma 4 ensures that Vrf only accepts a `QUOTE` originating from Prv's QE, ruling out any scenario where a valid attestation is produced without QE's involvement;

[G4] **Quote Integrity:** Lemma 5 ensures that QE invokes quote generation only when the received MAC is well-formed, binding quote generation to a MAC computed correctly over the expected hash values and the TDX's secret key, and guaranteeing that `TDREPORT` has not been tampered. Together, these two lemmas establish an unbroken chain of trust from the TDX's internal state through QE to Vrf;

[G5] **Output Integrity**: Lemma 3 ensures that whenever Vrf accepts an attestation, the reported output $\mathcal{O}$ is exactly the result of applying the operation op to input I via *runOp(op, I)* in the TDX. This rules out any scenario in which verification passes with a fabricated or mismatched result produced outside of the TDX;

[G6] **Input Measurement Integrity:** Lemma 6 ensures that for every *VrfAccepts* event, there must exist an *InputMessage* event carrying the same *op*, I, and *chal*. Since *InputMessage* is only emitted by *Get_Operation_Inputs*, which runs inside the TDX, and the subsequent *Execute_Operation* and *TD_Produce_Report* rules consume and propagate those exact values through internal linear facts before any output is produced, the accepted quote is guaranteed to have been computed over inputs that were loaded into TDX-protected memory.

3.7.4 Case 2: In-memory Private Inputs/Outputs

Here we describe the changes to the rules/lemmas for Case 1: 3.7.2–3.7.3 to enable Case 2 that accounts for private inputs/outputs.

```
// Case 2: In-memory and private inputs/outputs
Rule 7: Operation Execution (memory-mapped)
rule Execute_Operation:
```

```
  let O = runOp(op, I) in
  [ OperationInputs(op, I, chal, hTD)]
  --[ RunOperation(op, I, chal, O), SecretIO(O), SecretIO(I) ]
  [ CompleteOperation(op, I, chal, O, hTD) ]

Rule 8: TD Report Generation
rule TD_Produce_Report:
    let
      O = runOp(op, I)
      mac = h(<h(I), h(op), h(chal), h(O), hTD, k>)
    in
    [!Key(k), CompleteOperation(op, I, chal, runOp(op, I), hTD)]
  --[TDProducesReport(op, I, chal, runOp(op, I), hTD, k)]->
    [Out(<h(I), h(op), h(chal), h(O), hTD, mac>),
      ReportReady(op, I, chal, runOp(op, I), hTD, k)]

Rule 9: Quote Generation
rule QE_Produce_Quote:
    let
      mac = h(<hI, hOp, hChal, hO, hTD, k>)
      hashes = <hI, hOp, hChal, hO, hTD>
      sig = sign(hashes, qsk)
    in
    [ !QESignKey(qsk), In(<hI, hOp, hChal, hO, hTD, mac>), ReportReady(op, I, chal, O,
    hTD, k)
    ]
  --[ MacCheck(hI, hOp, hChal, hO, hTD, mac, k), QEQuotes(hI, hOp, hChal, hO, hTD, pk(
    qsk)) ]->
    [Out(<hashes, sig>), SignatureReady(op, I, chal, O, hTD, sig)]

Rule 10: Verification
rule Vrf_Verifies:
    let
      hashes = <hI, hOp, hChal, hO, hTD>
    in
    [ !QEVerKey(pkQ),
      SignatureReady(op, I, chal, O, hTD, sig),
      In(<hashes, sig>) ]
  --[ VrfAccepts(op, I, chal, O, hTD, verify(sig, <h(I), h(op), h(chal), h(O), hTD>,
    pkQ), pkQ) ]->
    []
```

Modifications to Rules. The key change from public to private input/output is how $\mathcal{I}$ and $\mathcal{O}$ are exposed. In Case 1, *TD_Produce_Report* outputs both directly to the network. In Case 2, it outputs only their hashes, keeping their original values hidden.

Execute_Operation also adds *SecretIO* action fact to mark $\mathcal{I}$ and $\mathcal{O}$ for privacy tracking.

Modifications to Lemmas. Goals [**G2-6**] in Case 2 are upheld using the same lemmas from Case 1, with one additional lemma added to ensure that [**G1**] still holds. Lemma 7 ensures that $\mathcal{I}$ and $\mathcal{O}$ remains secret. It states that any value marked *SecretIO* is never learned by $\mathcal{A}$dv, which makes it satisfy [**G1**]. We note that this privacy guarantee is with respect to network $\mathcal{A}$dv only, not a malicious Prv, since Prv can directly observe disk contents on the machine it controls. We do not consider disk encryption in this model.

```
// Case 2: In-memory and private inputs/outputs
Lemma 7: Private Input/Output
lemma private_IO [reuse]:
  "All X #i. SecretIO(X) @ #i ==> not (Ex #j. K(X) @ #j)"
```

3.7.5 Case 3: Memory-mapped Public Inputs/Outputs

Here we describe only the changes to Case 1 protocol from Sections 3.7.2–3.7.3 to enable Case 3 protocol by accounting for memory-mapped datasets with public inputs/outputs. The changes to the model are highlighted in yellow in Rules 11–14 and Lemmas 8–12.

Modifications to Rules. From Case 1, the rules for TDX Initialization (Rule 1) and Get Inputs (Rule 2) are also in the model for Case 3. However, Case 3 requires the remaining rules to change, plus the addition of three new rules.

Many changes to the operation execution are to model an incremental operation that has several steps: first, starting the execution, making a request to disk, modeling the disk itself, and finalizing that step (and the multiset). Rule 11 shows the first part of this process, in which a session ID is specified ($\sim$*sid*) along with an action fact *SessionStarted* to log this event. The postcondition of this rule is a fact that specifies the initial state. Notably, the *State* fact contains a field for output, and in this initial state, it is equal to performing the *runOp()* on an empty dataset.

Rule 12 specifics the first step for execution with a memory-mapped dataset. A record of the dataset is requested from the disk by specifying the index. This rule results in sending *Out* a get request from disk for the particular index, along with a fact to signify a pending lookup (*PendingLookup*).

Rule 13 specifies a simple rule to simulate fetching an item from the disk. The get request is received from *In*, and a batch that was fetched from the index is returned through *Out*. Both messages from this rule pass through an untrusted channel to simulate $\mathcal{A}$dv control over the disk.

```
// Case 3: Mem-mapped and public input/outputs
Rule 11: Execute Operation (mem-mapped)
rule Execute_Operation:
  [ OperationInputs(op, chal, hTD), Fr(~sid) ]
  --[ SessionStarted(~sid, op, chal, hTD) ]->
  [ State(~sid, op, chal, hTD, runOp(op, 'empty'), 'empty', 'none', %1)} ]

Rule 12: Execution step (lookup from disk)
rule Execution_lock_and_lookup:
  [ State(sid, op, chal, hTD, pretrained, block_data_multiset, block_indices, %index) ]
  -->
  [ Out(<'get', sid, %index>), PendingLookup(sid, op, chal, hTD, pretrained,
    block_data_multiset, block_indices, %index) ]

Rule 13: Return item from disk
rule DatasetStorage:
  [ In(<'get', sid, index>) ]
  -->
  [ Out(<'batch', sid, datasets(index)> ) ]

Rule 14: Execution step (use disk item)
rule Execution_unlock_and_continue:
  let
    model = runOp(op, batch)
    new_block_data_multiset = block_data_multiset ++ <%index, batch>
    new_block_indices = h(<batch, block_indices>)
  in
  [ In(<'batch', sid, batch>), PendingLookup(sid, op, chal, hTD, pretrained,
    block_data_multiset, block_indices, %index) ]
  --[ Trained(sid, op, chal, hTD,  model, new_block_data_multiset, new_block_indices)
    ]->
  [ State(sid, op, chal, hTD, model, new_block_data_multiset, new_block_indices, %index
     %+ %1) ]
```

Rule 14 specifies the continuation of the execution that uses the fetched batch from the requested index, and the accumulation of the multiset that is used to construct the MSH. The multiset construction is shown by accumulating $< \%index, batch >$ into the multiset and hashing it. This also creates a postcondition containing fact *State*, resulting in the possibility of re-evaluation of Rule 12 (e.g., to use more data records and grow the multiset further) or to continue to report generation.

```
// Case 3: Mem-mapped and public inputs/outputs
Lemma 8: Input Measurement Integrity
lemma input_measurement_integrity:
```

```
"All op I hI chal O hTD vk #n.
  VrfAccepts(op, I, hI, chal, O, hTD, true, vk) @ #n
  & not( I = 'empty')
 ==> Ex indices sid #j.
 Trained(sid, op, chal, hTD, O, I, indices) @ #j"

Lemma 9: Equal sampling sequences implies equal multiset
lemma Seq_Multiset_Match [use_induction,reuse]: all-traces
"All sid1 sid2 m1 m2 mset1 mset2 hi op1 op2 chal1 chal2 hTD1 hTD2 #i #j.
 Trained(sid1, op1, chal1, hTD1, m1, mset1, hi) @ i
& Trained(sid2, op2, chal2, hTD2, m2, mset2, hi) @ j
==> mset1 = mset2"

Lemma 10: Equal multiset implies equal input sample sequence
restriction MS_Seq_Match:
"All sid1 sid2 op1 op2 chal1 chal2 hTD1 hTD2 m1 m2 mset hi1 hi2 #i #j.
Trained(sid1, op1, chal1, hTD1, m1, mset, hi1) @ i
& Trained(sid2, op2, chal2, hTD2, m2, mset, hi2) @ j
    ==> hi1 = hi2"

Lemma 11: Equal input sample sequence implies equal output
lemma Seq_O_Match [use_induction,reuse]: all-traces
"All sid1 sid2 op chal1 chal2 hTD1 hTD2 m1 m2 mset1 mset2 hi #i #j.
Trained(sid1, op, chal1, hTD1, m1, mset1, hi) @ i
& Trained(sid2, op, chal2, hTD2, m2, mset2, hi) @ j
==> m1 = m2"

Lemma 12: Equal multiset implies equal output
lemma MS_O_match [reuse]: all-traces
"All sid1 sid2 op chal1 chal2 hTD1 hTD2 m1 m2 mset hi1 hi2 #i #j.
 Trained(sid1, op, chal1, hTD1, m1, mset, hi1) @ i
 & Trained(sid2, op, chal2, hTD2, m2, mset, hi2) @ j
 ==> m1 = m2"
```

The rules that specify `TDREPORT` Generation, Quote Generation, and Verification are largely the unmodified Rule 4, 5, and 6, respectively. A very minor change is that the `MSH` measurement is included along with all other hashes in each fact and action fact.

Modifications to Lemmas. One lemma from Case 1 is modified, and three more are introduced to account for the multiset construction. [**G1-4**] use Lemmas 1, 2, 4, and 5, respectively, as discussed in Section 3.7.3. [**G5-6**] are upheld by the the following lemmas:

[**G1**] **Output Integrity**: uses Lemma 3, discussed in Section 3.7.3 along with new Lemmas 11 and 12. Lemma 11 provides that two training operations that produce equivalent multiset hash (denoted hI) result in the same output (e.g., $m1$ and $m2$).

Similarly, Lemma 12 establishes that two training operations that use the same input multiset (*mset*) results in the same output;

[G2] Input Integrity: uses Lemmas 8, 9, and 10. Lemma 8 is a modified version of Lemma 6 that checks for the *Trained* event rather than *InputMessage*. In addition, since the dataset is measured incrementally, the integrity check does not apply when the input is empty. Given these two modifications, this lemma provides that a `QUOTE` passing verification implies that the input dataset is not empty and that it was indeed used for training. Lemma 9 provides that given two training operations have occurred with equal sampling sequences, the two inputs must be equal. Finally, Lemma 10 specifies that two training operations with the same recorded multiset must have used the same sampling sequences. This lemma uses the keyword *restriction*, meaning that it must be proven by hand. The proof for this is shown in Section 3.7.8.

3.7.6 Case 4: Memory-mapped Private Inputs/Outputs

Here we describe the changes to the Case 3 model from Section 3.7.5 to account for Case 4 with private inputs and outputs, and memory-mapped datasets.

Modifications to Rules. Only one rule requires modification, and that is Rule 15 for `TDREPORT` Generation. Here, $\mathcal{I}$ is logged with *SecretInput* and $\mathcal{O}$ is logged with *SecretOutput*, along with *op* and $\mathcal{I}$ that were used to produce $\mathcal{O}$. Additionally, the hash of the multiset (hI) is included in the set of output hash values sent via *Out*.

Modifications to Lemmas. Goals [**G2-6**] are upheld using the same lemmas as Case 3 protocol, described in Section 3.7.5. Goal [**G1**] is supported by the verification of Lemma 1 for key secrecy, as discussed in Section 3.7.3, and it is also supported by the verification of Lemma 13 and 14. Lemma 13 specifies that $\mathcal{O}$ cannot be learned given any operation and non-empty input dataset, and Lemma 14 follows a similar structure to Case 3 to check for discovery of $\mathcal{I}$ by $\mathcal{A}$dv.

```
// Case 4: Mem-mapped and private inputs/outputs
Rule 15: TD Report Generation (mem-mapped)
rule TD_Produce_Report:
    let
    O = runOp(op, I)
    mac = h(<h(hI), h(op), h(chal), h(O), hTD, k>)
    in
    [ !Key(k), State(sid, op, chal, hTD, O, I, hI, index) ]
  --[ TDProducesReport(op, I, hI, chal, O, hTD, k),
      SecretOutput(O, op, I),  SecretInput(I)
    ]->
```

```
    [ Out(<hI, h(op), h(chal), h(O), hTD, mac>),
      ReportReady(op, I, hI, chal, O, hTD, k) ]
```

```
// Case 4: Mem-mapped and private inputs/outputs
Lemma 13: Private Output
lemma private_O:
  "All O I op #i. SecretOutput(O, op, I) @ #i
  ==> not ( Ex #j. not (
      O = train(op, 'empty')
    ) & K(O) @ #j )"

Lemma 14: Private Input
lemma private_I:
  "All I #i. SecretInput(I) @ #i
   ==> not (Ex #j. not(I = 'empty') & K(I) @ #j)"
```

3.7.7 Verification Results

Each protocol case has the lemmas outlined in Sections 3.7.3–3.7.6, along with several "helper lemmas" that test functionality of the protocol. Table 3.9 shows performance statistics when using Tamarin to analyze the four protocol cases and prove all lemmas (functional and security). Case 2 is the most simplistic for two reasons. First, it uses a simpler model of the in-memory dataset. Second, it has fewer symbols that are controllable by $\mathcal{A}$dv, resulting in less exploration by Tamarin. Both Case 3 and Case 4 require verifying the lemma that takes the most steps for verification: Lemma 9, which requires 42 steps. The higher memory usage and run-time of Case 1 are due to a functional helper lemma that forces the prover to evaluate each rule. In this case, it must simultaneously solve for execution semantics of the custom function that is used to produce $\mathcal{O}$ plus the abstraction hO. In Case 2, execution semantics do not require as in-depth an exploration since $\mathcal{A}$dv only learns hO.

3.7.8 Multiset hashing security reduction

PAL*M's ability to access datasets out-of-order that cannot fit all at once into the TEE's memory depends on its use of an incremental multiset hash MSH satisfying the homomorphism

$$\texttt{MSH}(M_1 \cup M_2) = \texttt{MSH}(M_1) \cdot \texttt{MSH}(M_2).$$

Table 3.9: Statistics for lemma verification: Max Steps denotes the highest steps for any lemma; Total Steps is the sum of steps across all lemmas; Peak Mem. is the maximum RAM usage; Time is the total verification run-time.

Protocol	Max Steps	Total Steps	Peak Mem. (MiB)	Time (s)
Case 1	20	38	26799	89.2
Case 2	16	38	246	2.33
Case 3	42	98	2268	65.7
Case 4	42	115	270	5.10

$G^{\mathcal{B}_{\mathcal{A}}}_{\texttt{MSH-Collision}}$

$M_1, M_2 \xleftarrow{\$} \mathcal{B}_{\mathcal{A}}(n)$
if $\texttt{MSH}(M_1) = \texttt{MSH}(M_2) \wedge M_1 \neq M_2$
  return 1
else
  return 0
endif

Figure 3.5: `MSH` collision game.

