# MVTEE: Multi-Variant Trusted Execution for Secure Model Inference

Kailun Qin
Shanghai Jiao Tong University
Shanghai, China
kailun.qin@sjtu.edu.cn

Dawu Gu
Shanghai Jiao Tong University
Shanghai, China
dwgu@sjtu.edu.cn

## Abstract

Trusted Execution Environments (TEEs) have been proposed as a promising approach for secure model inference, providing in-use data protection to ensure confidentiality and integrity against untrusted third parties, with additional attestability. However, TEE-protected secure model inference remains susceptible to numerous software vulnerabilities and fault attacks, potentially undermining the designed protection objectives and giving a false sense of security and reliability.

To counter these diverse threats, we introduce MVTEE, a TEE-based model inference system employing Multi-Variant Execution (MVX) that runs multiple, diversified inference variants in parallel and monitors execution divergence at checkpoints. The idea of MVTEE is not to prevent threats, but to leverage the fact that a specific vulnerability typically impacts only one variant, ensuring timely threat detection and response before any damage is done. MVTEE applies a random-balanced model partitioning for checkpoint insertion and leverages the heterogeneous nature of model inference stack to generate variants with multi-level diversification. We base MVTEE on a cross-process monitoring architecture with a two-stage variant bootstrap design and support asynchronous selective MVX for efficient execution. Our evaluations demonstrate that MVTEE, in a real-world setup, secures model inference with acceptable performance overhead in sequential execution and maintains comparable or even improved performance in pipelined execution.

## CCS Concepts

• **Security and privacy → Software and application security**; **Trusted computing**.

## Keywords

Multi-Variant Execution, Trusted Execution Environment, DNN Inference, Software Security, Software Fault Tolerance

**ACM Reference Format:**
Kailun Qin and Dawu Gu. 2025. MVTEE: Multi-Variant Trusted Execution for Secure Model Inference. In *26th ACM Middleware Conference (Middleware '25), December 15–19, 2025, Nashville, TN, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3721462.3730956

*Middleware '25, Nashville, TN, USA*


## 1 Introduction

Major cloud service providers, such as Amazon AWS [1], Microsoft Azure [104], and Google Cloud [9], offer Machine Learning (ML) inference services to reduce ML infrastructure costs [43] and manage large-scale inference queries [35, 36]. These services deploy pre-trained models from model owners and pull user data for inference. Today, threats including malicious tenants, compromised providers, and physical breaches have raised significant concerns. Users demand data privacy, while model owners need to protect their high-value intellectual properties. Further, models in safety-critical applications, like high-performance computing or autonomous vehicles, require protection against tampering to ensure reliable results.

Recently, state-of-the-art research and production cloud services have embraced the widely accessible CPU Trusted Execution Environments (TEEs) for secure ML inference, protecting the confidentiality and integrity of in-use models and data from untrusted third parties, with additional attestability [53, 58, 63, 65, 66, 74, 78, 94]. However, this widespread adoption has also made them more attractive targets for threats that undermine security or reliability:

- Software vulnerabilities: ML frameworks and libraries are increasingly plagued by memory and runtime errors, which can lead to security breaches and incorrect outputs [24, 42, 44, 88, 139]. Unfortunately, these vulnerabilities fall outside of the TEE threat model by design [115]. This places an unrealistic expectation on developers to identify and mitigate such issues in advance.
- Fault susceptibility: ML inference and TEEs are vulnerable to faults from hardware issues or attacks [18, 38, 70, 76, 86, 93, 108]. These faults can lead to undesirable consequences affecting all subsequent inputs. A recent attack highlighted the vulnerability of model inference to runtime fault injections through a single bit-flip at the code or library level [70]. Considering that the attack targets only one of many vulnerable ML inference dependencies, it is likely that more similar attacks will surface in the future.

We stress that the causes and targets of these issues vary widely, and some are inherently complex to resolve. Therefore, our core idea is not to prevent them to happen but to ensure they are detected and resolved before leading to damage. In this paper, we present MVTEE, a novel TEE-based model inference system using the Multi-Variant Execution (MVX) technique (Figure 1) – a method that enhances software security and reliability by running multiple, functionally equivalent but diversified program **variants** in parallel and **monitoring** execution divergence at designated **checkpoints**. The fundamental concept behind MVX is that a particular defect or exploit can impact only one specific variant, while the other variants will either crash or yield inconsistent execution results.

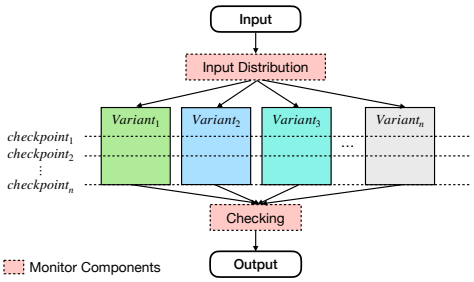



**Figure 1: High-Level Architecture of MVX**

- To insert checkpoints that enable effective early detection and response, MVTEE applies random-balanced model partitioning, which also provides pipelined execution opportunities.
- To generate variants with minimal manual effort, MVTEE leverages the heterogeneous, interoperable, and modular nature of the ML inference stack to introduce multi-level diversification.
- To monitor checkpoints under TEE's threat model, we adopt a cross-process MVX monitoring architecture for better isolation and scalability. It supports attestable runtime variant initialization and updates based on a two-stage variant bootstrap design.
- For efficient execution, we further apply selective MVX and asynchronous cross-validation strategies.