Specifically, given an ordered dataset $\mathcal{D} = (d_1, d_2, \ldots, d_n)$ of n records, we require that the TEE, after receiving records $p_1, \ldots, p_n$, can be assured that the values $d'_1, \ldots, d'_n$ that it receives are the same values $d'_i = d_{p_i}$ in the original dataset, so long as

$$\texttt{MSH}\left(\{(1, d_1), \ldots, (n, d_n)\}\right) = \texttt{MSH}\left(\{(p_1, d'_1), \ldots, (p_n, d'_n)\}\right).$$

The left-hand side can be easily precomputed whenever p is a permutation; for convenience, we denote this hash $\texttt{MSH}(\mathcal{D})$.

The adversary's goal is defined in the following game in which the adversary $\mathcal{A}$ attempts to provide the TEE with records not matching $\mathcal{D}$, shown in Figure 3.6. Our goal is to reduce this to the `MSH` collision game $G^{\mathcal{B}}_{\texttt{MSH-Collision}}$, shown in Figure 3.5.

Theorem 1 (Incremental hashing of streamed data.)**.** *For any adversary $\mathcal{A}$ and dataset $\mathcal{D} = (d_1, d_2, \ldots, d_n)$, there exists $\mathcal{B}_{\mathcal{A}}$ such that:*

$$\Pr\left(G^{\mathcal{D},\mathcal{A}}_{\textit{Streamed-Incorrect-Data}}\right) \leq \Pr\left(G^{\mathcal{B}_{\mathcal{A}}}_{\textit{MSH-Collision}}\right)$$

$G^{\mathcal{D},\mathcal{A}}_{\textbf{Streamed-Incorrect-Data}}$

H $\leftarrow$ MSH ($\{\}$)
for $i = 1 \ldots n$
  $(p_i, d'_i) \xleftarrow{\$} \mathcal{A}()$
  $H \leftarrow H \cdot \texttt{MSH}(\{(p_i, d'_i)\})$
endfor
if $H = \texttt{MSH}(\mathcal{D}) \wedge \exists i : d'_i \neq d_{p_i}$
  return 1
else
  return 0
endif

$\Downarrow$ Reformulated using MSH homomorphism

$G^{\mathcal{D},\mathcal{A}}_{\textbf{Streamed-Incorrect-Data}'}$

$M_1, M_2 \xleftarrow{\$} \mathcal{B}_{\mathcal{A}}(n)$:
  for $i = 1 \ldots n$
    $p_i, d'_i \xleftarrow{\$} \mathcal{A}()$
  endfor
  return $\{(1, d_1), \ldots, (n, d_n)\}, \{(p_1, d'_1), \ldots, (p_n, d'_n)\}$
if $\texttt{MSH}(M_1) = \texttt{MSH}(M_2) \wedge \exists i : d'_i \neq d_{p_i}$
  return 1
else
  return 0
endif

$\Downarrow$ $\mathcal{B}_{\mathcal{A}}$ also wins

$G^{\mathcal{B}_{\mathcal{A}}}_{\texttt{MSH}\textbf{-Collision}}$

$M_1, M_2 \xleftarrow{\$} \mathcal{B}_{\mathcal{A}}(n)$
if $\texttt{MSH}(M_1) = \texttt{MSH}(M_2) \wedge M_1 \neq M_2$
  return 1
else
  return 0
endif

Figure 3.6: Game hops to prove Theorem D.1.

Proof. We observe that $G^{\mathcal{D},\mathcal{A}}_{\text{Streamed-Incorrect-Data}}$ can be restructured in terms of a new adversary $\mathcal{B}_{\mathcal{A}}$, shown in Figure 3.6. This game is very similar to $G^{\mathcal{B}_{\mathcal{A}}}_{\texttt{MSH}\text{-Collision}}$, except that the requirement $M_1 \neq M_2$ is replaced by $\exists i : d'_i \neq d_{p_i}$. That is, there exists some i such that there exist distinct elements $(p_i, d_{p_i}) \in M_1$ and $(p_i, d'_i) \in M_2$. Since M_1 is defined as

$$M_1 = \{(1, d_1), \ldots, (n, d_n)\},$$

it contains exactly one element having any particular index $i \in \{1, \ldots, n\}$, and none with indices outside this range. Since $(p_i, d'_i) \neq (p_i, d_{p_i})$, M_1 and M_2 therefore differ at the element with this index, and so $\mathcal{B}_{\mathcal{A}}$ also wins $G^{\mathcal{B}_{\mathcal{A}}}_{\texttt{MSH}\text{-Collision}}$. □

3.8 Discussion

Other Generative Models. Since PAL*M is built on PyTorch and Hugging Face, it applies to any hosted model with accessible weights, including other generative models not evaluated in this work (e.g., diffusion models [61]). For diffusion models, only the measured input and output changes (e.g., images instead of text). Properties may differ and can be added due to PAL*M versatility, however the approach for attestation would remain unchanged.

Other Data Integrity Techniques. PAL*M uses MSH to incrementally hash dataset records into a single value, independent of the access order, but requires all record are read exactly once. Thus, it applies only to full-dataset operations (common in LLM workloads). If partial access within a dataset is required, a Merkle tree can be used at an $\mathcal{O}(\log n)$ cost per lookup. A *proof of binding* can link MSH to a Merkle root, enabling selection based on system requirements.

Selecting *Chal*. Section 3.4.4 outlines a property attestation protocol that is initiated by Inr's request containing *Chal*. In conventional remote attestation settings, *Chal* is a nonce or counter used to prevent replay attacks and ensure response freshness. Given ML property attestation in this work is non-interactive with Vrf, selection of *Chal* should be derived from a timestamp rather than a traditional cryptographic nonce or counter. However, for proof of inference, a nonce remains valuable to prevent Prv from cherry-picking favorable executions. In this case, Vrf can publish nonce values with expiration times for use challenges. To further mitigate cherry-picking, Vrf may request only proof of session inference, allowing observability of response history.

Elliptic Curve-based MSH. The MSH overhead observed in Section 3.6.1 stems from the hardness of the discrete logarithm problem, which requires operating over a large prime (in

our case, a 2048-bit prime). Prior work has explored elliptic curve-based constructions [19, 98]. Maitin-Shepard, Tibouchi, and Aranha proposed `ECMH` [98], a `Mset-Mu-Hash` scheme based on binary elliptic curves and characteristic-2 variant of the Shallue-van de Woestijne encoding [24] for deterministic hashing to curve points. A simpler alternative is the try-and-increment method [15], which repeatedly attempts to map a value onto a valid elliptic curve point until successful.

We test proof of attribute distribution using an elliptic-curve-based `MSH` with try-and-increment over the NIST P-256 curve, and we observe a reduction in raw measurement overhead by 47.41%. This further supports the analysis from Section 3.6.1 that the dominant source of overhead is the `MSH` construction itself. Because try-and-increment has input-dependent timing [45], it is suitable only for public datasets. For private datasets, deterministic constructions, such as `ECMH` [98], should be used.

Chapter 4

EVOLVE: Enabling Verifiable Output Lineage for Virtualized ML Workload Execution

4.1 Introduction

This chapter introduces EVOLVE, a PoX mechanism for CVMs that enables run-time property attestation for ML operations. EVOLVE consists of two main components: an instrumentor and a run-time monitor. LLVM-based instrumentation is used to enforce policies at the intermediate representation level, ensuring all write and indirect call operations from the payload are routed through the monitor. The monitor mediates these policies at run-time, maintaining an execution flag (EXEC) that records whether the execution proceeded without violation. At the end of execution, EVOLVE produces a hardware-attested TDX quote binding the inputs, output, and execution integrity flag as verifiable proof. We prototype and evaluate EVOLVE using Intel TDX.

In summary, we claim the following contributions: we

1. design an instrumentation module (Section 4.4.2) and run-time monitor (Section 4.4.3) for CVMs that enables *intra-CVM access control* by mediating all memory write and indirect call requests from the instrumented payload, enforcing write integrity, output authorization, caller authentication, and tracking whether execution was violation-free;
2. a PoX protocol based on EVOLVE that demonstrates to a remote verifier the trustworthiness of reported outputs with respect to the requested operation (Section 4.4.4);

3. implement EVOLVE on Intel TDX as an extension of virtual firmware TD-Shim [36], with a range of model-routing container applications using classifier models; we also evaluate EVOLVE by measuring run-time overhead and fixed instrumentation cost, showing the approach is practical for real-world payloads (Section 4.5).

4.2 Background

4.2.1 Run-time attacks in ML software

Run-time attacks exploit vulnerabilities during program execution to corrupt/leak critical data or cause unintended (malicious) execution behavior [46, 152, 97, 149, 27]. They typically involve an adversary exploiting a memory vulnerability (e.g., buffer overflow, use-after-free), to corrupt control or data pointers towards their goal. Examples include remote code execution, in the form of control flow hijacking [97, 152] and code-reuse attacks (e.g., return/jump-oriented programming [144, 13]) or data-oriented attacks (direct data manipulation or data-oriented programming [27]).

ML software libraries containing memory vulnerabilities are equally vulnerable to such run-time attacks. This fact is demonstrated by recently disclosed vulnerabilities [115, 114, 113, 112, 111, 117, 116] in popular ML software libraries (e.g., PyTorch [137], TensorFlow [166], Paddle [131]) (including the recent vulnerability from PyTorch [117]), with potential to replace models at run-time [115], inject code [114, 113, 111], or hijack execution [112, 117, 116].

The nature of ML software and its ecosystem introduces additional vulnerabilities for run-time attacks. For example, today's ML software frameworks (e.g., PyTorch [137], Scikit-learn [134], Tensorflow [166], Keras [87]) use model file formats that are vulnerable to code injection [179]. For example, `pickle` is a common Python serialization format containing bytecode that can be exploited to modify a model in-place at run-time [33, 105]. As shown by Zhao et al. [179], this risk is prevalent for many model file formats used by popular frameworks. Furthermore, supply-chain attacks like *AI Jacking* [119], in which an attacker registers backdoored model/dataset files with names of reputable organizations, have the potential to lead to malicious behavior at run-time.

Model routing is a deployment technique in which an incoming request is dynamically directed to one of several available models, commonly used to balance cost and response quality by sending simpler queries to smaller, cheaper models and more complex queries to larger, more capable ones [25, 39, 128, 123, 125].

Because the routing decision occurs after deployment, it constitutes a run-time attack surface distinct from the models themselves. Recent work shows that routing decisions can be adversarially manipulated without modifying the underlying models, including biasing routing toward an unintended model [153], inference-time and training-time router backdoors [93], and cost escalation attacks [162]. An adversary can thus cause a request to be processed by an unintended model while the routing infrastructure and models remain unmodified, which would be undetectable to static, load-time measurement.

4.2.2 Proofs of Execution

PoX architectures [22, 120, 110, 140, 102] extend classic remote attestation techniques to attest to characteristics about the execution of an attested software. Specifically, PoX enables Prv to demonstrate to Vrf that:

1. Prv has a specified function installed;
2. Prv has executed the specified function at the time of the challenge/request;
3. any outputs reported by Prv were only modified by the most recent execution of that function.

A typical PoX protocol is shown in Figure 4.1 (abstracted from prior work [120]). First, Vrf requests PoX for expected application software (App) by sending a challenge (*Chal*), typically taking the form of a cryptographic nonce. Upon receiving this request, a RoT in Prv will take authenticated state measurement over: memory region (MEM) housing App, outputs (O) produced from App execution, metadata reflecting the execution status (EXEC), and *Chal*. This measurement produces an authentication token (H), typically taking the form of a MAC or digital signature. Upon a response from Vrf containing O, EXEC, and H, Vrf will inspect it to determine whether the execution and reported outputs are trustworthy.

To reflect the integrity of the execution, PoX requires a RoT that has exclusive access to EXEC, can monitor entry/exit into App, and can monitor access to O. The latter two requirements are used for the RoT to update EXEC. These requirements are typically met through trusted hardware mechanisms, such as dedicated hardware modules [22, 120] or TEEs [140, 110]. Furthermore, computation of H requires a secret key, hence implying a hardware-supported RoT. PoX has been primarily explored in embedded systems [22, 120, 140, 110], while similar notions have been explored on high-end systems by leveraging architecture-specific capabilities (e.g., late launch in AMD64 [6]) and trusted platform modules (TPM) [102].

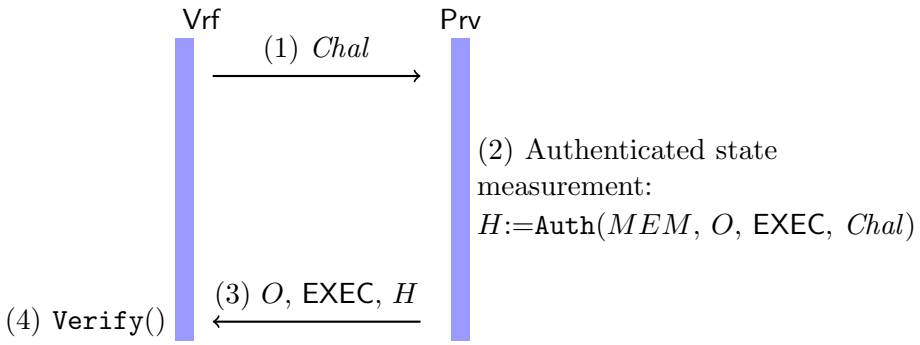


Figure 4.1: PoX built atop traditional remote attestation: a prover (Prv) demonstrates integrity of its software and execution properties to a verifier (Vrf).

4.2.3 LLVM Compiler

LLVM [89] is a compiler infrastructure organized around a common intermediate representation (IR) shared across source languages and target architectures. A source program is first translated into LLVM IR, which is a typed, low-level, architecture-independent representation distinct from both the original source and the final machine assembly. The same IR module can then be transformed, analyzed, or optimized before LLVM's backend lowers it to assembly for a specific target. Because all language frontends converge on this one IR, and all target backends consume it, transformations written against the IR apply uniformly regardless of source language or target architecture. IR-level instrumentation built on LLVM is a well-established strategy for software-level security [101, 130, 167, 88] and software fault isolation (SFI) [109, 132, 151, 16, 151], which similarly rely on the IR's uniform instruction forms to insert checks that would otherwise require handling many target-specific instruction variants.

LLVM IR represents a program as a set of functions, each composed of basic blocks of instructions in static static single assignment (SSA) form [70]. Two instructions are relevant here. A `store` instruction [73] writes a value to a memory address and is the single IR-level form for every memory write the source program performs, regardless of the operand's source-level type, addressing mode, or the specific machine instruction the backend eventually emits for it. An `alloca` instruction [71] allocates space in the current function's stack frame and returns a pointer to it; it is the standard representation for source-level local variables that have not been promoted to registers. A `call` instruction [72] invokes a function and can represent either a direct call, where the callee is statically identified by a function symbol, or an indirect call, where the callee is specified by a function-pointer value and the target is determined at run-time.

LLVM exposes a pass infrastructure for inserting custom transformations into this pipeline. A pass is a self-contained unit of code that inspects or rewrites IR; LLVM's pass manager [74] schedules passes and runs them over a module. Under LLVM's current pass manager design, a pass is implemented as a class exposing a `run` method invoked by the pass manager, and passes can be packaged into a shared library (a pass plugin) that the compiler loads at build time via a registration callback, rather than requiring the pass to be compiled into LLVM itself. Pass plugins can register callbacks at specific extension points in the optimization pipeline, which determines at what stage relative to other optimizations the custom pass executes.

4.3 Problem Statement

4.3.1 System and Adversary Model

System Model. Similarly to Chapter 3, this chapter assumes a Prv that executes an ML payload inside of a CVM (e.g., Intel TDX, AMD SEV-SNP, Arm Realms). For the remainder of the chapter, we refer to components using Intel TDX terminology, but our concepts apply across CVM technologies. This class of system typically runs multiple concurrent virtual machines atop a host VMM. Creation and configuration of a CVM is managed by the VMM. VMM controls accesses to I/O devices (e.g., GPU, disk).

Virtual firmware provides a lightweight boot and initialization layer for hardware-isolated virtual environments. Such systems are commonly used in confidential computing platforms to minimize boot time and reduce software complexity. For example, TD-Shim [36] is a simplified virtual firmware for Intel TDX that provides the minimal functionality required to launch guest workloads. These virtual firmware solutions are designed to execute *bare-metal containers*, i.e., containerized applications that run within a dedicated virtualized environment. Such applications may execute either directly on the virtual firmware or atop a simplified kernel. In this work, we focus on ML workloads belong to the former category and execute directly on the virtual firmware. We refer to this class of virtual-firmware workload as the *payload*. We explore implications for the latter class of workloads in Section 4.5.6. We assume the payload has multiple *operations*, each having its own model and code.

Because EVOLVE is implemented as an extension of TD-Shim, it also inherits several hardware protections provided by TD-Shim before launching the payload [36]. Data execution prevention (DEP) is configured such that all data pages are marked non-executable

and all code pages are read-only; no page is simultaneously writable and executable. TD-Shim additionally enables Intel control-flow enforcement technology (CET) [78, 83], which provides a shadow stack to protect return addresses against return-oriented programming (ROP) and indirect branch tracking (IBT) to constrain indirect branches, along with stack guard pages. Consequently, the payload code and its models are immutable at run-time.

Adv. Model We assume $\mathcal{A}\mathsf{dv}$ attempts to manipulate the payload's execution to induce unintended behavior with respect to its inputs, operations, or outputs. For example, $\mathcal{A}\mathsf{dv}$ may modify input or output data directly, or they may redirect control flow to overwrite outputs from a separate operation of their choice. Consistent with prior work [5], $\mathcal{A}\mathsf{dv}$ also controls the host environment, including the VMM and any memory or devices outside the TD (e.g., other VMs, storage, and I/O peripherals).

We consider two classes of run-time attacks. First, *control-flow hijacking*, where $\mathcal{A}\mathsf{dv}$ redirects the payload's execution to code that would not run during benign execution (e.g., invoking a different operation to write the output or jumping to unauthorized functions). Second, direct data manipulation (DDM) [26], where $\mathcal{A}\mathsf{dv}$ corrupts a pointer to perform an illegal read or write (e.g., overwriting an input, output, or execution state) without going through the authorized operation. Data-oriented programming (DOP) [63], where $\mathcal{A}\mathsf{dv}$ corrupts non-control data to influence execution without diverting control flow, is out of scope, as it does not alter the control-flow or write-integrity properties that EVOLVE enforces (discussed further in Section 4.5.6). Similar to the adversary model of PAL*M in Section 3.3.2, side-channel, denial of service, and physical attacks are also excluded.

We trust the Intel TDX CPU to correctly provide inter-CVM memory isolation, isolation from the host VMM, and hardware-backed measurement and attestation mechanisms (e.g., `TDREPORT` and `QUOTE`). However, TDX does not provide intra-TD isolation. Consequently, $\mathcal{A}\mathsf{dv}$ that gains execution within a TD can directly manipulate any code or data that is not otherwise protected. Furthermore, because TD creation is orchestrated by the $\mathcal{A}\mathsf{dv}$-controlled VMM, a TD is not inherently trusted. Instead, trust is established through TDX attestation mechanisms such as `QUOTE` [80].

4.3.2 Challenges

PoX is relevant for attestation of ML operations, as run-time attacks exploiting vulnerabilities in ML libraries can inject code and alter outputs without detection by current approaches. However, covering ML operations requires enabling PoX beyond embedded systems and into CVMs. However, current approaches for operation-level PoX do not directly apply to our application scenario because they:

1. provide PoX at CVM granularity [58], precluding observability to modifications of outputs made by unauthorized code at run-time;
2. require dedicated hardware modules for monitoring of App and accurate maintenance of EXEC [120, 22].
3. are designed for TEEs in embedded systems (e.g., TrustZone in Arm Cortex-M) [140, 110, 2], in which trusted software executes first and can configure the attested App;

Deploying a similar method in CVMs must rely solely on security features that are available in their architectures (e.g., AMD SEV-SNP, Intel TDX, Arm CCA). Since the VMM manages the environment and execution of each CVM, the VMM cannot be trusted to ensure EXEC reflects the true execution status of App or a true set of outputs produced by App. Therefore, directly extending prior approaches for PoX to CVMs is not possible. In summary, extending PoX to CVMs running ML software requires addressing the following challenges:

1. enabling intra-CVM isolation between App and a PoX measurer that loads, measures, and launches App upon request by Vrf;
2. monitoring access to outputs of App made by all software in the CVM;
3. ensuring exclusive access EXEC to a PoX measurer, which it updates during the run-time of App;
4. constructing PoX evidence, reflecting the execution status of App and whether reported outputs are trustworthy, to Vrf;

4.4 EVOLVE Design

4.4.1 Overview of EVOLVE

EVOLVE addresses these challenges and enables PoX of ML workloads deployed into virtual machines. EVOLVE has two main components: a software instrumentor and a run-time monitor. A high-level overview of an EVOLVE-enabled system is presented in Figure 4.2.

Before deployment, App source code is passed through the EVOLVE instrumentor in step (0), which instruments all memory write and indirect call operations and produces an instrumented binary $\texttt{App}_{\texttt{Instr}}$. The instrumented binary is then loaded into the TD in step (1).

EVOLVE partitions TD memory into two regions: the *shim region* and the *payload region*. Each region is further divided into named *sections* with fixed address ranges established by the linker at build time. The shim region contains the EVOLVE monitor section,

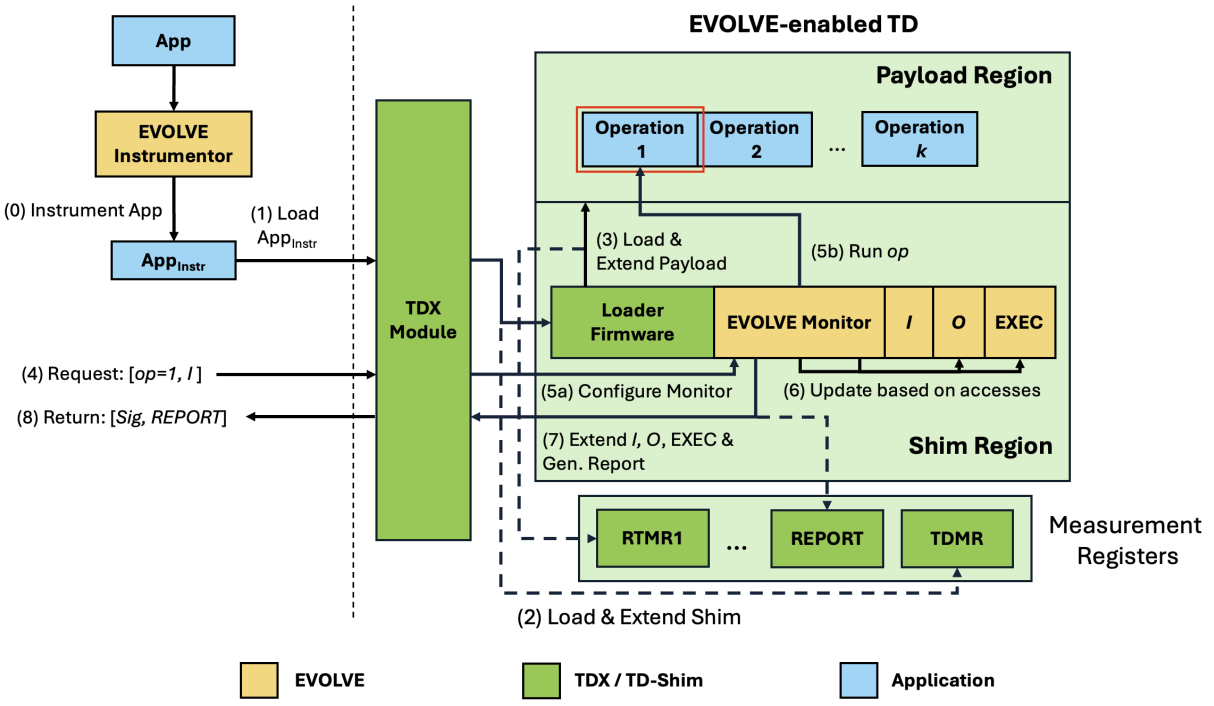


Figure 4.2: System overview of EVOLVE: an ML App with multiple operations is first instrumented before it is loaded and measured into EVOLVE-enabled TD. Upon receiving a request for a particular operation, EVOLVE's monitor tracks whether only the requested operation alters the reported output data.

the input section (I), the output section (O), and the EXEC section, collectively referred to as *critical sections*. In the payload region, each operation occupies its own section, referred to as *operation sections* (op). In the ML case, each op maps a ML model (e.g., op_1 maps to model1, op_2 maps to model2).

On startup, the TDX module loads and extends the shim region into the measurement registers in step (2). The loader firmware then loads and extends the instrumented payload into the payload region in step (3), binding the payload measurement to the TD initial configuration.

In step (4), the user sends a request containing the operation selection and input $[op, I]$ to the TD. The monitor configures itself and dispatches the requested operation in step (5a) and (5b), respectively, then places the input to I. As the operation executes, all memory write and indirect call requests from the payload are routed through the monitor, which updates O and EXEC accordingly in step (6). If all accesses are authorized, EXEC

remains 1. If any violation is detected, EXEC is set to 0. At the end of execution in step (7), the monitor extends I, O and EXEC into the TDX measurement registers and generates a hardware-attested report via the TDX module. The report and signature are returned to the user in step (8) as verifiable proof that the operation executed correctly and without violation.

EVOLVE design addresses the challenges with obtaining PoX beyond embedded systems. First, it enables intra-CVM isolation through a preprocessing and instrumentation phase that (a) defines the shim and payload region and (b) ensures exclusive access to critical sections (Section 4.4.2). Second, EVOLVE monitor component ensures all memory accesses to outputs and indirect calls are tracked, and an EXEC flag is properly updated to attest to the output lineage according to the expected operation (Section 4.4.3). EVOLVE enables Prv to construct authenticated PoX evidence and participate in a protocol for Vrf, in which Vrf can determine the trustworthiness of reported outputs (Section 4.4.4).

4.4.2 Instrumentation

To enforce the policies, EVOLVE needs to intercept every memory write and indirect call that the payload makes and route it through the monitor. A naive approach is to do this at the assembly level, pattern matching against specific write instructions such as `movl`, `movss`, and `movb`. However, the compiler may generate many variants of these instructions depending on the data type, addressing mode, and optimization level, making it difficult to capture all writes and calls reliably. Instead, EVOLVE instrumentor augments LLVM [89] by inserting an additional transformation for EVOLVE instrumentation into the pipeline, working at the IR level produced by the LLVM compiler internally between the source code and the final assembly. As shown in Figure 4.3, the normal compilation pipeline takes source code, produces LLVM IR, converts it to assembly, and links it into the final binary. At this IR level, all memory writes and indirect calls are uniformly represented by a single `store` and `call` instruction, respectively, regardless of the underlying data type or addressing mode. As described in Section 4.2.3, this means we only need to handle one instruction form instead of many assembly variants.

EVOLVE instrumentor inserts an additional pass at the LLVM IR stage before assembly is generated (see Secion 4.2.3). As shown in Figure 4.3, the pass first collects all `store` instructions across functions in the payload. However, not every `store` is instrumented. Functions that belong to the monitor are excluded via an exact-match allowlist to prevent the monitor from intercepting its own writes. Additionally, stores to stack-local variables are skipped. In LLVM IR, stack-local variables are represented as `alloca` instructions

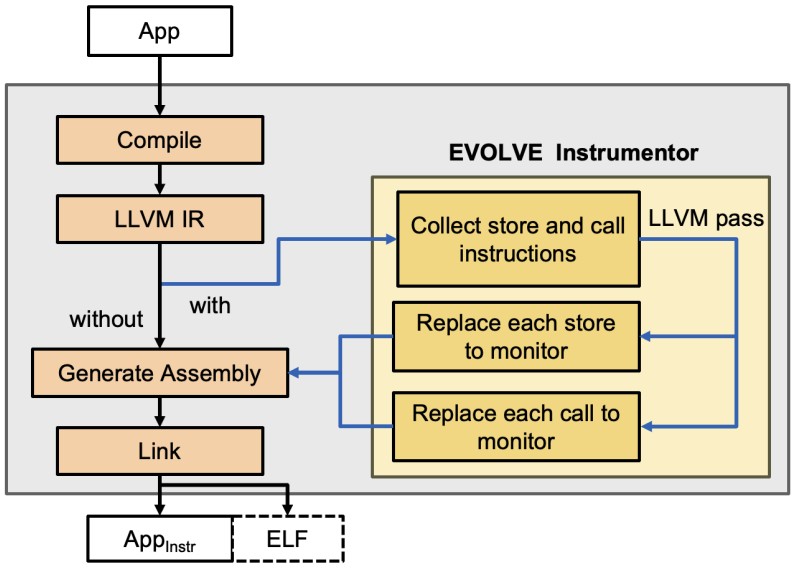


Figure 4.3: EVOLVE instrumentor: extends the LLVM pipeline to replace each `store` and `call` IR with a call to the EVOLVE monitor.

(see Section 4.2.3), thus, the stack occupies a completely different address range from any critical section, an `alloca`-derived pointer can never point to I, O, or any other critical section. Any write that does target a critical section will still be caught by the monitor's destination address check since those pointers would not come from `alloca` and would therefore be instrumented. For each remaining `store`, the pass extracts the destination address and the value being written, inserts a call to EVOLVE monitor to replace the store, which performs critical checks for access control and PoX evidence validity (described further in Section 4.4.3). The modified IR then flows back into the normal pipeline and continues to assembly generation and linking. The result is that every non-excluded memory write in the payload is replaced by a function call that goes through the monitor, and the payload source code is otherwise unchanged.

As shown in Figure 4.3, similar to memory writes, the pass first collects all indirect `call` instructions across functions in the payload. An indirect call is one where the target function is not known at compile time (i.e., a call through a function pointer), in contrast to a direct call to a fixed, named function. For each indirect call, the pass replaces the callee with a call into the EVOLVE monitor's foreign function interface (FFI) layer, passing the original target so it can be validated, such that no raw indirect call remains in the instrumented binary.

After instrumentation, the payload is compiled and linked with the monitor into an executable and linkable format (ELF) binary. Since a call from the payload into the monitor is undesirable, as it would tightly couple the payload to the monitor's internal implementation, the monitor includes a FFI layer. The FFI layer solves this by acting as an interface between the two components. It exposes a set of C-callable functions (e.g., `ffi_mem_write`, `ffi_indirect_call`, `ffi_get_or`) that the instrumented payload can call. Each FFI function internally captures the caller's return address before calling the corresponding monitor function as a proof of call origin for the monitor to check against (discussed further in Section 4.4.3). This separation ensures the payload interacts only with the FFI interface and the monitor internals remain isolated. The linker places each component into its designated memory section using a custom linker script. The resulting ELF binary is packaged with the loader firmware image.

4.4.3 Monitor

Metadata registers. EVOLVE tracks three values throughout execution to record what happened and whether it happened correctly. I holds the input provided to the ML model, such as input features and the model choice sent by the user, and is written by the monitor before or during execution. O holds the result produced by the operation (e.g., a prediction from the model) and must only be written by the operation section that the user requested. EXEC is a binary flag that initialized to 1 when execution begins and is set to 0 by the monitor if anything goes wrong. A value of 0 in EXEC means the execution cannot be trusted. A violation that triggers EXEC to be set to 0 occurs when one or more of the following conditions are met for either a memory write or an indirect call.

For memory writes:

- A write targets a critical section (I, monitor, or EXEC) directly from the payload.
- O is written by an unauthorized caller (e.g., an operation other than the one requested by the user).

For indirect calls:

- An indirect call targets an address within the shim region (I, O, monitor, or EXEC) that is not an authorized FFI entry point (e.g., a function that sets an EXEC flag).

The values of I, O, and EXEC are collected at the end of the execution and packed into the TDX attestation quote, allowing a verifier to check both the correctness of the output and whether the execution proceeded without any policy violation.