We demonstrate the practicality of MVTEE through performance evaluations on a set of models. In a practical setup, we secure model inference with acceptable overhead in sequential execution. In pipelined execution, we achieve comparable performance when the majority of the model is protected with MVX, and we see improved performance when only a specific part of the model requires hardening. We also explore the fundamental overheads of MVTEE and analyze the efficiency of selective MVX. Our security analysis shows that MVTEE can effectively protect against targeted attacks.

Overall, this paper makes the following contributions:

- We introduce a TEE-based MVX system for secure ML inference, addressing its inherent security and reliability challenges. Both MVX for ML inference and TEE-based MVX are less explored in the previous research.
- We present an automatic approach to construct MVX checkpoints and inference variants through random-balanced model partitioning and ML-native multi-level diversification.
- We propose a generic and security-first design for a TEE-based MVX system that includes efficiency optimizations and supports attestable runtime variant initialization and updates.
- We demonstrate MVTEE's efficiency through evaluations on various real-world models and provide a security analysis of its attack surfaces, explaining the applied mitigations.

# 2 Background

## 2.1 Secure ML Inference Service

ML inference has massive adoption in applications [48, 49]. In particular, Deep Neural Networks (DNNs) which consist of multiple layers organized in a Directed Acyclic Graph (DAG), have emerged as the predominant ML technique due to their efficacy [62]. To enable easy, scalable, fault-tolerant, and cost-efficient model inference, cloud inference services have gained traction. This paradigm is supported by major cloud service providers [1, 9, 104].

To address security concerns with cloud-based inference, two primary methods have been explored. The first involves cryptographic techniques that come at a high performance cost [34, 50, 95, 131]. The alternative is to build inference on TEEs, which is more prevalent in large-scale deployments [53, 58, 63, 65, 66, 74, 78, 94, 103].

## 2.2 Trusted Execution Environment

TEEs isolate code and data in secure memory segments that are inaccessible to non-TEE executions, including privileged administrators and system software. Nowadays, the majority of hardware vendors support CPU TEEs in two types: process-based (e.g., Intel SGX [27, 82]) and virtual machine (VM)-based (e.g., Intel TDX [6], AMD SEV [100], ARM CCA [71]). While GPU TEEs are recently becoming available [10], they are less accessible, with their attack surfaces underexplored. TEEs also provide an attestation mechanism that allows a verifier to authenticate the code and data within a local or remote TEE and securely transmit secrets to it [83, 106].

However, software vulnerabilities in workloads fall outside of the TEE threat model [17, 25, 64, 114], and attestation only provides load-time integrity [85, 109]. In practice, workloads inevitably contain vulnerabilities, and resolving all of them beforehand is impractical. TEEs are also susceptible to fault attacks, such as those exploiting dynamic voltage scaling interfaces [52, 86] or Rowhammer attacks on TEE memory [46]. Notably, such fault attacks can be executed remotely, posing a real threat to secure cloud services. Besides, researchers have identified that TEEs are vulnerable to side-channel and transient execution vulnerabilities [89, 113, 126]. Attackers can also exploit multiple vulnerabilities to mount attacks.

## 2.3 Challenges to TEE-based Secure Inference

Recent advances in ML-targeted attacks complicate achieving the intended security and reliability objectives of TEE-based secure inference. Firstly, a large number of vulnerabilities have been identified in ML frameworks and libraries, especially those written in memory-unsafe languages [24, 29, 42, 44, 56, 101, 139]. Data from the CVE website shows that since 2019, TensorFlow has had over 140 overflow and memory corruption vulnerabilities, which can result in code execution or Denial of Service (DoS) [5]. Another growing concern is faults targeting the weight parameters of DNN models [18, 38, 76] or runtime inference code/libraries [70]. These attacks identify and alter vulnerable bits to compromise model integrity and reduce prediction accuracy. Most TEE-based secure ML inference solutions overlook these threats in their threat models. This oversight can undermine protection efforts, causing severe consequences in security- and safety-critical applications.

## 2.4 Multi-Variant Execution

The N-Version system, or Multi-Variant Execution (MVX), hardens programs through replication and diversification, based on that any defect or attack would impact only a subset of variants [16, 28, 32, 33, 40, 77, 97, 119, 120]. Specifically, a monitor distributes inputs to multiple diversified variants running concurrently and periodically checks their execution results. If discrepancies occur, MVX uses a voting mechanism to decide which output to accept, whether to halt, or to restart from a saved state. MVX is primarily used for attack detection, but also for seamless software updates [39, 80, 91]

However, MVX is not without challenges. First, running multiple variants simultaneously consumes additional CPU and memory resources [40]. Replicating entire programs also complicates and slows down synchronization, especially when with extensive checkpoints. Another major obstacle in practice is that creating and maintaining diverse yet consistent software variants is difficult and expensive [61]. Manual diversification is burdensome and may require the involvement of several independent teams, increasing cost and time. Other known challenges include false alarms [97, 98], and support for multithreading [118] and shared memory [117].

# 3 Overview

## 3.1 System Model

**Threat model.** TEEs consider the following as untrusted: (i) hardware outside the CPU package, (ii) privileged system software, (iii) co-located applications, including unrelated TEEs (except architectural TEEs), and (iv) user-space components in the untrusted world. Denial-of-service (DoS) attacks are considered out of scope. MVTEE adopts this standard TEE threat model as its basis.