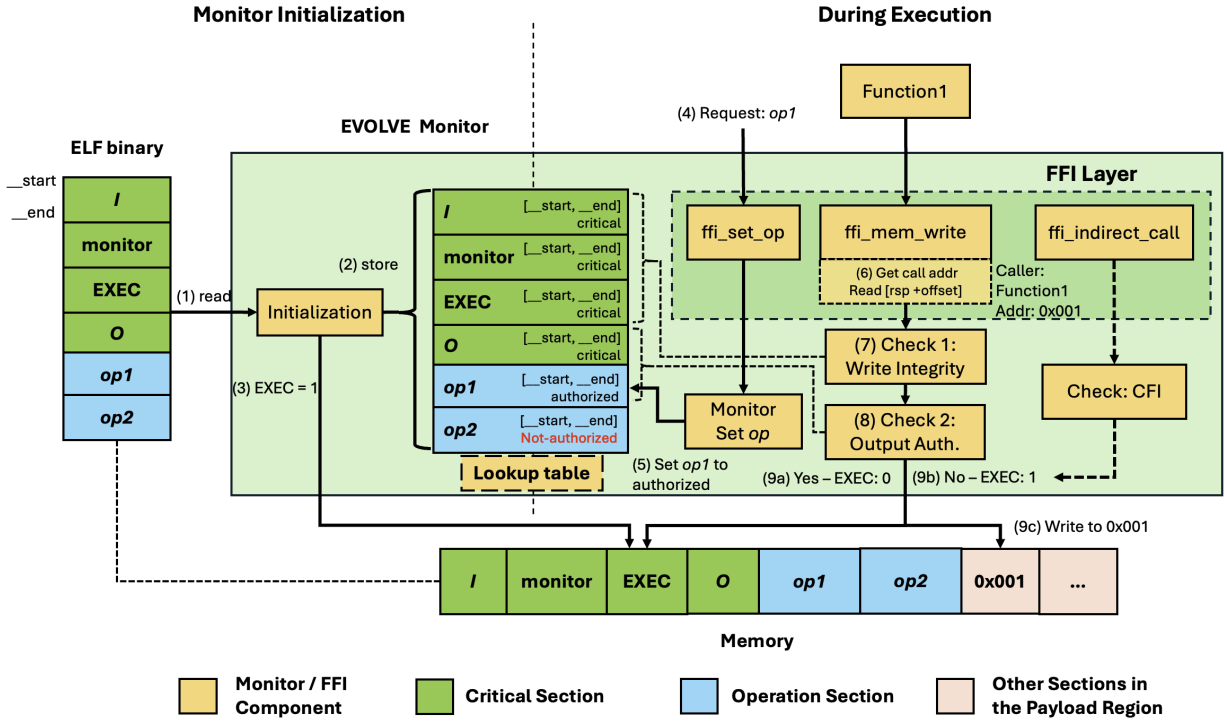


Figure 4.4: EVOLVE monitor internals: shows the process for extracting labels from the instrumented ELF binary to determine the location of memory regions, the FFI layer exposed to the payload, and the process for validating memory write and indirect call instructions. In the example, `Function1` performs a memory write, which is routed through the FFI layer to the monitor for the memory write check.

Initialization. The monitor is placed in the shim region with its own dedicated section, isolating it from the payload at the binary level. As shown in Figure 4.4, during initialization, the monitor reads the start and end addresses of each section directly from the binary in step (1). These boundaries are embedded by the linker at build time. The monitor then registers them into an internal lookup table in step (2), tagging each entry as I, monitor, EXEC, O, and one section per operation (e.g., in model routing explored in Section 4.5, `model1` and `model2` for the two models). Once all sections are registered, the monitor sets EXEC to 1 in step (3), indicating the start of a valid execution. Any write to EXEC is verified to ensure it only originates from within the monitor section, preventing the payload from directly manipulating EXEC.

As shown in Figure 4.4, in step (4), the user sends a request specifying the operation

(e.g., op_1). The monitor receives the request through `ffi_set_op`. In step (5), the monitor updates the lookup table to mark op_1 as the authorized operation and op_2 (and other operation, if any), as not-authorized. This step ensures that only the requested operation is permitted to write to O for this execution.

During execution. The model choice and input features are passed into the payload region, which runs the requested operation (e.g., op_1). Because EVOLVE instrumentor has replaced every memory write and indirect call in the payload with a call through the FFI layer; for example from Figure 4.4, `Function1` which perform memory write has to go through the monitor.

The monitor enforces two main properties on every such write. First, *write integrity*: when a write request arrives via `ffi_mem_write`, the FFI layer reads the return address of the call in step (6). Whenever a function call occurs, the CPU automatically places a return address on the stack, indicating where execution should resume after the function finishes. By reading this return address from a stack location (`[rsp + offset]` in x86 architecture), the FFI layer can determine the payload code location that invoked `ffi_mem_write`. The FFI layer then passes both the destination address of the write and this caller address to the monitor. The monitor first checks in step (7) whether the destination address falls within a critical section (I, monitor, or EXEC), using the internal lookup table. If so, EXEC is set to 0, recording that a violation occurred in step (9a) and blocking the write to those critical sections.

Second, *output authorization*: in step (8), the monitor checks whether the write target O and whether the caller address falls outside the authorized operation section set in step (5). In the authorized case, the caller falls within op_1, so EXEC remains 1 in step (9b). In the unauthorized case, anything not op_1 attempts to write to O, The monitor detects that the caller address falls outside the authorized operation section and sets EXEC to 0. In both cases (exclude the first check in step 7), the write proceeds to the destination address in step (9c), but any violation is permanently recorded in EXEC and will be visible to the verifier at attestation time.

If the call is an indirect call, the monitor enforces control-flow integrity (CFI). The monitor checks whether the indirect call targets the shim region. If so, the call is not allowed, as the target may be an internal monitor function (e.g., a function that sets an EXEC flag), unless the target corresponds to a legitimate FFI entry point that is explicitly allowed. We only need to check against function entry points because Intel CET's IBT already prevents indirect calls and jumps from landing at arbitrary offsets within a function, so any indirect call is guaranteed to target a valid entry point [83]. Unlike memory writes, which are recorded but permitted to proceed if the write does not target a critical section,

an unauthorized indirect call is blocked: the monitor clears EXEC and blocks the call, so the target is never reached. This ensures CFI by restricting indirect calls to authorized targets.

After execution. Once execution completes, the monitor checks that EXEC remains 1, confirming that no violations were detected. It then collects I, O, and EXEC and packs them into the `REPORTDATA`, and requests a hardware-attested report from the TDX module. This authenticated report can be used in a protocol to enable verification by a remote Vrf (discussed next in Section 4.4.4).

4.4.4 EVOLVE-based PoX Protocol

Figure 4.5 shows EVOLVE-enabled PoX protocol between Vrf and Prv. Prv consists of untrusted VMM and the QE on the host machine, a trusted TDX Module, and EVOLVE TD which is split into the payload region and EVOLVE monitor.

Offline phase. Before any request is made, the application is instrumented using EVOLVE instrumentor with LLVM pass (from Section 4.4.2), producing $\texttt{App}_{\texttt{Instr}}$ where every memory write is redirected through the monitor. This step happens only once per deployment.

Load time. The VMM specifies and creates the TD in step (1). The TDX Module then measures $\texttt{App}_{\texttt{Instr}}$ into h_{TD} and boots the TD in step (2). As part of the boot sequence, the monitor runs its initialization routine in step (3) and records the all relevant address boundaries of all sections, I, O, monitor, EXEC, and in the model routing case, model1 and model2. It then sets EXEC to 1 to mark the beginning of trusted execution.

Receiving a request. Vrf initiates the protocol by sending the operation (*op*) and an attestation challenge (*Chal*) as a request: *op*, *Chal* to Prv in step (4). The VMM forwards the request into the TD in step (5), then the monitor processes the request in step (6), configures itself with the authorized section based on *op* where only this section is allow to produce an output for this execution. The monitor then passes I and *op* to the payload region to begin execution in step (7)

Execution. The payload region runs *op* in step (8). Because all stores and calls have been replaced by the LLVM pass during the offline phase, each memory write and indirect call is intercepted by the monitor in step (9). For each intercepted write or call, the monitor performs checks against the policies in step (10). A failure to meet either check results in a violation by setting EXEC to 0. If the write or call passes the policy check, it proceeds in step (11).

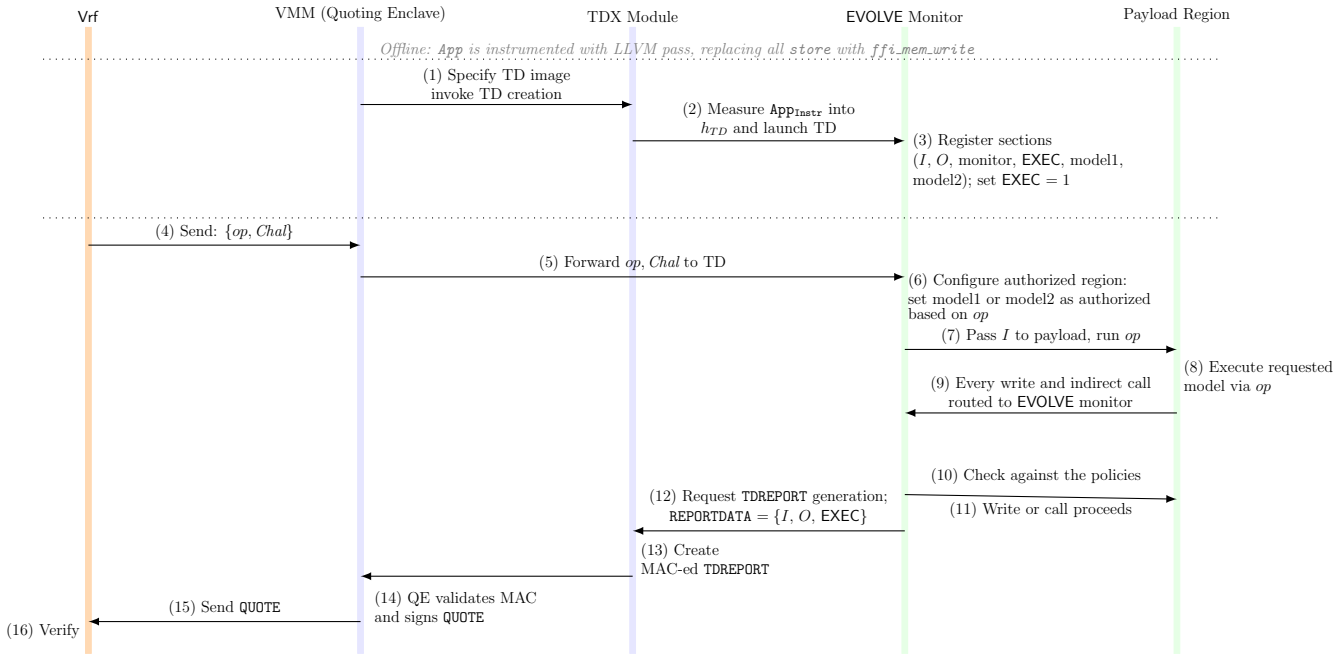


Figure 4.5: EVOLVE-enabled PoX protocol between Vrf and Prv, comprising TDX and EVOLVE TD. The protocol assumes App has already been instrumented. At load time, the TDX Module measures the instrumented binary and launches the TD. The monitor initializes its registers, all section regions and, sets EXEC = 1. At request time, Vrf sends a operation choice, the monitor configures the authorized operation, mediates memory writes and indirect calls, and generates a hardware-attested QUOTE. Vrf verifies the quote, including validating that EXEC = 1 .

Generating QUOTE. After execution, the monitor requests a TDREPORT from the TDX Module in step (12), with REPORTDATA = I, O, EXEC to capture all relevant data. The TDX Module produces a MAC-ed TDREPORT in step (13) and returns it to the VMM, which passes it to the QE in step (14). The QE verifies the MAC and produces a signed QUOTE in step (15), which is sent to Vrf in step (16).

Verification. Vrf verifies the QUOTE signatures to confirm that it originated from a genuine TDX TD and checks *Chal* to confirm freshness. It then inspects h_{TD} against a known reference value to confirm the correct instrumented binary was used. The final check is the EXEC flag: if EXEC is 1, Vrf is assured that O was written only by the authorized operation section and no policy violations occurred during execution. If EXEC is 0, on the other hand, the result is considered untrustworthy regardless of the value in O. Similar

to the verification step described in Section 3.4.4, Vrf should also validate the inputs by inspecting I packed in the `REPORTDATA`, consulting reference values from Tru as needed to confirm that the correct input features and model choice were used for this execution.

4.5 Implementation & Evaluation

4.5.1 Prototype Configuration

EVOLVE prototype runs on an Intel TDX-enabled host equipped with an Intel Xeon Silver 4514Y CPU and 512 GB of RAM. The TD is configured with 4 vCPUs and 2 GB of memory. EVOLVE is implemented as an extension to the TD-Shim [36]. EVOLVE monitor is therefore implemented in Rust while the payload is implemented in C. The firmware image is launched using QEMU [12].

We use model routing [158, 57] as a representative scenario to demonstrate EVOLVE. We consider model routing between two types of models with varying complexity: XOR decision tree (DT) and multi-layer perceptron (MLP) classifier models. The models used in this evaluation are summarized in Table 4.1. We train a small and large configuration of DT and MLP classifier models using the Breast Cancer Wisconsin Diagnostic (BCWD) [174] and ISOLET [34] datasets from UC Irvine Machine Learning Repository [86]. The DT configuration corresponds to the final trained model structure (number of trees, nodes, leaves, and depth), while the MLP configuration corresponds to the final network architecture (layer sizes and depths).

For the DT models, the small configuration uses a conservative setup with the square root of input features per split, shallow trees (maximum depth constrained by a small fixed bound or a dataset-size–dependent limit), and 10 trees. The large configuration increases model capacity by using substantially more candidate features per split, allowing deeper trees (scaled with sample size), and increasing the number of trees in proportion to input dimensionality. For example, the small DT for ISOLET contains 10 trees with 582 nodes, 296 leaves, and depth 5, whereas large DT scales to 246 trees with 231K nodes, 155K leaves, and depth 20.7.

For the MLP models, the small configuration uses a single hidden layer sized to the number of input features plus one. The large configuration uses a deeper architecture with four hidden layers whose widths decrease linearly from the input dimension to the output dimension, forming a progressively narrowing network toward the prediction layer. For example, on BCWD (30 features), the small MLP uses a single hidden layer of size 31,

Table 4.1: Summary of models used in evaluation. We train a small and large configuration of DT and MLP classifier models using the Breast Cancer Wisconsin Diagnostic (BCWD) [174] and ISOLET datasets [34]. The DT configuration corresponds to the final trained model structure (number of trees, nodes, leaves, and depth), while the MLP configuration corresponds to the final network architecture (layer sizes and depths). Finally, the last column shows the memory size of the model.

Dataset	**Model Size**	**Model Type**	**Configuration**	**Mem. Size**
BCWD [174]	Small	Decision Tree	(10, 318, 164, 5)	24.53 KB
	Large	Decision Tree	(12, 356, 184, 6.6)	28.23 KB
	Small	MLP	[31]	12.68 KB
	Large	MLP	[31, 21, 11, 1]	23.64 KB
ISOLET [34]	Small	Decision Tree	(10, 582, 296, 5)	45.22 KB
	Large	Decision Tree	(246, 231K, 155K, 20.7)	29.17 MB
	Small	MLP	[618]	4.58 MB
	Large	MLP	[618, 413, 207, 2]	8.35 MB

while the large MLP uses four layers [31, 21, 11, 1], decreasing uniformly to the output layer.

The model weights for each model are compiled into C header files for bare-metal container workloads of TD-Shim using emlearn framework [118], resulting in sizes spanning 12.68 KB to 29.17 MB. For each model and configuration, I and O are initialized to 0 and written at run-time by user-provided inputs (e.g., the input feature vector and the requested operation) and the output of the authorized operation section (e.g., authorized model1), respectively. EXEC is initialized to 0 and set to 1 by the monitor at startup.

As discussed in Section 4.3.1, as we consider bare-metal containerized applications, the user inputs are sent to the TD via a virtual serial port exposed by QEMU as a transmission control protocol (TCP) socket. After EVOLVE loads the payload and configuring the environment, it waits for a user request. The user requests the model for inference via another serial console. This choice is forwarded to the monitor through the FFI layer, which records the authorized model section for the duration of execution.

4.5.2 Metrics

The main overhead of EVOLVE consists of fixed costs and performance costs. Fixed costs capture the one-time storage overhead of deploying EVOLVE, specifically the binary size added to the system by the monitor and the instrumentation pass.

- **TD-Shim extension size:** the binary size of the monitor code added to TD-Shim, reflecting the storage used when deploying EVOLVE.
- **LLVM pass size:** the binary size of the instrumentation pass shared library.

Performance costs capture the run-time impact of EVOLVE on the payload execution, measuring how much overhead the instrumentation and monitor checks introduce compared to a baseline without EVOLVE.

- **Baseline:** the execution time of the payload compiled and run without EVOLVE instrumentation. We omit a non-TDX baseline, as prior work has evaluated the overhead of TDX itself [30].
- **Compiler overhead:** the additional time to compile and link the payload with the LLVM instrumentation pass compared to a standard build without instrumentation.
- **Monitor call time:** the time taken by the monitor to perform its policy check as the number of registered operation sections increases. Since the monitor checks the caller address against each registered operation section sequentially, we expect this to scale linearly with the number of models.
- **Total run-time overhead:** the execution time of the entire instrumented payload compared to the baseline, capturing the cost of routing every memory write and indirect call through the monitor at run-time.
- **Run-time overhead (%):** the run-time overhead from EVOLVE as a percentage of the baseline execution time.