MVTEE aims to protect the same assets as previous secure inference designs. We consider the models (including topology and weights) and all inference data (input, output, and intermediate data) as sensitive. We highlight runtime security of ML inference, focusing on (i) software memory-safety and runtime errors, and (ii) faults in models or the ML framework/libraries, which can lead to data leaks, integrity breaches, or incorrect inference outcomes.

Side channel attacks are orthogonal and are thus out of scope. We do not consider algorithm-level model stealing, inversion, backdoor, and membership inference attacks where dedicated defenses are available. TEEs can help mitigate some of these attacks like pre-load model/code poisoning through attestation and isolation.

We assume MVTEE components are reliable and flawlessly implemented. The tools used for variant generation, which may contain vulnerabilities or produce false positives/negatives, are considered out of scope. Same as other MVX systems, MVTEE's precise security guarantees depend on the exact transformations (which address specific classes of attacks) applied to each variant. Note that such transformations would not increase the runtime Trusted Computing Base (TCB) of the system.

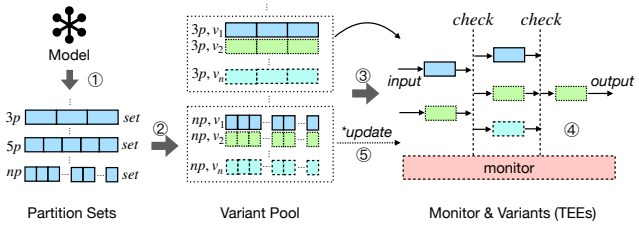



**Figure 2: Usage and Deployment Model**

**Usage and deployment model.** As shown in Figure 2, in the offline phase, we systematically partition the to-be-protected model into partition sets ① and generate diversified variants for each partition ②. This creates a pool of inference variants in an automatic manner. During the online phase, we setup the MVTEE monitor TEE and multiple variant TEEs. Specifically, the variants are dynamically initialized from the pre-established variant pool ③, based on a (selective) MVX plan (§4.3) maintained by the monitor. Throughout execution, the monitor synchronizes and verifies variants' outputs at checkpoints and distributes them to the next stage variants for stateless, non-interfering inferences in a pipelined fashion ④. The monitor can then use a voting process to decide on necessary protective measures in case of detected inconsistencies. A full or partial secure update of variants can be arranged by MVTEE ⑤.

## 3.2 Design Goals

MVTEE achieves the following design goals:

**Security- and safety-first principle.** MVTEE aims to diversify model inference over space and time. Specifically, we apply random-balanced partitioning to generate checkpoints for effective early threat detection and response. We build a pool of diversified variants for dynamic variant initialization and updates. We also adopt an MVX architecture that prioritizes security and fault isolation with minimal TCB and attack surfaces.

**ML native.** MVTEE leverages the heterogeneous, interoperable, and composable nature of ML inference stack and ecosystem to produce multi-level diversification of variants. This also promotes the applicability and reduces the cost of manual diversification.

**Adherence to TEE threat model.** MVTEE must not weaken the security assumptions in the original TEE threat model. We base on a cross-process monitoring architecture that anchors the root of trust in the monitor TEE rather than relying on the trustworthiness of the host kernel. The monitor manages the attestation, key distribution, secure binding, fault tolerance and execution state monitoring of variant TEEs. We design a two-stage variant bootstrap to maintain the secrecy of the MVX details from privileged attackers.

**Efficient execution.** MVTEE aims to deliver reliable but efficient inference. We introduce selective MVX that replicates a flexible number of variants only on the selected, sensitive partitions. We further support execution in an asynchronous cross-validation mode to accommodate potentially significant execution time differences between inference variants in practice.

# 4 Design

## 4.1 Model Partition

To effectively generate checkpoints that enable early detection, termination, and recovery from defects or attacks, we use model partitioning. This method divides the computational graph of the model into smaller subgraphs, with the connections between these subgraphs naturally forming the checkpoints. We highlight that the number and size of the produced partitions are crucial for maintaining both security/safety and performance benefits. Specifically, the capability of early reaction is directly proportional to the number of partitions/checkpoints, whereas the efficiency of sequential partition execution is inversely proportional. In terms of performance, overly small partitions might lose optimization opportunities that are provided by a ML compiler or inference runtime, as they would now need to be applied within individual partitions rather than across multiple ones. From security and safety perspectives, small partitions can introduce risks due to their simplified structure, potentially offering attackers unintended advantages. It could also miss failed exploit attempts or delay safety guarantees, as some fault-caused discrepancies may be hidden by the model's resilience

[38] or not instantly visible [73]. In contrast, large partitions would compromise our objective of prompt response to issues.

Therefore, to provide sufficient diversity and runtime flexibility, we use a randomized graph partitioning based on the global min-cuts algorithm [51] for checkpoint generation, as presented in Algorithm 1. Specifically, we formalize soft preferences (based on a customized and extensible weight function) and hard constraints to "bias" towards more balanced partition sizes by default. Optionally, the algorithm can be run multiple times to identify correct and globally optimal configurations that meet specific requirements (e.g., balance, security levels). We repeat this model partitioning with different target numbers, creating a diverse range of partition sets and checkpoint configurations.