4.5.3 Fixed costs

Two factors contribute towards EVOLVE's fixed costs: (1) size of the LLVM instrumentation pass and (2) size of EVOLVE as a TD-Shim extension. The LLVM instrumentation pass adds 245 KB to the build system. This is a build-time tool that runs inside the compiler and is never included in the deployed firmware image, so it does not contribute to the TD memory footprint. As for the EVOLVE TD-Shim extension, we measure this by comparing the TD-Shim binary size with and without EVOLVE, which increases from 221.5 KB to 232.4 KB, a 10.8 KB (4.91%) increase. In contrast to the cost of the LLVM pass, this cost is part of the deployed firmware image and represents the storage footprint of running

Table 4.2: End-to-end performance costs of EVOLVE for model routing scenario (mean ± stdev, $n = 30$).

	Decision Tree				MLP			
	Small		Large		Small		Large	
	Baseline	EVOLVE	Baseline	EVOLVE	Baseline	EVOLVE	Baseline	EVOLVE
BCWD								
Build-time	22.43 s ±1.28	25.24 s ±1.93	22.03 s ±0.90	25.25 s ±1.35	22.62 s ±1.07	26.45 s ±1.31	22.65 s ±1.22	25.81 s ±1.83
Run-time	2.75 s ±0.36	2.79 s ±0.37	2.77 s ±0.35	2.81 s ±0.30	2.68 s ±0.46	2.78 s ±0.41	2.76 s ±0.42	2.86 s ±0.33
Build-time (%)	—	+12.52	—	+14.59	—	+16.93	—	+13.94
Run-time (%)	—	+1.26	—	+1.46	—	+3.69	—	+3.56
ISOLET								
Build-time	27.34 s ±1.66	30.58 s ±2.11	27.09 s ±1.39	31.12 s ±1.76	33.64 s ±1.85	39.91 s ±2.21	33.96 s ±2.04	39.89 s ±2.24
Run-time	2.83 s ±0.41	2.87 s ±0.24	2.89 s ±0.32	2.91 s ±0.22	3.03 s ±0.31	3.09 s ±0.18	3.11 s ±0.22	3.18 s ±0.24
Build-time (%)	—	+11.84	—	+14.88	—	+18.62	—	+17.73
Run-time (%)	—	+1.48	—	+0.66	—	+2.03	—	+2.39

EVOLVE inside the TD. Notably, the standard TD-Shim is only used as a virtual firmware that launches a payload. In contrast, EVOLVE enables a shim-layer that remains active and enables dynamic monitoring of the payload.

4.5.4 Performance Costs

Table 4.2 shows the build time and run-time to launch and execute the operation in the payload, averaged over 30 runs, for the selection of models. The baseline is the time to run the same model routing scenario without any instrumentation or monitor from EVOLVE.

Build-time overhead is consistent across both datasets for each model type. For the DT models, build-time overhead ranges from 11.84% to 14.88% across both datasets and model sizes, an increase from 22.03–27.34 s (baseline) to 25.24–31.12 s (with EVOLVE). For the MLP, the overhead is slightly higher at 13.94% to 18.62%, with larger times

reflecting the larger model size, increasing from 22.62–33.96 s (baseline) to 25.81–39.91 s (with EVOLVE). The overhead is driven by the additional steps in the EVOLVE build pipeline described in Section 4.4.2: building the LLVM instrumentation pass, replacing all store and call instructions in the payload, and compiling the Rust monitor library. We note that in the model routing scenario, both the small and large models are loaded simultaneously, as $\mathcal{A}$dv may attempt to alter the routing decision toward the smaller, less capable model. Thus, their build times are expected to be similar and within the standard deviation of each other, which is confirmed in Table 4.2.

Run-time overhead is negligible across all configurations, ranging from 0.66% to 3.69%, with the higher overhead for the large MLP models reflects the greater number of monitor calls during inference. It is important to note that build time is a one-time cost per deployment, not a per-request cost. In a real-world deployment, the firmware image is instrumented and built once, and the resulting binary is loaded into the TD for every subsequent inference request. The build overhead is therefore amortized across all executions and does not contribute to the per-request latency experienced by the verifier. The relevant per-request costs are the run-time costs shown in Table 4.2, which are further broken down in Table 4.3.

Monitor run-time overhead. Table 4.3 breaks down the per-request run-time overhead of EVOLVE into four steps: (1) initialize and load the payload, (2) configure the monitor after receiving a request, (3) model execution, and (4) quote generation.

For the initialize and load payload phase, the overhead is negligible across all configurations and datasets, as this phase is dominated by TDX initialization with the only difference being the slightly larger binary from the monitor code added by EVOLVE.

The three sources of overhead introduced by the EVOLVE monitor are monitor configuration, model execution, and quote generation, most of which have no meaningful baseline equivalent (< 0.01 ms), except for model execution. Monitor configuration overhead is consistent and extremely low across all model types and sizes at 0.001 ms. This is expected because the monitor configuration only involves a single lookup and write to the internal lookup table, independent of the model size or complexity.

Model execution overhead varies significantly by model type, size, and dataset. For the DT, the overhead is negligible across both datasets: on BCWD, EVOLVE adds only 0.001 ms for both small and large; on ISOLET, small remains negligible while large increases from 0.248 ms to 0.263 ms, a difference of only 0.015 ms. This is expected as decision tree inference generates few memory writes or indirect calls regardless of dataset. For the MLP, the overhead is more pronounced and scales with both model size and dataset complexity. On BCWD, MLP Small overhead is 0.008 ms and MLP Large is 0.023 ms

Table 4.3: Run-time overhead of EVOLVE monitor. Mean ± stdev over 30 runs (excluded the first run).

	Decision Tree				MLP			
	Small		Large		Small		Large	
	Baseline	EVOLVE	Baseline	EVOLVE	Baseline	EVOLVE	Baseline	EVOLVE
BCWD								
Init. & Load Payload	2.76 s ±0.4499	2.79 s ±0.3981	2.77 s ±0.4523	2.81 s ±0.4374	2.68 s ±0.3956	2.78 s ±0.3139	2.76 s ±0.3240	2.86 s ±0.3360
Monitor Config.	— —	0.001 ms ±0.0001	— —	0.001 ms ±0.0001	— —	0.001 ms ±0.0002	— —	0.001 ms ±0.0002
Model Execution	0.002 ms ±0.0005	0.003 ms ±0.0009	0.003 ms ±0.0005	0.004 ms ±0.0006	0.010 ms ±0.0002	0.018 ms ±0.0002	0.015 ms ±0.0031	0.038 ms ±0.0085
Quote Generation	0.019 ms ±0.0029	0.019 ms ±0.0030	0.019 ms ±0.0037	0.020 ms ±0.0034	0.019 ms ±0.0044	0.020 ms ±0.0037	0.020 ms ±0.0045	0.020 ms ±0.0037
ISOLET								
Init. & Load Payload	2.83 s ±0.05	2.87 s ±0.06	2.89 s ±0.05	2.90 s ±0.09	3.02 s ±0.15	3.09 s ±0.10	3.11 s ±0.26	3.18 s ±0.18
Monitor Config.	— —	0.001 ms ±0.0001	— —	0.001 ms ±0.0001	— —	0.001 ms ±0.0002	— —	0.001 ms ±0.0002
Model Execution	0.003 ms ±0.0005	0.004 ms ±0.0006	0.248 ms ±0.0439	0.263 ms ±0.0490	3.63 ms ±0.5705	3.95 ms ±0.4936	7.71 ms ±1.328	8.93 ms ±1.024
Quote Generation	0.018 ms ±0.0029	0.018 ms ±0.0029	0.018 ms ±0.0029	0.018 ms ±0.0036	0.020 ms ±0.0040	0.020 ms ±0.0044	0.021 ms ±0.0048	0.021 ms ±0.0054

over the baseline. On ISOLET, the overhead is substantially higher due to the larger feature space (617 features). MLP Small increases from 3.628 ms to 3.949 ms (+0.321 ms), and MLP Large increases from 7.708 ms to 8.926 ms (+1.218 ms). This reflects the fact that larger MLP models with more features generate significantly more memory write per inference, each of which is routed through EVOLVE monitor.

Quote generation overhead is consistent across all configurations at approximately 0.018–0.021 ms for both baseline and with EVOLVE, as it is always a single `TDCALL` at the end of execution regardless of model size or dataset. Overall, the total per-request overhead of EVOLVE is low for DT models across both datasets, and scales with the number of memory writes/indirect calls for MLP models, with the worst case at +1.218 ms for large MLP trained on ISOLET dataset.

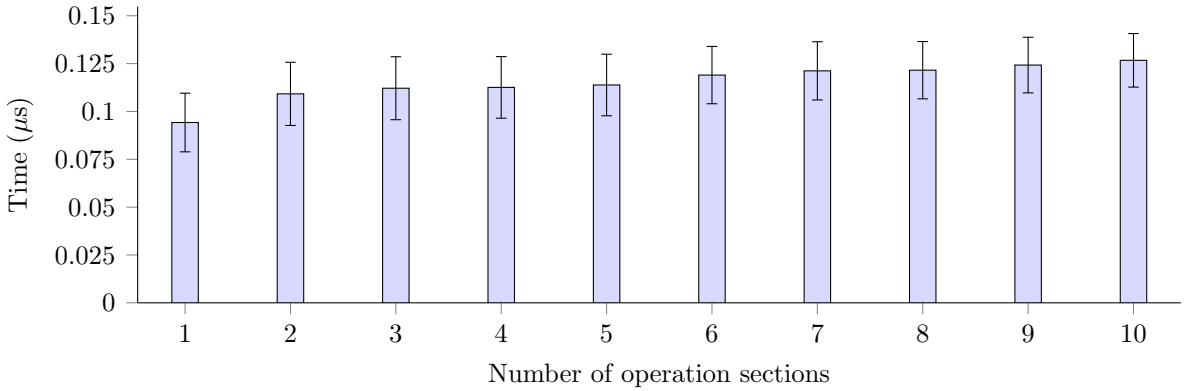


Figure 4.6: Monitor call time as the number of operation sections increases (mean ± stdev over 30 runs, run 1 excluded as warm-up). Each bar represents the average of 100 `ffi_mem_write` calls per run.

Monitor call time. Figure 4.6 shows how the monitor call time scales as the number of operation sections increases from 1 to 10. Each bar represents the average of 100 `ffi_mem_write` calls per run, averaged over 30 runs with the first excluded as warm-up. The results show a consistent upward trend, from 0.094 μs at $N = 1$ to 0.127 μs at $N = 10$, with consistent variance across all runs (0.014–0.015 μs). The overhead is not strictly linear but follows a generally increasing trend with the largest jump occurring between $N = 1$ and $N = 2$ (15.89% increase) and smaller increment thereafter. This is consistent with the monitor's implementation, which checks the destination address and caller against the internal lookup table sequentially. As more operation sections are registered, the scan covers more entries, leading to a gradual increase in per-call overhead. The relatively small increase per additional section confirms that the monitor's check is efficient in practice with negligible overhead even as the number of operations grows.

4.5.5 Security Analysis

Based on $\mathcal{A}$dv capabilities defined in Section 4.3.1, they may attempt to corrupt EVOLVE-generated PoX evidence by (1) corrupting TD creation (2) corrupting execution without detection (3) forging EVOLVE-generated reports. Here we describe while EVOLVE security guarantees are upheld in each scenario.

(1) Corrupting CVM Creation. $\mathcal{A}$dv may attempt to insert backdoor into the monitor or alter the monitor functionality by recompiling its own. However, h_{TD} is ex-

tended into the PoX evidence obtained by Vrf, and Vrf has knowledge of the expected EVOLVE monitor implementation. Therefore, such attempts would be detected by Vrf and subsequent PoX evidence would be rejected. Due to TDX protections, direct access to TD memory after creation via VMM is not possible [5]. Thus, direct illegal modifications to EVOLVE or the payload's code/data is not possible.

(2) Corrupting Payload Execution. $\mathcal{A}$dv may attempt to corrupt the execution itself after the TD has been created and launched through run-time attacks (discussed in Section 4.2.1). One possibility is that $\mathcal{A}$dv may attempt to reroute the request to use a different operation (i.e., model and/or code) than what was requested. This is not possible without detection with EVOLVE because (1) the request first passes through EVOLVE, which configures its monitoring of the requested operation and (2) execution of another operation to produce outputs propagated to O will be detected by EVOLVE monitor and result in clearing of EXEC (Section 4.4.3), which will be detected by a Vrf in subsequent PoX protocol (Section 4.4.4).

$\mathcal{A}$dv may attempt to overwrite components of the operation (e.g., the model or code [115, 114, 113, 111]), such that labels extracted from the original payload ELF by EVOLVE no longer reference the expected versions. However, this is not possible because TD-Shim enforce DEP, marking all code pages as read-only and all data pages as non-executable [35]. As a result, the model and operation code placed in the payload region cannot be overwritten at run-time, ensuring that the section boundaries registered by EVOLVE at initialization continue to reference the expected code throughout execution.

For the same reason, $\mathcal{A}$dv cannot remove the instrumentation from App at run-time. Furthermore, use of an application without instrumentation is detectable through the PoX evidence, as it is measured into the run-time measurement register before EVOLVE receives any request (Section 4.4.1).

Similarly, since all memory store instructions are instrumented, all attempts that could corrupt O are preceded by a comparison of the caller's address to the address range of the expected operation (Section 4.4.2). This prevents use of an unauthorized operation to corrupt outputs without detection. Since direct writes without detection are not possible, $\mathcal{A}$dv may instead attempt to jump to a gadget in EVOLVE code that resets EXEC to 1 after using an incorrect operation. This is not possible for two reasons. First, TD-Shim enables Intel CET [35, 78, 83]: the shadow stack protects return addresses from corruption, while IBT ensures that indirect calls and jumps can only land on valid function entry points rather than in the middle of a function. IBT therefore provides *coarse-grained* forward-edge CFI: it prevents landing at arbitrary offsets but does not prevent $\mathcal{A}$dv from redirecting an indirect call from one legitimate function entry point to another, such as

a monitor-internal function. EVOLVE builds on IBT to provide *fine-grained* forward-edge CFI through its instrumentation (Section 4.4.2). The payload can only reach the EVOLVE monitor through its FFI layer, and every indirect call is validated against the target's section. Thus, any attempt to call directly into a monitor-internal function, rather than a legitimate FFI entry point, is blocked. Since no function exposed through the FFI layer can be used to directly reset EXEC to 1 (Section 4.4.3), $\mathcal{A}$dv cannot forge a valid execution state even by manipulating control flow.

(3) Forging PoX Reports. $\mathcal{A}$dv may attempt to forge reports by either (1) forging outputs of the operation (2) forging the `TDREPORT` and `QUOTE` or (3) replay a previous result from the same operation. First, $\mathcal{A}$dv cannot report outputs that were not produced with the requested operation without executing it. EVOLVE only reads from O region to report outputs in the PoX evidence, and O is only modifiable by the payload through EVOLVE mointor, which updates EXEC accordingly (Section 4.4.3). Secondly, forging the `TDREPORT` and `QUOTE` would require leaking either the Intel TDX CPU's internal MAC key or the QE's secret key. Both are secured through Intel TDX's security features, aside from architectural side-channels and physical attacks that are not within the $\mathcal{A}$dv model of this work (Section 4.3.1). EVOLVE extends data into the `TDREPORT` that is relevant for completing the PoX evidence, but this data is completely independent of either key. Finally, $\mathcal{A}$dv may attempt to replay the response from a previous execution with the same operation and payload. However, Vrf includes attestation challenge (*Chal*) for freshness in its input and requires it to be used as auxiliary data in the attestation response. During execution, access to *Chal* is also prevented in a similar manner to inputs and code. Since *Chal* is requires as a part of the evidence construction, and it cannot be replace or modified, $\mathcal{A}$dv cannot successfully replay a previous result.