**Algorithm 1:** Random Contraction for Model Partitioning

**Input** : Model Graph $G$, target number of partitions $t$
**Output**: Array of partitions, each containing a list of nodes

**Procedure** `ModelRandomContraction(`$G, t$`)`:
  $par, parSize \leftarrow \{n : n \text{ for } n \text{ in } G\}, \{n : 1 \text{ for } n \text{ in } G\}$;
  $edges \leftarrow \{(i, j) \text{ for } i, j \text{ in } G \text{ if } i \text{ outputs to } j\}$;
  ComputeWeights($edges$, $par$, $parSize$);
  **while** *number of partitions* $> t$ **do**
    $(i, j) \leftarrow$ RandEdgeByWeight($edges$, $par$, $parSize$);
    **if** *CheckConstraints*($par[i], par[j]$); **then**
      MergePartitions($i$, $j$, $par$, $parSize$);
      UpdateWeights($edges$, $par$, $parSize$);
    **end**
  **end**
  **return** *Partitions formed by nodes sharing the same par*;

We emphasize that this partition-as-checkpoint design can bring parallelization opportunities through pipelined execution, nicely hiding the overhead associated with MVX monitoring and synchronization, as they enable compute-communication overlapping.

## 4.2 Variant Generation

The ML inference ecosystem offers a variety of frameworks, tools, runtimes, acceleration libraries, and hardware backends with great interoperability and composability. This high natural diversity provides opportunities for efficient defect and intrusion detection with minimal manual effort, which often involves maintenance burdens and compatibility challenges. Inspired by this observation, MVTEE proposes an automatic, ML-native approach to generate variants with multi-level diversification, as demonstrated in Figure 3.

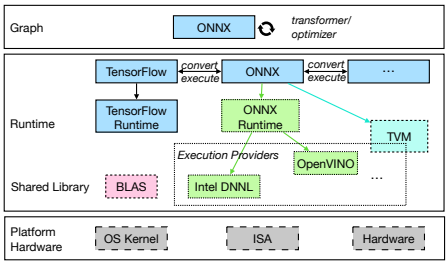



**Figure 3: Multi-Level Diversification of Variants**

**Model graph level.** A DNN model is represented as a DAG containing primitive operators connected to form functions in different topologies. Graph-level diversification adds complexity and unpredictability, making it harder for attackers to identify or exploit vulnerabilities. This diversification extends to all supported lower-level backends. Specifically, MVTEE uses Open Neural Network Exchange (ONNX) [11] for graph-level diversification. ONNX is an open-source format that defines an extensible computational graph model, operators, and standard data types. It serves as a common intermediate representation (IR) for various ML frameworks. We apply ONNX-to-ONNX transformations or optimizations to construct functionally equivalent graph-level model variants by:

- Dummy operators: adding operators that won't change the original outputs such as zero or identity operators.
- Equivalent operator replacement: e.g., transforming a single operator into equivalent smaller ones (decomposition) or the vice versa (fusion), substituting a convolutional operator with an equivalent fully connected linear operator.
- Channel manipulation: duplicating or shuffling the output channels of a convolutional or linear layer and adjusting the subsequent layer's weights accordingly.
- Selective optimization: instead of comprehensive optimization, selectively fusing or eliminating operators as a defense.
- Other mathematical-property-based graph rewriting [87]: e.g., reordering operators that are commutative in nature.

**Inference instance level.** MVTEE produces diversification at the inference instance level (including the inference runtimes, acceleration libraries and supported hardware) via three paths.

First, we leverage the interoperability offered by cross-framework converters to transform a model in ONNX format to various framework formats (e.g., PyTorch, TensorFlow, Mindspore, Paddle). This enables MVTEE to execute under different ML frameworks using distinct and configurable runtime settings.

Second, we use ONNX Runtime (ORT) [12], the default inference engine of ONNX, leveraging its Execution Providers (EP) framework to create variants that work with different executors, acceleration libraries and hardware. ORT supports a wide array of EPs today.

Third, we capitalize on ML compilers such as TVM [22]. The ML compiler processes high-level model specifications (e.g., in ONNX format) and performs code generation, translation, and optimization across various abstraction levels (e.g., graph-level and operator-level) for different hardware backends. At the operator level, the ML compiler often uses auto-tuning techniques to iteratively identify the most efficient implementation options [14, 23, 116, 141]. This generates trial candidates that vary in kernel parameters, tensor operation strategies like fusion and parallelization, as well as tensor data layouts and memory management, thereby naturally achieving diversification. Model compilation may also involve third-party toolchains like LLVM to apply further low-level IR transformations (including instrumentation). In addition, TVM includes several built-in executors and is itself supported by ORT as an EP.

Note that some inference runtimes include built-in support for graph-level transformations. In such cases, MVTEE can either utilize this feature to create configurable or non-deterministic diversification, or explicitly disallow them to retain the deterministic diversification established at the previous model graph level.

**Other levels of variant generation.** Since ML dependencies are compiled with conventional compilers, existing compiler-assisted security mechanisms can be applied in conjunction. These include different types of sanitizers, stack protection, compiler-inserted padding, runtime checks, and hardware-assisted enforcement. Moreover, system-level security mechanisms like Address Space Layout Randomization (ASLR) are supported in MVTEE. We also support execution in SGX and TDX, providing TEE-level variants, and offer Application Binary Interface (ABI) or Instruction Set Architecture (ISA) level diversification through distributed execution on different backends. While we focus on software-level rather than algorithm-level variant generation, these methods can be applied independently. Inheriting from MVX, MVTEE is not limited by specific security mechanisms and allows the concurrent use of multiple defenses by assigning different measures to each variant.

## 4.3 TEE-based MVX

MVTEE is composed of a monitor TEE and multiple variant TEEs at runtime. The monitor acts as the security manager, ensuring the correct initialization and execution states of all variants. Based on a runtime-provisioned MVX configuration that specifies the partition set (number and sizes of partitions) and the variant claims (type and number of variants per partition), the monitor manages the attestation, key distribution, binding and fault tolerance of variants.

We base our system on an enclave abstraction (i.e., a one-to-one mapping between TEE, process, and variant) to provide high level of isolation with a minimal trust assumption. We highlight that this abstraction is not specific to process-based TEEs but is also applicable to VM-based TEEs [60]. In MVTEE, we currently support SGX and TDX, with the potential to extend to other CPU TEEs.

**Monitoring architecture.** There are different types of MVX monitoring designs as illustrated in Figure 4, each offering distinct security-performance trade-offs. In MVTEE, we adopt a Cross-Process User-Space (CP/US) monitor design where the MVX monitor operates within a separate TEE backed by a TEE OS with minimal attack surfaces and a very low TCB.

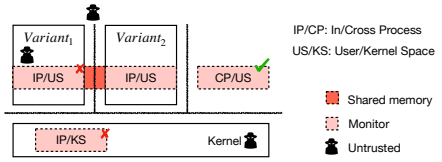



**Figure 4: Monitor Architecture Design Choices**

This design is driven by two main considerations: (i) under TEE's threat model, the privileged kernel is outside the TCB, rendering Kernel-Space (KS) monitoring inapplicable; (ii) targeting potential vulnerabilities within the in-process software stack, a cross-process monitor provides better security and fault isolation, ensuring that a compromised variant cannot compromise the monitor. This architecture also naturally supports execution in a distributed setting. We rely on cryptographically protected secure channels established after successful attestation for monitor-variant interactions.

**Two-stage variant bootstrap.** To align with TEE's threat model, where MVX details must remain confidential from untrusted parties (e.g., orchestrators responsible for resource management in standard cloud deployments), we introduce a two-stage variant bootstrap design. Initially, each variant TEE is assigned an *init-variant*, similar to the concept of Kubernetes init containers [7]. As illustrated in Figure 5, this design segregates files and settings into public and private parts. The public part comprises the *init-variant* and its settings, while the private part hosts the main variant-specific files and settings in encrypted form. This setup ensures that untrusted orchestrators are only involved in the initial placement of variant TEE containers, each loaded with an *init-variant* and public settings. Following successful attestation, each *init-variant* receives a variant-specific key and a variant identifier, allowing for the decryption and setup of main variant-specific files and settings.

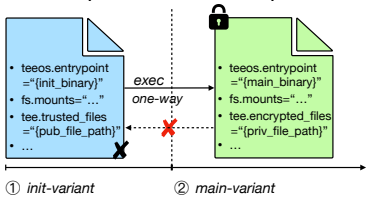



**Figure 5: Two-Stage Variant Bootstrap Example**

We observe that the two stages necessitate distinct environments and resources for execution. In particular, their security settings such as allowed files, environment variables, `IOCTLs` and system calls (syscalls) can also differ. These settings that regulate application execution are typically specified in a specialized manifest file provided by the TEE OS. This file is loaded, parsed and enforced at its boot time. Therefore, we require the support of two-stage manifests, where a different second-stage manifest can be installed post launch. We allow access to this installation interface via TEE OS's customized interfaces. To prevent abuse, we further require one-time installation: once setup, it is locked, unmodifiable, and not accessible by the main variant executing in the second stage.

The stage transition is designed to be one-way, triggered by the first `exec()` syscall from the *init-variant*. The newly installed manifest is then enforced and supersedes any prior settings following this transition. This phased lifecycle design not only simplifies the *init-variant*'s functionality, reducing its TCB and attack surfaces, but also minimizes the attack surface for each specific variant by limiting its environment and resources strictly to necessities.

**Attestable variant initialization and updates.** Variants are dynamically initialized following the protocol depicted in Figure 6. Initially, the orchestrator schedules the monitor TEE and a set of variant TEEs (starting with *init-variant*) per the model owner's request, where they await initialization commands ①. The model owner first verifies the authenticity and integrity of the monitor TEE through a challenge-response attestation based on a hardware-signed TEE report, and establishes a secure connection with it ②. An MVX configuration is then provisioned to the monitor ③. To protect against potential replay attacks, a nonce is used. Based on the MVX specifications, a selection of partition variants is made (either deterministically or randomly) from the pre-established pool and maintained within the monitor ④. Following this, the monitor and variants setup secure channels through RA-TLS [54] implemented at the socket level. It then assigns a variant-specific key and its corresponding variant identifier to each variant ⑤. The *init-variant* installs this key into the TEE OS for future decryption, fetches the encrypted variant-specific manifest and files, and establishes secure connections with adjacent partitions if required. The TEE OS configures and locks the manifest, attesting its successful installation

by sending evidence back to the monitor ⑥. Upon receipt of this confirmation, the monitor verifies and binds each connection with the respective variant and meta data ⑦. Finally, the initialization results, along with the nonce, are sent back to the model owner for verification ⑧. During inference runtime, users perform a combined attestation of all TEEs through the monitor, then provision their secret inputs for subsequent execution, which is carried out through the partitioned variants in a pipelined manner.