4.5.6 Discussion

Code duplication and shared libraries. EVOLVE determines the origin of each write to O based on the section of the calling code, which requires each operation's code to reside in its own section. As a result, if multiple operations share common code (e.g., a shared inference library), that code must currently be duplicated into each operation's section so that the caller can be unambiguously identified. This introduces a memory overhead proportional to the amount of shared code and the number of operations. However, without sacrificing security, duplication of shared code is a requirement for related mechanisms, such as TEEs due to memory isolation [80, 156] and compartmentalization techniques [91], which similarly replicate or carefully partition shared code to maintain isolation boundaries. A future version of EVOLVE could treat a shared-library region as a distinct section and track

which operation invoked it (e.g., by recording the originating operation across the call into shared code), allowing sharing code while still attributing outputs to the correct operation.

Extent of run-time protection. EVOLVE enforces write integrity and control-flow integrity at the boundary between the payload and the monitor, ensuring that outputs are produced only by the authorized operation and that the monitor cannot be compromised. However, EVOLVE does not protect against attacks contained *entirely within* an operation's own code. For example, if the model code itself contains an exploitable gadget, $\mathcal{A}\mathsf{dv}$ could hijack control flow within that operation without ever crossing the monitor boundary, and such an attack would not be reflected in EXEC. Mitigating in-operation control-flow attacks is complementary to EVOLVE and could be addressed by combining EVOLVE with existing hardware mechanisms such as Intel CET, which constrains control flow within the payload itself, or with additional intra-operation CFI instrumentation. Relatedly, data-oriented attacks [28] are classified into DDM [26] and DOP [63]. EVOLVE covers DDM in which a payload source pointer writes into a non-payload target memory address, as such writes are routed through the monitor and validated against the critical regions. However, the remaining types of data-oriented attacks, including DDM and DOP that remains within the payload's own memory, are not covered.

Towards a formal proof. A complete formal security proof of EVOLVE is left as future work. The key challenge is that the formal verification approach used in PAL*M does not directly apply here. In Section 3.7, the formal proof of PAL*M operates at the protocol level, treating the internals of the TD and trusted GPU as a single trust boundary and reasoning only about message passing and cryptographic guarantees. EVOLVE, however, introduces two additional components that would need to be modeled: the LLVM instrumentation pass and the monitor implementation, which go beyond protocol-level verification and would require new formal methods not explored in PAL*M. This makes a complete proof significantly more complex; therefore, we leave it as future work. Here we outline the key proof obligations and a possible approach using the Tamarin prover [103]. At a high level, the security of EVOLVE relies on three properties: (1) *instrumentation completeness*, (2) *monitor integrity*, and (3) *attestation soundness*.

First, instrumentation completeness requires that every non-excluded memory write in the payload be replaced by a call to the monitor, so that no write can reach a critical section without passing through the predefined interface (`ffi_mem_write`). In a Tamarin model, the payload's memory writes would be modeled as rules that produce action facts, and completeness would be expressed as a lemma stating that whenever a write action fact targeting a critical section appears in a trace, a corresponding monitor invocation action fact must precede it.

Second, monitor integrity requires that the monitor's internal state (i.e., lookup table, EXEC, and the authorized operation section) cannot be tampered with by the payload. This would be modeled by defining rules that govern how monitor state facts can be created or updated, with the payload represented as an untrusted agent whose rules can only interact with the monitor through the FFI interface, and a lemma asserting that no rule reachable by $\mathcal{A}$dv can directly modify monitor state facts. Third, attestation soundness, as in PAL*M at the protocol level, requires that the QUOTE faithfully reflect the monitor's recorded state. This would be expressed as a lemma stating that whenever Vrf accepts QUOTE, there exists a trace in which the monitor recorded the corresponding EXEC, I, and O values, with a MAC and QE signature modeled as trusted rules that $\mathcal{A}$dv cannot forge. Given these three lemmas, a Tamarin proof would establish that Vrf who receives a valid QUOTE with EXEC being 1 can conclude that the output in O was produced by the authorized operation without any policy violation.

Incorporating EVOLVE with Property Attestation. EVOLVE is complementary to property attestation systems such as Laminator [41] and PAL*M (Chapter 3). While these works attest that an ML model/dataset satisfies a desired property, it assumes the underlying execution environment correctly executes the attested computation. In contrast, EVOLVE provides proof that a specific operation executed and that the reported output was derived exclusively from that operation without unauthorized modification. As a result, EVOLVE can serve as a foundational execution-attestation layer beneath property attestation systems and incorporated as a part of property attestation evidence. For example, EVOLVE evidence can be used to establish the integrity of a particular inference or training operation and then validate higher-level properties of the resulting artifacts using either Laminator or PAL*M. Combining the two would provide both execution integrity and property-specific assurances.

Adapting to Kernel Payloads. The current design of EVOLVE assumes bare-metal container payload that executes directly atop the virtual firmware, as one version supported by the TD-Shim [36]. TD-Shim also supports kernel-based deployments. Extending EVOLVE to support these deployments would require an alternative approach, as instrumenting store instructions of both application and kernel code would incur too much overhead. Furthermore, this poses an additional challenge since the kernel can manage memory of the applications. One possibility is to draw from enclave models (e.g., Intel SGX [77]) to isolate an application from the kernel at run-time based on requests and still maintain the intra-CVM isolation. However, this would require careful consideration and validation to ensure security is upheld, thus, it is an intriguing direction for future work.

Large Generative Models. The current version of EVOLVE is compatible with CPU-based inference workloads that execute entirely within the CVM. Future work can include

extending EVOLVE to LLM inference frameworks, such as llama.cpp, to enable PoX for prompt processing and generation. However, scaling EVOLVE to modern generative artificial intelligence (AI) workloads would require addressing similar challenges to PAL*M related to managing of the GPU. In particular, although direct assignment of GPU is straightforward by following the methods of PAL*M, handling of the NVIDIA drivers that enable secure communication between the GPU and container payload is a challenge.

Chapter 5

Related Work

5.1 Cryptographic Proofs of ML operations

A broad line of work focuses on verifying properties of ML systems. Cryptographic approaches, such as those based on secure multi-party computation (SMPC) and ZKPs, provide strong verifiability guarantees without requiring trust in hardware extensions. SMPC has been used to attest distributional properties of datasets, although it requires interaction with a verifier for every attestation, limiting scalability [40]. Similarly, Arc combines SMPC with publicly verifiable commitments to establish auditable links between training and inference executions [96]. ZKPs have been applied to verify the correctness of training for a range of models, including logistic regression [52], neural networks [1], and LLM fine-tuning [145], as well as to generate proofs of correct inference for diverse ML models, including LLMs [65, 94, 160]. Beyond functional correctness, ZKPs have also been used to attest security and compliance properties such as differential privacy [155, 48] and fairness [47, 154]. While these approaches offer strong cryptographic assurances, their computational cost remains a significant challenge for practical deployment, with some systems requiring minutes to verify a single training iteration [1].

5.2 TEE-based ML attestation

TEEs have also been explored for both confidentiality of models and data [178, 161, 95, 108] and integrity guarantees [41, 148, 58, 140, 159, 23]. Laminator uses Intel SGX to verify

properties of classifier model execution on CPUs [41], while Schnabl et al. attest the execution of code, ML models, and inference outputs [148]. PAL*M [23] demonstrated how to obtain property attestation for large generative models by handling large datasets residing outside of TEE boundary and incorporating TEE-enabled GPUs. VerifiableFL [58] introduces an integrity-focused enclave architecture that produces authenticated evidence of training inputs and outputs in federated learning deployments. Similarly, Rattanavipanon and Nunes propose stateful proofs of execution that allow an external verifier to confirm correct data usage, model updates, and local differential privacy guarantees on TrustZone-enabled devices [140]. At a broader scope, *Atlas* verifies operations across the ML lifecycle by having TEE-equipped provers attest artifact transformations and publish the resulting evidence to a transparency log [159]. Collectively, these systems demonstrate the effectiveness of TEEs for attestation of ML operations, but two gaps remain that are addressed by this thesis.

Firstly, none of these works focus on attesting properties of generative models while accounting for challenges of their computing environments. PAL*M (presented in Chapter 3) addresses this by providing a framework that is tailored towards large generative models, and accounts for their unique challenges posed by need for accelerators and externally-stored datasets. Secondly, most are designed for conventional ML models, federated learning settings, or specific assumptions about the execution environment that do not account for deviations in expected run-time behavior. EVOLVE (presented in Chapter 4) accounts for this through an extension to TD-Shim that imposes intra-CVM access control, monitors bare-metal containerized ML applications at run-time, and provides evidence to verifiers that reported outputs were indeed only influenced by the expected operation within the application.

5.3 PoX Architectures

Flicker [102] is one of the first architectures to propose a PoX-like mechanism. Flicker leverages late launch and trusted platform modules to enable security-sensitive code to execute in complete isolation from the remaining system. Over the recent years, several works have explored architectures towards enabling PoX on low-end microcontroller units (MCU) [22, 120, 110]. For example, APEX [120] is a formally-verified architecture that enables proofs of execution for cyber-physical systems under full software compromise. It is obtained through a hardware-design that is formally verified with linear temporal logic (LTL) to ensure PoX evidence is always invalidated if the executable is corrupted or interrupted, if its output is modified by unauthorized software, or if metadata used by

the monitoring hardware or attestation software is corrupted. EVOLVE is influenced by APEX as a foundational work towards enabling low-cost PoX, and EVOLVE uses similar terminology to describe memory regions holding the attested software and its outputs (e.g., operation section and O). ASAP [22] extends APEX to enable PoX for asynchronous software in the operation section, allowing for Vrf to obtain PoX even if the *op* in operation section uses interrupts, which would violate the enforcement from APEX. PEARTS [110] introduces the notion of *real-time proofs of execution* that preserves execution integrity while enabling real-time scheduling. PEARTS is the first PoX system that is compatible with real-time operating system FreeRTOS and demonstrates potential towards extending concepts from APEX into off-the-shelf devices with more complex software stacks. EVOLVE brings these concepts to CVMs by providing a lightweight extension to TD-Shim, which supports containerized workloads inside Intel TDX TD.

Chapter 6

Conclusion

In this thesis, I focus on addressing two limitations of current TEEs-based approaches to machine learning property attestations: (1) these frameworks do not scale to large generative models and (2) their reliance on static attestation, which leaves run-time attacks undetected. To address these, this thesis presents two contributions.

First, I present PAL*M: a comprehensive framework capable of spanning CPU–GPU environments, large-scale data, and emerging generative models. We define how to measure properties of generative models and use *incremental multiset hash* functions to capture properties across dataset operations and training. As a result, PAL*M enables accountability for modern AI systems, laying the groundwork for regulators, developers, and verifiers to concretely reason about a machine learning system's trustworthiness.

Second, I present EVOLVE: a proof-of-execution mechanism for confidential virtual machines that enables run-time property attestation of ML operations. Through its instrumentation and monitor components, EVOLVE ensures that property attestation evidence is enhanced with additional data about the provenance of reported outputs at operation-level granularity.

Looking ahead, several directions remain open for future work. First, extending property attestations to distributed settings where inference or training spans multiple CPUs or GPUs across machines requires new attestation chaining mechanisms that can compose evidence across hardware boundaries. Second, modern AI deployments increasingly rely on complex pipelines such as agentic systems, where multiple models interact sequentially or in parallel, and attesting properties of such pipelines requires reasoning about the composition of individual attestations across multi-step, multi-model executions. Third, extending

run-time attestation to large generative models, in both single CPU–GPU setting and distributed settings. Together, these directions point toward a broader vision of trustworthy and auditable AI systems that can be held accountable not just for what they were trained on, but for what they actually do at every step of deployment.

References

[1] Kasra Abbaszadeh et al. Zero-knowledge proofs of training for deep neural networks. In *ACM SIGSAC Conference on Computer and Communications Security*, CCS '24, page 4316–4330, New York, NY, USA, 2024. Association for Computing Machinery.

[2] Tigist Abera, N Asokan, Lucas Davi, Jan-Erik Ekberg, Thomas Nyman, Andrew Paverd, Ahmad-Reza Sadeghi, and Gene Tsudik. C-FLAT: control-flow attestation for embedded systems software. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 743–754, 2016.

[3] Abdelrahman Abouelenin et al. Phi-4-Mini Technical Report: Compact yet Powerful Multimodal Language Models via Mixture-of-LoRAs. *CoRR*, abs/2503.01743, 2025.

[4] EU Artificial Intelligence Act. The EU Artificial Intelligence Act. *European Union*, 2024.

[5] Erdem Aktas et al. Intel Trust Domain Extensions (TDX) security review. *Google security review*, 2023.

[6] AMD. Secure Virtual Machine Architecture Reference Manual. https://www.0x04.net/doc/amd/33047.pdf, 2005. Secure Startup with SKINIT, Chapter 3.1.

[7] AMD. AMD secure encrypted virtualization (sev). https://www.amd.com/en/developer/sev.html, 2020.

[8] Anonymous. Anonymized Github Repository for PAL*M Prototype. https://github.com/AnonymousAuthors552/PALM, January 2026.

[9] Anthropic and Pattern Labs. Confidential inference via trusted virtual machines. https://www.anthropic.com/research/confidential-inference-trusted-vms, June 2025. Research report on confidential inference systems using trusted virtual machines.

[10] Daiyaan Arfeen et al. PipeFill: Using GPUs During Bubbles in Pipeline-parallel LLM Training. In *MLSys*. OpenReview.net/mlsys.org, 2025.

[11] Arm. Arm TrustZone for Cortex-A. https://www.arm.com/technologies/trustzone-for-cortex-a, 2004.

[12] Fabrice Bellard et al. QEMU, a fast and portable dynamic translator. In *Usenix ATC, Freenix Track*, pages 41–46, 2005.

[13] Tyler Bletsch, Xuxian Jiang, Vince W Freeh, and Zhenkai Liang. Jump-oriented programming: a new class of code-reuse attack. In *Proceedings of the 6th ACM symposium on information, computer and communications security*, pages 30–40, 2011.

[14] Ondrej Bojar et al. Findings of the 2014 Workshop on Statistical Machine Translation. In *Proceedings of the Ninth Workshop on Statistical Machine Translation*, pages 12–58, Baltimore, Maryland, USA, June 2014. Association for Computational Linguistics.

[15] Dan Boneh, Ben Lynn, and Hovav Shacham. Short signatures from the Weil pairing. In *International conference on the theory and application of cryptology and information security*, pages 514–532. Springer, 2001.

[16] Kjell Braden, Lucas Davi, Christopher Liebchen, Ahmad-Reza Sadeghi, Stephen Crane, Michael Franz, and Per Larsen. Leakage-resilient layout randomization for mobile devices. In *NDSS*, volume 16, pages 21–24, 2016.

[17] Sergey Bratus et al. TOCTOU, traps, and trusted computing. In *International Conference on Trusted Computing*, pages 14–32. Springer, 2008.

[18] Brave Software. Verifiable privacy and transparency: A new frontier for brave ai privacy. https://brave.com/blog/browser-ai-tee/, 11 2025. Accessed: 2025-11-26.

[19] Daniel R. L. Brown. The Encrypted Elliptic Curve Hash. Cryptology ePrint Archive, Paper 2008/012, 2008.

[20] Tom Brown et al. Language Models are Few-Shot Learners. In *Advances in Neural Information Processing Systems*, volume 33, pages 1877–1901. Curran Associates, Inc., 2020.

[21] Canonical. Canonical TDX Github Repository. https://github.com/canonical/tdx, September 2025.

[22] Adam Caulfield, Norrathep Rattanavipanon, and Ivan De Oliveira Nunes. ASAP: reconciling asynchronous real-time operations and proofs of execution in simple embedded systems. In *Proceedings of the 59th ACM/IEEE Design Automation Conference*, pages 721–726, 2022.

[23] Prach Chantasantitam, Adam Ilyas Caulfield, Vasisht Duddu, Lachlan J Gunn, and N Asokan. PAL*M: Property Attestation for Large Generative Models. *arXiv preprint arXiv:2601.16199*, 2026.

[24] Jorge Chávez-Saab et al. SwiftEC: Shallue–van de woestijne indifferentiable function to elliptic curves: Faster indifferentiable hashing to elliptic curves, 2025.

[25] Lingjiao Chen, Matei Zaharia, and James Zou. Frugalgpt: How to use large language models while reducing cost and improving performance. *arXiv preprint arXiv:2305.05176*, 2023.

[26] Shuo Chen, Jun Xu, Emre Can Sezer, Prachi Gauriar, and Ravishankar K Iyer. Non-control-data attacks are realistic threats. In *USENIX security symposium*, volume 5, page 146, 2005.

[27] Long Cheng, Salman Ahmed, Hans Liljestrand, Thomas Nyman, Haipeng Cai, Trent Jaeger, N Asokan, and Danfeng Yao. Exploitation techniques for data-oriented attacks with existing and potential defense approaches. *ACM Transactions on Privacy and Security (TOPS)*, 24(4):1–36, 2021.