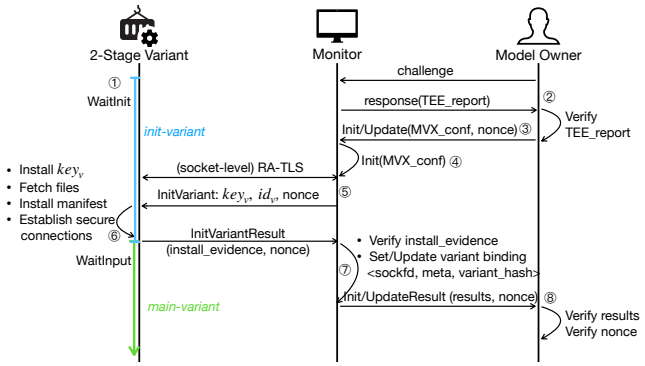



**Figure 6: Workflow of Variant Initialization/Updates**

We support full and partial variant updates, following similar workflows to initialization. Full updates reshuffle partition sets and reconstruct bindings, while partial updates replace or scale certain variants, updating bindings in an appending-only way for auditing purposes. While reusing TEEs could reduce overhead, we opt against it in updates due to (i) potential security risks from incomplete and unsound software-level cleanups, which can open up new attack surfaces and contradict our security-first principle, and (ii) updates may include changes to model partitions or runtimes, making the associated loading costs unavoidable.

**Selective MVX.** In practice, not all parts within a trained model requires protection. For instance, many modern ML models utilize transfer learning [111], which is cost-effective and relies on smaller datasets. Typically, these models begin with a publicly available trained model from repositories like Huggingface and are fine-tuned by retraining certain final layers or by partially altering the model. Consequently, only a subset of layers – those that contain sensitive information – need to be protected. Similarly, different operators, parameters, or instructions show varied susceptibility to faults [38, 55, 70, 73, 75]. Based on this observation, we have designed selective MVX to focus on hardening the partitions most susceptible to threats while offering flexibility that can reduce the resource waste and overhead associated with static full replication. Specifically, we support vertical scaling by allowing certain partitions to activate MVX, and horizontal scaling by controlling the number of variants per MVX-enabled partition. These can be configured to adapt to dynamic online environments, to meet varying security, Quality of Service (QoS), or resource demands.

**Execution model.** Variant TEEs are organized by the monitor into a DAG that mirrors the original model topology, processing private user data in a pipelined manner. Specifically, sequential execution occurs when data is processed linearly, with each batch completing all partition stages before the next batch begins. Conversely, pipelined execution involves processing batch streams simultaneously, with each pipeline stage handling a different batch. We intentionally avoid installing all or multiple partitions within the same variant TEE for enhanced security and isolation, preventing checkpoint bypass if a variant has already been compromised. The monitor initially dispatches identical user inputs to variants in the first partition for inference and gathers their intermediate results.

For efficient execution, we propose a slow-fast path design as illustrated in Figure 7. With a slow path, the monitor suspends at designated checkpoints to evaluate the variant outputs against predefined rules and employs a voting strategy to decide on further actions. We use criteria-based consistency checks with thresholds and different metrics (§5.2) to differentiate attacks from benign divergences. This allows for evaluation based on variant-specific characteristics that may result from diversification or inference variability. Specifically, we apply relevant consistency metrics (e.g., similarity- or error-based measures) to variant pairs and adjust thresholds based on variant noise levels to balance the precision and recall of attack identification in the case of divergences. After a successful evaluation, it will select and replicate the intermediate results, then forward them to the subsequent partition variants. The fast path, on the other hand, allows outputs to directly fall through to the next partition variants. By default, MVTEE operates in a hybrid mode. Under selective MVX, it automatically activates the slow path when multiple variants are applied per partition and switches to the fast path when a single variant is used. Furthermore, all inter-TEE data communications are encrypted and authenticated with unique sequence numbers for freshness, and are preferably oblivious to avoid timing side channels.

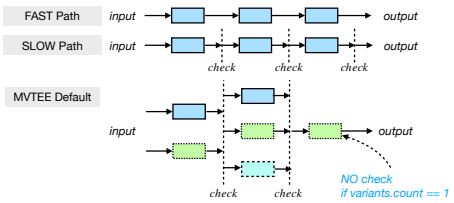



**Figure 7: Slow-Fast Path Design**

In line with our CP/US architecture, MVTEE implements cross-process voting. Different voting mechanisms imply varying levels of agreement, impacting detection reliability, with panel sizes also involving reliability/resource trade-offs. We default to a unanimous consent strategy to prioritize security and reliability, but MVTEE is flexible in terms of voting algorithms. Under this strategy, we allow both synchronous and asynchronous execution modes:

- Sync mode: The monitor pauses at specific checkpoints to evaluate all variant outputs against predefined rules. Any instance of dissent prompts, the monitor performs an immediate response to adjust the execution.
- Async Mode: Given the execution time can vary significantly among variants in practice, we design a cross-validation strategy to mitigate the impact of delays in variant execution, as demonstrated in Figure 8. The pipeline is allowed to proceed once a majority consensus is reached at the checkpoints. When results from delayed variants are received, and if any dissent is noted, we react to the execution at the earliest next checkpoint. This mode is inherently inapplicable for full MVX without partitioning.

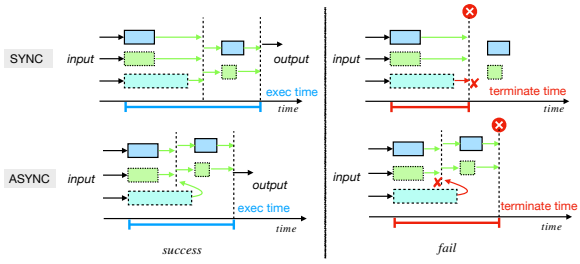



**Figure 8: Execution in Sync/Async Mode**

## 5 Implementation

Our prototype consists of 3.6K Lines of Code (LoC) of Python and 1.7K LoC of C. It is composed of two main parts: the offline ML MVX tooling and the runtime TEE MVX system. The runtime of MVTEE is extended based on Gramine-SGX/TDX (v1.7) [60, 112].

### 5.1 Offline ML MVX Tool

We develop an offline ML MVX tool to streamline and automate the process of model partitioning and the generation of variants for each partition, as presented in Figure 2. This tool offers several modules: model inspection, model partitioning, and the construction of partition variants. The inputs required for this tool are: (i) the target model file for secure inference, (ii) configuration files that detail model partitioning settings and variant specifications, and (iii) base Gramine manifest files for restricting the environment and resources. The final outputs include partition variants with their respective Gramine manifests in encrypted form. Besides, we generate monitor and base variant container images that package the Gramine TEE OS, TEE-related files, along with the corresponding public executables and manifests. For real-world deployments, declarative configurations can be integrated into the Security Development Lifecycle for diversification customization, and auto variant correctness verification can be added into CI/CD.

**Model inspection and partition modules.** These modules are built on ONNX. Through model inspection, we collect information such as IR version, graph inputs/outputs, initializers, and nodes, including their indices and detailed metadata. MVTEE's partition module currently offers two modes: manual and automatic. In manual mode, the module functions as a graph slicer, generating graph slices based on the specified node names or indices. This mode is practical for model owners with expert knowledge on how to create effective checkpoints (e.g., they have prior knowledge on which operators are more sensitive). Automatic partitioning implements the random contraction algorithm. It offers a customizable weight function to define formalized soft preferences and a constraint function to specify hard requirements. By default, the module is configured to prioritize balanced partitions in terms of size to trade-off between security and performance. With additional information, such as the security/safety sensitivity of nodes, the module can be extended to prioritize other objectives. Specifically, users set: (i) the target number of partition sets, (ii) the number of partitions/checkpoints per set, and (iii) customized preferences and constraints settings to obtain randomized model partitions. To speed up the process, our tool also supports parallel graph partitioning.

**Variant construction module.** This module first supports a set of ONNX to ONNX transformations/optimizations elaborated in §4.2 to achieve model graph-level diversification on the partitioned subgraphs. For diversification at the inference instance level, we generate variant-specific Gramine manifests and bootstrap scripts based on variant configurations in JSON format. This configuration specifies the runtime, dependencies and runtime-specific diversification strategies (e.g., applied compiler passes, third-party toolchain flags, and the type of TVM executor in the case of a TVM runtime). Other diversification strategies such as N-versioning, conventional compiler-assisted diversification, or a combination with system-level diversification settings can be applied independently. With Gramine, MVTEE supports various inference runtimes on x86-64 CPU TEEs, including ORT, TVM, OpenVINO, Pytorch, TensorFlow, and TensorFlow Lite. As the final step, we use the `gramine-sgx-pf-crypt` utility to encrypt the manifest, necessary files, and dependencies using a variant-specific key.

### 5.2 Runtime TEE MVX System

**Enhancements to Gramine.** We first extend Gramine to support the two-stage manifests, which can be enabled via a newly added manifest option. Specifically, we allow the one-time installation of a second-stage manifest via pseudo file system interfaces. Not all features in the standard Gramine manifest are supported for the second-stage; MVTEE focuses on the security-related ones, such as trusted/encrypted files settings and syscall restrictions. The new manifest takes effect on the subsequent `exec()` syscall. It mandates execution solely from Gramine's encrypted files, as per the design of *init-variant*, and prohibits any key manipulation in the second stage, ensuring all necessary configurations are installed by the *init-variant* prior to main variant execution. Upon `exec()`, we reset the status as thoroughly as possible before switching to the second-stage manifest to ensure that the two stages are completely independent. This includes zeroing out all applicable virtual memory areas, closing unrelated file descriptors, resetting the program break, clearing thread-local storage, removing custom signal handlers, unlinking ELF objects, and unloading any dynamically loaded libraries or ELF objects in use of the *init-variant* stage.

For enhanced security, we enforce Gramine's code as RX-only pages, using the new page permission management instructions provided by the Enclave Dynamic Memory Management feature of SGX2 [81]. Besides, rather than depending on the RA-TLS library to establish secure connections, we enhance Gramine to support socket-level RA-TLS and data encryption. This lower-level enforcement helps prevent potential application-level bypasses and ensures that all data transmitted over the network is consistently authenticated and encrypted (with AES-GCM-256). Additionally, we extend Gramine to include syscall restrictions, further reducing the runtime attack surface across separate variant execution stages.

**Monitor and *init-variant*.** The MVTEE runtime is implemented following the CP/US design, where the monitor and the variants operate in separate TEEs based on the enhanced Gramine. They can be deployed either in a co-located or distributed setting. For TCB and attack surface minimization, we only implement the necessary functionalities in the monitor and *init-variant*. This also reduces the extra Enclave Page Cache (EPC) consumption, and with the support of dynamic memory management provided in TEE backends, the TEE initialization overhead can be minimized.