[28] Long Cheng, Salman Ahmed, Hans Liljestrand, Thomas Nyman, Haipeng Cai, Trent Jaeger, N Asokan, and Danfeng Yao. Exploitation techniques for data-oriented attacks with existing and potential defense approaches. *ACM Transactions on Privacy and Security (TOPS)*, 24(4):1–36, 2021.

[29] Jack Choquette. NVIDIA Hopper H100 GPU: Scaling Performance. *IEEE Micro*, 43(3):9–17, 2023.

[30] Marcin Chrapek et al. Confidential LLM inference: Performance and cost across CPU and GPU TEEs. *arXiv preprint arXiv:2509.18886*, 2025.

[31] Jalen Chuang, Alex Seto, Nicolas Berrios, Stephan van Schaik, Christina Garman, and Daniel Genkin. TEE.fail: Breaking Trusted Execution Environments via DDR5

Memory Bus Interposition. In *47th IEEE Symposium on Security and Privacy (IEEE S&P '26)*. IEEE Computer Society, 2026.

[32] Dwaine Clarke et al. Incremental multiset hash functions and their application to memory integrity checking. In *International conference on the theory and application of cryptology and information security*, pages 188–207. Springer, 2003.

[33] David Cohen. Data Scientists Targeted by Malicious Hugging Face ML Models with Silent Backdoor. https://jfrog.com/blog/data-scientists-targeted-by-malicious-hugging-face-ml-models-with-silent-backdoor/, 2024.

[34] Ron Cole and Mark Fanty. ISOLET. UCI Machine Learning Repository, 1991. DOI: https://doi.org/10.24432/C51G69.

[35] Confidential Containers. TD-Shim Threat Model and Secure Design. https://github.com/confidential-containers/td-shim/blob/main/doc/threat_model.md, 2022.

[36] Confidential Containers. TD Shim specification (version 0.7). https://github.com/confidential-containers/td-shim/blob/main/doc/tdshim_spec.md, 2023.

[37] Scott Constable, Jo Van Bulck, Xiang Cheng, Yuan Xiao, Cedric Xing, Ilya Alexandrovich, Taesoo Kim, Frank Piessens, Mona Vij, and Mark Silberstein. {AEX-Notify}: Thwarting precise {Single-Stepping} attacks through interrupt awareness for intel {SGX} enclaves. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 4051–4068, 2023.

[38] Michael Han Daniel Han and Unsloth team. Unsloth. https://github.com/unslothai/unsloth, 2023.

[39] Dujian Ding, Ankur Mallick, Chi Wang, Robert Sim, Subhabrata Mukherjee, Victor Rühle, Laks Lakshmanan, and Ahmed H Awadallah. Hybrid LLM: Cost-Efficient and Quality-Aware Query Routing. In *International Conference on Learning Representations*, volume 2024, pages 41348–41366, 2024.

[40] Vasisht Duddu et al. Attesting distributional properties of training data for machine learning. In *European Symposium on Research in Computer Security*, pages 3–23. Springer, 2024.

[41] Vasisht Duddu, Lachlan J Gunn, and N Asokan. Laminator: Verifiable ML property cards using hardware-assisted attestations. In *Proceedings of the Fifteenth ACM Conference on Data and Application Security and Privacy*, pages 317–328, 2024.

[42] Hossam ElAtali, Lachlan J. Gunn, Hans Liljestrand, and N. Asokan. BliMe: Verifiably secure outsourced computation with hardware-enforced taint tracking. In *31st Annual Network and Distributed System Security Symposium, NDSS 2024, San Diego, California, USA, February 26 - March 1, 2024*. The Internet Society, 2024.

[43] Hugging Face. Loading big models into memory. https://huggingface.co/docs/accelerate/concept_guides/big_model_inference, [Accessed] 2026.

[44] Congyu Fang et al. Proof-of-learning is currently more broken than you think. In *2023 IEEE 8th European Symposium on Security and Privacy (EuroS&P)*, pages 797–816. IEEE, 2023.

[45] Armando Faz-Hernandez, Sam Scott, Nick Sullivan, Riad S. Wahby, and Christopher A. Wood. Hashing to Elliptic Curves. RFC 9380, August 2023.

[46] Aurélien Francillon and Claude Castelluccia. Code injection attacks on harvard-architecture devices. In *Proceedings of the 15th ACM conference on Computer and communications security*, pages 15–26, 2008.

[47] Olive Franzese et al. Secure and confidential certificates of online fairness. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*, 2025.

[48] Olive Franzese et al. Secure noise sampling for differentially private collaborative learning. In *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security*, pages 4649–4663, 2025.

[49] Prakhar Ganesh et al. On the impact of machine learning randomness on group fairness. In *Proceedings of the 2023 ACM Conference on Fairness, Accountability, and Transparency*, pages 1789–1800, 2023.

[50] Leo Gao et al. The Pile: An 800GB Dataset of Diverse Text for Language Modeling. *arXiv preprint arXiv:2101.00027*, 2020.

[51] Leo Gao et al. The Language Model Evaluation Harness. https://zenodo.org/records/12608602, July 2024.

[52] Sanjam Garg et al. Experimenting with Zero-Knowledge Proofs of Training. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, CCS '23, page 1880–1894, New York, NY, USA, 2023. Association for Computing Machinery.

[53] Timnit Gebru, Jamie Morgenstern, Briana Vecchione, Jennifer Wortman Vaughan, Hanna Wallach, Hal Daumé Iii, and Kate Crawford. Datasheets for Datasets. *Communications of the ACM*, 64(12):86–92, 2021.

[54] ggml org. Llama.cpp github repository. https://github.com/ggml-org/llama.cpp/blob/master/tools/server/README.md, [Accessed] 2026.

[55] Google. Confidential VM pricing — Google Cloud. https://cloud.google.com/confidential-computing/confidential-vm/pricing, [Accessed] 2026.

[56] Zhongshu Gu, Enriquillo Valdez, Salman Ahmed, Julian James Stephen, Michael Le, Hani Jamjoom, Shixuan Zhao, and Zhiqiang Lin. Nvidia gpu confidential computing demystified. *arXiv preprint arXiv:2507.02770*, 2025.

[57] Neel Guha, Mayee F Chen, Trevor Chow, Ishan S Khare, and Christopher Re. Smoothie: Label free language model routing. *Advances in Neural Information Processing Systems*, 37:127645–127672, 2024.

[58] Jinnan Guo et al. ExclaveFL: Providing transparency to federated learning using exclaves. *arXiv preprint arXiv:2412.10537*, 2024.

[59] Casper Hansen. AutoAWQ. https://github.com/casper-hansen/AutoAWQ, 2023.

[60] Dan Hendrycks et al. Measuring Massive Multitask Language Understanding. In *ICLR*. OpenReview.net, 2021.

[61] Jonathan Ho, Ajay Jain, and Pieter Abbeel. Denoising Diffusion Probabilistic Models. In *NeurIPS*, 2020.

[62] Edward J. Hu et al. LoRA: Low-Rank Adaptation of Large Language Models. In *ICLR*. OpenReview.net, 2022.

[63] Hong Hu, Shweta Shinde, Sendroiu Adrian, Zheng Leong Chua, Prateek Saxena, and Zhenkai Liang. Data-oriented programming: On the expressiveness of non-control data attacks. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 969–986. IEEE, 2016.

[64] Yaqi Hu et al. LLM-based misbehavior detection architecture for enhanced traffic safety in connected autonomous vehicles. *IEEE Transactions on Vehicular Technology*, 2025.

[65] Chenyu Huang et al. zkMLaaS: a verifiable scheme for machine learning as a service. In *GLOBECOM 2022-2022 IEEE Global Communications Conference*, pages 5475–5480. IEEE, 2022.

[66] Hugging Face. Datasets Arrow. https://huggingface.co/docs/datasets/v4.1.1/about_arrow, 2025. Accessed: 2025-10-01.

[67] Hugging Face. Preprocess. https://huggingface.co/docs/datasets/en/use_dataset, 2025. Accessed: 2025-01-19.

[68] Hugging Face. Trainer. https://huggingface.co/docs/transformers/main_classes/trainer#transformers.Trainer.get_train_dataloader, 2025. Accessed: 2025-01-15.

[69] Hugging Face. Datasets. https://huggingface.co/docs/datasets/index, 2026. Accessed: 2026-03-12.

[70] LLVM Compiler Infrastructure. LLVM language reference manual. https://llvm.org/docs/LangRef.html, [Accessed] 2026.

[71] LLVM Compiler Infrastructure. LLVM language reference manual: '`allooca`' instruction. https://llvm.org/docs/LangRef.html#alloca-instruction, [Accessed] 2026.

[72] LLVM Compiler Infrastructure. LLVM language reference manual: '`call`' instruction. https://llvm.org/docs/LangRef.html#call-instruction, [Accessed] 2026.

[73] LLVM Compiler Infrastructure. LLVM language reference manual: '`store`' instruction. https://llvm.org/docs/LangRef.html#store-instruction, [Accessed] 2026.

[74] LLVM Compiler Infrastructure. LLVM user guides: Using the new pass manager. https://llvm.org/docs/NewPassManager.html, [Accessed] 2026.

[75] Intel. Intel Provisining Certification Service for ECDSA Attestation. https://api.portal.trustedservices.intel.com/provisioning-certification.

[76] Intel. Trust Domain Integrity Overview.

[77] Intel. Intel Software Guard Extensions (Intel SGX). https://www.intel.com/content/www/us/en/developer/tools/software-guard-extensions/overview.html, 2015.

[78] Intel. Complex Shadow-Stack Updates (Intel Control-Flow Enforcement Technology). https://www.intel.com/content/www/us/en/content-details/785687/complex-shadow-stack-updates-intel-control-flow-enforcement-technology.html, 2023.

[79] Intel. Intel Trust Domain Extensions (Intel TDX). https://www.intel.com/content/www/us/en/developer/tools/trust-domain-extensions/overview.html, 2023.

[80] Intel. Intel Trust Domain Extensions (Intel TDX) module base architecture specification. https://cdrdv2-public.intel.com/853286/intel-tdx-module-base-spec-348549006.pdf, Apr 2025.

[81] Intel. Intel Trust Domain Extensions Module Base Architecture Specification. https://www.intel.com/content/www/us/en/developer/tools/trust-domain-extensions/documentation.html, [Accessed] 2026.

[82] Intel. Running Average Power Limit Energy Reporting CVE-2020-8694, CVE-2020-8695 INTEL-SA-00389. https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/advisory-guidance/running-average-power-limit-energy-reporting.html, [Accessed] 2026.

[83] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 1: Basic Architecture*. Intel Corporation, 2026. Chapter 18: Control-flow Enforcement Technology (CET).

[84] Insu Jang et al. Heterogeneous isolated execution for commodity GPUs. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 455–468, 2019.

[85] Yukun Jiang, Hai Huang, Mingjie Li, Yage Zhang, Michael Backes, and Yang Zhang. Sparse Models, Sparse Safety: Unsafe Routes in Mixture-of-Experts LLMs. *arXiv preprint arXiv:2602.08621*, 2026.

[86] Markelle Kelly, Rachel Longjohn, and Kolby Nottingham. The UCI Machine Learning Repository. https://archive.ics.uci.edu.

[87] Keras. Keras repository. https://github.com/keras-team/keras, [Accessed] 2025.

[88] Yonghae Kim, Jaekyu Lee, and Hyesoon Kim. Hardware-based always-on heap memory safety. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1153–1166. IEEE, 2020.

[89] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *International symposium on code generation and optimization, 2004. CGO 2004.*, pages 75–86. IEEE, 2004.

[90] Woomin Lee, Taehun Kim, Seunghee Shin, Junbeom Hur, and Youngjoo Shin. T-Time: A Fine-Grained Timing-Based Controlled-Channel Attack Against Intel TDX. In *European Symposium on Research in Computer Security*, pages 323–341. Springer, 2025.

[91] Hugo Lefeuvre, Nathan Dautenhahn, David Chisnall, and Pierre Olivier. SoK: Software Compartmentalization. In *2025 IEEE Symposium on Security and Privacy (SP)*, pages 3107–3126. IEEE, 2025.

[92] Arm Limited. Overview: Realms. https://learn.arm.com/learning-paths/servers-and-cloud-computing/cca-container/overview/.

[93] Qiqi Lin, Xiaoyang Ji, Shengfang Zhai, Qingni Shen, Zhi Zhang, Yuejian Fang, and Yansong Gao. Life-Cycle Routing Vulnerabilities of LLM Router. *arXiv preprint arXiv:2503.08704*, 2025.

[94] Tianyi Liu, Xiang Xie, and Yupeng Zhang. zkCNN: Zero knowledge proofs for convolutional neural network predictions and accuracy. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 2968–2985, 2021.

[95] Ziyu Liu et al. MirrorNet: A TEE-friendly framework for secure on-device DNN inference. In *2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD)*, pages 1–9. IEEE, 2023.

[96] Hidde Lycklama et al. Holding secrets accountable: Auditing privacy-preserving machine learning. In *USENIX Security Symposium*, 2024.

[97] Zheyuan Ma, Xi Tan, Lukasz Ziarek, Ning Zhang, Hongxin Hu, and Ziming Zhao. Return-to-Non-Secure Vulnerabilities on ARM Cortex-M TrustZone: Attack and Defense. In *2023 60th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2023.

[98] Jeremy Maitin-Shepard, Mehdi Tibouchi, and Diego F Aranha. Elliptic curve multiset hash. *The computer journal*, 60(4):476–490, 2017.

[99] Upasana Mandal, Shubhi Shukla, Nimish Mishra, Sarani Bhattacharya, Paritosh Saxena, and Debdeep Mukhopadhyay. Exploring side-channels in Intel Trust Domain Extensions. Cryptology ePrint Archive, Paper 2025/079, 2025.

[100] Sourab Mangrulkar et al. PEFT: State-of-the-art Parameter-Efficient Fine-Tuning methods. https://github.com/huggingface/peft, 2022.

[101] Ali Jose Mashtizadeh, Andrea Bittau, Dan Boneh, and David Mazières. Ccfi: Cryptographically enforced control flow integrity. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 941–951, 2015.

[102] Jonathan M McCune, Bryan J Parno, Adrian Perrig, Michael K Reiter, and Hiroshi Isozaki. Flicker: An execution infrastructure for tcb minimization. In *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008*, pages 315–328, 2008.

[103] Simon Meier, Benedikt Schmidt, Cas Cremers, and David Basin. The TAMARIN prover for the symbolic analysis of security protocols. In *International conference on computer aided verification*, pages 696–701. Springer, 2013.

[104] Microsoft. NC family VM size series - Azure Virtual Machines — Microsoft Learn. https://learn.microsoft.com/en-us/azure/virtual-machines/sizes/gpu-accelerated/nc-family, [Accessed] 2026.

[105] Boyan Milanov. Exploiting ML models with pickle file attacks: Part 1. https://blog.trailofbits.com/2024/06/11/exploiting-ml-models-with-pickle-file-attacks-part-1/, 2024.

[106] Nimish Mishra, Kislay Arya, Sarani Bhattacharya, Paritosh Saxena, and Debdeep Mukhopadhyay. "OOPS!": Out-Of-Band Remote Power Side-Channel Attacks on Intel SGX and TDX. In *2025 62nd ACM/IEEE Design Automation Conference (DAC)*, pages 1–7. IEEE, 2025.

[107] Margaret Mitchell, Simone Wu, Andrew Zaldivar, Parker Barnes, Lucy Vasserman, Ben Hutchinson, Elena Spitzer, Inioluwa Deborah Raji, and Timnit Gebru. Model Cards for Model Reporting. In *Proceedings of the conference on fairness, accountability, and transparency*, pages 220–229, 2019.

[108] Fan Mo et al. DarkneTZ: towards model privacy at the edge using trusted execution environments. In *International Conference on Mobile Systems, Applications, and Services*, MobiSys '20, page 161–174, New York, NY, USA, 2020. Association for Computing Machinery.

[109] Santosh Nagarakatte, Jianzhou Zhao, Milo MK Martin, and Steve Zdancewic. Softbound: Highly compatible and complete spatial memory safety for c. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 245–258, 2009.

[110] Antonio Joia Neto, Norrathep Rattanavipanon, and Ivan De Oliveira Nunes. PEARTS: Provable execution in real-time embedded systems. In *2025 IEEE Symposium on Security and Privacy (SP)*, pages 3765–3782. IEEE, 2025.

[111] NIST National Vulnerability Database (NVD). CVE-2022-29216 Detail. https://nvd.nist.gov/vuln/detail/CVE-2022-29216, 2022.

[112] NIST National Vulnerability Database (NVD). CVE-2022-459076 Detail. https://nvd.nist.gov/vuln/detail/CVE-2022-45907, 2022.

[113] NIST National Vulnerability Database (NVD). CVE-2022-45908 Detail. https://nvd.nist.gov/vuln/detail/CVE-2022-45908, 2022.

[114] NIST National Vulnerability Database (NVD). CVE-2022-46742 Detail. https://nvd.nist.gov/vuln/detail/CVE-2022-46742, 2022.

[115] NIST National Vulnerability Database (NVD). CVE-2023-43654 Detail. https://nvd.nist.gov/vuln/detail/cve-2023-43654, 2023.

[116] NIST National Vulnerability Database (NVD). CVE-2025-32434 Detail. https://nvd.nist.gov/vuln/detail/CVE-2025-32434, 2025.

[117] NIST National Vulnerability Database (NVD). CVE-2025-32434 Detail. https://nvd.nist.gov/vuln/detail/CVE-2026-24747, 2026.

[118] Jon Nordby, Mark Cooke, and Adam Horvath. emlearn: Machine Learning inference engine for Microcontrollers and Embedded Devices, March 2019.

[119] Nadav Noy. Legit Discovers "AI Jacking" Vulnerability in Popular Hugging Face AI Platform. https://www.legitsecurity.com/blog/tens-of-thousands-of-developers-were-potentially-impacted-by-the-hugging-face-aijacking-attack, 2023.

[120] Ivan De Oliveira Nunes, Karim Eldefrawy, Norrathep Rattanavipanon, and Gene Tsudik. APEX: A verified architecture for proofs of execution on remote devices under full software compromise. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 771–788, 2020.

[121] Ivan De Oliveira Nunes et al. Toward Remotely Verifiable Software Integrity in Resource-Constrained IoT Devices. *IEEE Communications Magazine*, 62(7):58–64, 2024.

[122] NVIDIA. AI Security With Confidential Computing. https://www.nvidia.com/en-us/data-center/solutions/confidential-computing/.

[123] NVIDIA. Llm router blueprint. https://github.com/NVIDIA-AI-Blueprints/llm-router, 2025.

[124] NVIDIA. NVIDIA Deployment Guide for SecureAI. https://docs.nvidia.com/cc-deployment-guide-tdx.pdf, February 2025.

[125] NVIDIA. Triton inference server. https://github.com/triton-inference-server/server, 2025.

[126] NVIDIA. NVIDIA attestation. https://docs.nvidia.com/attestation/index.html, [Accessed] 2025.

[127] NVIDIA. NVIDIA nvos user manual for nvlink switches. https://docs.nvidia.com/networking/display/nvidianvosusermanualfornvlinkswitchesv25024282/spdm, [Accessed] 2026.

[128] Isaac Ong, Amjad Almahairi, Vincent Wu, Wei-Lin Chiang, Tianhao Wu, Joseph E Gonzalez, M Waleed Kadous, and Ion Stoica. RouteLLM: Learning to Route LLMs with Preference Data. *arXiv preprint arXiv:2406.18665*, 2024.

[129] OpenAI. Reimagining secure infrastructure for advanced ai. https://openai.com/index/reimagining-secure-infrastructure-for-advanced-ai/, 5 2024. Accessed: 2025-11-26.

[130] Meni Orenbach, Yan Michalevsky, Christof Fetzer, and Mark Silberstein. {CoSMIX}: A compiler-based system for secure memory instrumentation and execution in enclaves. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 555–570, 2019.

[131] PaddlePaddle. Paddle repository. https://github.com/PaddlePaddle/Paddle, [Accessed] 2025.

[132] Tapti Palit, Fabian Monrose, and Michalis Polychronakis. Mitigating data leakage by protecting memory-resident sensitive data. In *Proceedings of the 35th annual computer security applications conference*, pages 598–611, 2019.

[133] Kishore Papineni. BLEU: a Method for Automatic Evaluation of Machine Translation. In *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*, pages 311–318, 2002.

[134] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[135] Mahima Pushkarna, Andrew Zaldivar, and Oddur Kjartansson. Data Cards: Purposeful and Transparent Dataset Documentation for Responsible AI. In *Proceedings of the 2022 ACM conference on fairness, accountability, and transparency*, pages 1776–1826, 2022.

[136] PyTorch. torch.utils.data — PyTorch 2.9 documentation. https://docs.pytorch.org/docs/stable/data.html, 2025. Accessed: 2026-01-15.

[137] PyTorch. Pytorch repository. https://github.com/pytorch/pytorch, [Accessed] 2025.

[138] Wenjie Qu, Yijun Sun, Xuanming Liu, Tao Lu, Yanpei Guo, Kai Chen, and Jiaheng Zhang. {zkGPT}: An efficient non-interactive zero-knowledge proof framework for {LLM} inference. In *34th USENIX Security Symposium (USENIX Security 25)*, pages 2045–2063, 2025.

[139] Alec Radford et al. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.

[140] Norrathep Rattanavipanon and Ivan De Oliveira Nunes Nunes. SLAPP: Poisoning prevention in federated learning and DP via stateful proofs of execution. *IEEE Transactions on Information Forensics and Security (TIFS)*, 2025.

[141] Fabian Rauscher, Luca Wilke, Hannes Weissteiner, Thomas Eisenbarth, and Daniel Gruss. TDXploit: Novel Techniques for Single-Stepping and Cache Attacks on Intel TDX. In *USENIX Secuirty 2025*, 2025.

[142] Siva Reddy, Danqi Chen, and Christopher D. Manning. CoQA: A conversational question answering challenge. *Transactions of the Association for Computational Linguistics*, 7:249–266, 2019.

[143] Herbert Robbins and Sutton Monro. A stochastic approximation method. *The annals of mathematical statistics*, pages 400–407, 1951.

[144] Ryan Roemer, Erik Buchanan, Hovav Shacham, and Stefan Savage. Return-oriented programming: Systems, languages, and applications. *ACM Transactions on Information and System Security (TISSEC)*, 15(1):1–34, 2012.

[145] Bidhan Roy, Peter Potash, and Marcos Villagra. ZKLoRA: Efficient zero-knowledge proofs for LoRA verification. In *Championing Open-source DEvelopment in ML Workshop @ ICML25*, 2025.

[146] Ahmad-Reza Sadeghi and Christian Stüble. Property-based attestation for computing platforms: caring about properties, not mechanisms. In *Proceedings of the 2004 workshop on New security paradigms*, pages 67–77, 2004.

[147] Vinnie Scarlata, Simon Johnson, James Beaney, and Piotr Zmijewski. Supporting third party attestation for Intel SGX with Intel data center attestation primitives. *Intel Corporation, White paper*, 12, 2018.

[148] Christoph Schnabl et al. Attestable audits: Verifiable AI safety benchmarks using trusted execution environments. In *ICML Workshop on Technical AI Governance (TAIG)*, 2025.

[149] Felix Schuster, Thomas Tendyck, Christopher Liebchen, Lucas Davi, Ahmad-Reza Sadeghi, and Thorsten Holz. Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in C++ applications. In *2015 IEEE Symposium on Security and Privacy*, pages 745–762. IEEE, 2015.

[150] Michael Schwarz, Samuel Weiser, Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Malware guard extension: Using SGX to conceal cache attacks. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 3–24. Springer, 2017.

[151] David Sehr, Robert Muth, Cliff Biffle, Victor Khimenko, Egor Pasko, Karl Schimpf, Bennet Yee, and Brad Chen. Adapting software fault isolation to contemporary {CPU} architectures. In *19th USENIX Security Symposium (USENIX Security 10)*, 2010.

[152] Hovav Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM conference on Computer and communications security*, pages 552–561, 2007.

[153] Avital Shafran, Roei Schuster, Thomas Ristenpart, and Vitaly Shmatikov. Rerouting LLM Routers. *arXiv preprint arXiv:2501.01818*, 2025.

[154] Ali Shahin Shamsabadi et al. Confidential-PROFITT: Confidential PROof of fair training of trees. In *The Eleventh International Conference on Learning Representations*, 2023.

[155] Ali Shahin Shamsabadi et al. Confidential-DPproof: Confidential proof of differentially private training. In *The Twelfth International Conference on Learning Representations*, 2024.

[156] Youren Shen, Hongliang Tian, Yu Chen, Kang Chen, Runji Wang, Yi Xu, Yubin Xia, and Shoumeng Yan. Occlum: Secure and Efficient Multitasking Inside a Single Enclave of Intel SGX. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 955–970, 2020.

[157] Ying Sheng et al. FlexGen: High-Throughput Generative Inference of Large Language Models with a Single GPU. In *Proceedings of the 40th International Conference on Machine Learning*, volume 202 of *Proceedings of Machine Learning Research*, pages 31094–31116. PMLR, 23–29 Jul 2023.

[158] Tal Shnitzer, Anthony Ou, Mirian Silva, Kate Soule, Yuekai Sun, Justin Solomon, Neil Thompson, and Mikhail Yurochkin. Large language model routing with benchmark datasets. In *Annual Conference on Neural Information Processing Systems*, 2023.

[159] Marcin Spoczynski, Marcela S. Melara, and Sebastian Szyller. Atlas: A framework for ML lifecycle provenance & transparency. In *2025 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*, pages 448–461. IEEE, 2025.

[160] Haochen Sun, Jason Li, and Hongyang Zhang. zkLLM: Zero knowledge proofs for large language models. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, pages 4405–4419, 2024.

[161] Tong Sun et al. TensorShield: Safeguarding on-device inference by shielding critical DNN tensors with TEE. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2025.

[162] Haochun Tang, Yuliang Yan, Jiahua Lu, Huaxiao Liu, and Enyan Dai. Route to Rome Attack: Directing LLM Routers to Expensive Models via Adversarial Suffix Optimization. *arXiv preprint arXiv:2604.15022*, 2026.

[163] Rohan Taori et al. Stanford Alpaca: An instruction-following llama model. https://github.com/tatsu-lab/stanford_alpaca, 2023.

[164] Gemma Team. Gemma 3 Technical Report. *CoRR*, abs/2503.19786, 2025.

[165] Llama Team. The Llama 3 Herd of Models. *CoRR*, abs/2407.21783, 2024.

[166] Tensorflow. Tensorflow repository. https://github.com/tensorflow/tensorflow, [Accessed] 2025.

[167] Caroline Tice, Tom Roeder, Peter Collingbourne, Stephen Checkoway, Úlfar Erlingsson, Luis Lozano, and Geoff Pike. Enforcing {Forward-Edge}{Control-Flow} integrity in {GCC} & {LLVM}. In *23rd USENIX security symposium (USENIX security 14)*, pages 941–955, 2014.

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

[169] Ehsan Ullah et al. Challenges and barriers of using large language models (LLM) such as ChatGPT for diagnostic medicine with a focus on digital pathology–a recent scoping review. *Diagnostic pathology*, 19(1):43, 2024.

[170] Daan Vanoverloop, Andres Sanchez, Flavio Toffalini, Frank Piessens, Mathias Payer, and Jo Van Bulck. {TLBlur}:{Compiler-Assisted} automated hardening against

controlled channels on {Off-the-Shelf} intel {SGX} platforms. In *34th USENIX Security Symposium (USENIX Security 25)*, pages 1167–1186, 2025.

[171] Ashish Vaswani et al. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[172] Lilian Weng and Greg Brockman. Techniques for training large neural networks. https://openai.com/index/techniques-for-training-large-neural-networks/, 9 June 2022.

[173] Luca Wilke, Florian Sieck, and Thomas Eisenbarth. Tdxdown: Single-stepping and instruction counting attacks against intel tdx. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, pages 79–93, 2024.

[174] William Wolberg, Olvi Mangasarian, Nick Street, and W. Street. Breast Cancer Wisconsin (Diagnostic). UCI Machine Learning Repository, 1993. DOI: https://doi.org/10.24432/C5DW2B.

[175] Yangyang Yu et al. Fincon: A synthesized LLM multi-agent system with conceptual verbal reinforcement for enhanced financial decision making. *Advances in Neural Information Processing Systems*, 37:137010–137045, 2024.

[176] Jiashu Zhang et al. Freeride: Harvesting bubbles in pipeline parallelism. In *Proceedings of the 26th International Middleware Conference*, pages 126–139, 2025.

[177] Rui Zhang et al. "Adversarial examples" for proof-of-learning. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 1408–1422. IEEE, 2022.

[178] Ziqi Zhang et al. No privacy left outside: On the (in-) security of TEE-shielded DNN partition for on-device ML. In *2024 IEEE Symposium on Security and Privacy (SP)*, pages 3327–3345. IEEE, 2024.

[179] Jian Zhao, Shenao Wang, Yanjie Zhao, Xinyi Hou, Kailong Wang, Peiming Gao, Yuanchao Zhang, Chen Wei, and Haoyu Wang. Models are codes: Towards measuring malicious code poisoning attacks on pre-trained model hubs. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, pages 2087–2098, 2024.

[180] Yukun Zhu et al. Aligning books and movies: Towards story-like visual explanations by watching movies and reading books. In *Proceedings of the IEEE international conference on computer vision*, pages 19–27, 2015.

Glossary

AI artificial intelligence 77, 81, 82

AWQ Activation-aware Weight Quantization 27

CC Confidential Computing 34

CET control-flow enforcement technology 56, 63, 73, 75

CFI control-flow integrity 63, 64, 73–75

CPU central processing unit 1–5, 7, 12–15, 24, 27, 38, 56, 63, 66, 74, 76, 79, 81, 82

CVM confidential virtual machine 1–5, 7, 11, 13, 18, 34, 35, 51, 55–57, 59, 72, 76, 79, 80

DCAP Data Center Attestation Primitives 27

DDM direct data manipulation 56, 75

DEP data execution prevention 55, 73

DOP data-oriented programming 56, 75

DT decision tree 66, 69–71

ELF executable and linkable format 61, 73

FFI foreign function interface 60, 61, 63, 67, 74, 76

GPU graphics processing unit 1–4, 7, 11–15, 18, 19, 27, 31, 33–35, 38, 40, 55, 75, 77, 79, 81, 82

I/O input/output 9, 12, 28, 55, 56

IBT indirect branch tracking 56, 63, 73, 74

IR intermediate representation 54, 55, 59, 60

LLM large language model 1, 2, 7–9, 12, 15, 19, 26–29, 49, 77, 78

LoRA Low-Rank Adaptation 8, 26

LTL linear temporal logic 79

MAC message authenticated code 5, 24, 33, 36, 38, 40, 53, 65, 74, 76

MCU microcontroller unit 79

ML machine learning 1–4, 6, 9–12, 15, 17, 24, 25, 34, 49, 51, 52, 55–58, 61, 76, 78, 79, 81

MLP multi-layer perceptron 66, 67, 69–71

NPU neural processing unit 13

NRAS Remote Attestation Service 27

OS operating system 4, 15

PEFT Parameter-Efficient Fine-Tuning 8

PoX proof of execution 2, 3, 51, 53, 56, 57, 59, 60, 64, 72–74, 77, 79, 80

Q&A question-and-answer 26

QE quoting enclave 5, 14, 22, 24, 32, 33, 35, 36, 38, 40, 64, 65, 74, 76

RAM random access memory 27, 34, 66

ROP return-oriented programming 56

RoT root of trust 5, 53

SFI software fault isolation 54

SGX Software Guard eXtensions 1, 4–6, 12, 76, 78

SMPC secure multi-party computation 78

SPDM Security Protocol and Data Model 18, 34, 40

SSA static single assignment 54

TCP transmission control protocol 67

TD trust domain 5, 12–15, 18, 22–24, 27, 32–36, 38, 56–58, 64–70, 72, 73, 75, 80

TDX Trust Domain eXtensions 3–5, 7, 11–14, 22–24, 27, 33, 35, 36, 38, 40, 42, 51, 52, 55–59, 61, 64–66, 68, 70, 73, 74, 80

TEE trusted execution environment 1, 2, 4, 5, 7, 11–13, 18, 25, 33, 35, 46, 47, 53, 57, 74, 78, 79, 81

TPM trusted platform module 53

VM virtual machine 4, 12, 56

VMM virtual machine manager 5, 12, 14, 22–24, 32, 33, 35, 38, 55–57, 64, 65, 73

ZKP zero-knowledge proof 1, 78