We implement the monitor-side of the dynamic variant initialization and update protocol. The monitor maintains MVX settings and manages all variants' attestation, key distribution and binding updates. We also implement the input distribution, checkpoint synchronization and output replication within the monitor following the design detailed in §4.3. To assess the consistency of outputs from different partition variants, we implement configurable checking based on criteria such as cosine similarity, mean squared error, maximum absolute difference, and `np.testing.assert_allclose` (with predefined absolute and relative tolerances). Moreover, we implement the designed efficiency optimizations including selective MVX, slow-fast path and asynchronous cross-validation execution.

The *init-variant* is implemented following the two-stage bootstrap design. The core functionality of *init-variant* comprises the variant-side of the init/update protocol, leveraging the extended Gramine's one-time manifest installation for the second stage.

# 6 Evaluation

In this section, we evaluate the MVTEE system's performance to assess its practicality. In addition, we conduct a security analysis of MVTEE's attack surfaces, discussing the applied mitigations.

## 6.1 Experimental Setup

**Testbed.** Our experiments are conducted on dual-socket platforms with Intel Xeon Gold 6354 CPUs (36 cores per socket), 378GB of RAM, 128GB of SGX EPC and 10 Gbps Ethernet, running Ubuntu 22.04. We use `numactl` with `cpunodebind` and `membind` to bind processes to a single Non-Uniform Memory Access (NUMA) domain for consistent results. We perform warmup runs and repeat trials to minimize measurement noise, with average values reported.

**Models.** We evaluate the pre-trained DNN models: EfficientNet-b7, GoogleNet, Innception V3, MnasNet, MobileNet V3, ResNet-152 and ResNet-50. We use 32-bit floating-point precision with a batch size of 1 and input data size of 3×224×224 by default.

**Partitions.** We generate the partitions using our random-balanced algorithm. They are tested for correctness before evaluation.

**Variants.** To evaluate the fundamental performance of MVTEE and the impact of selective MVX, we apply identical/replicated variants running on ONNX runtime (v1.18.1) to minimize execution time variations among variants. For real setup performance analysis, we build variants run on either ONNX runtime or TVM (v0.15.0) graph executor, implementing multi-level diversification (§4.2).

**Execution.** We measure throughput and end-to-end latency under both sequential and pipelined execution. Unless otherwise noted, all inter-variant and variant-monitor data is encrypted (with AES-GCM-256) and transferred via TCP/IP sockets. We apply MVTEE's hybrid execution mode by default (Figure 7). Asynchronous execution is activated only in real setup evaluations, addressing execution time differences among different variants.

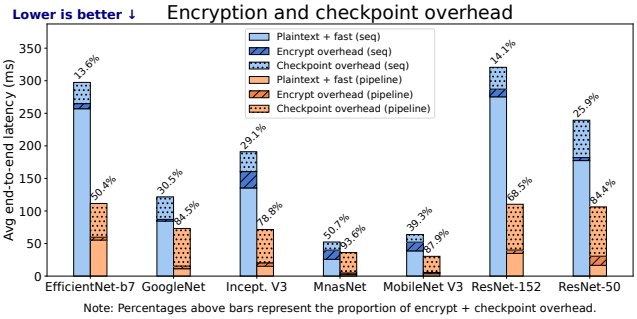
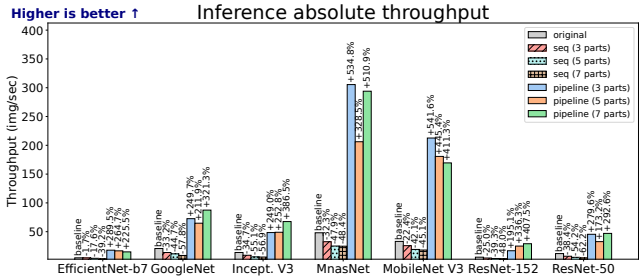



**Figure 10: Encryption and Checkpoint Overheads**

## 6.2 Fundamental Performance of MVTEE

We first evaluate the performance impact of our random-balanced partition strategy with different numbers of partitions. Results depicted in Figure 9 show performance across different models on a full fast path. We establish the performance of the original model as our baseline. In sequential execution, the throughput of inference decreases by 1.7% to 62.2% compared to the baseline, worsening as the number of partitions increases. Correspondingly, latency rises by 1.7% to 164.3%. However, in pipelined execution, MVTEE achieves throughputs 1.7x to 5.4x higher than the baseline and reduces latency by 63.4% to 84.4%. Additional partitions could possibly bring further performance benefits, since they form a longer processing pipeline and increase parallelism. This indicates that while checkpoint insertion via partitioning introduces overheads, the resulting pipeline opportunities can successfully hide them through compute-communication overlapping.

Next, we evaluate the encryption and checkpointing overheads stemming from MVTEE's cross-process monitoring architecture and its specific threat model. Our experiments use a 5-partition setup with no encryption and a full fast path as the baseline. The checkpointing overhead is measured on a full slow path. Figure 10 shows that encryption and checkpointing contributed to an overall overhead of 13.6% to 50.7% in sequential execution and an even higher proportion in pipelined execution – ranging from 50.4% to 93.6%. These overheads are more impactful on smaller models such as MobileNet and MnasNet. In particular, checkpointing is a major source of overhead due to additional variant-monitor data



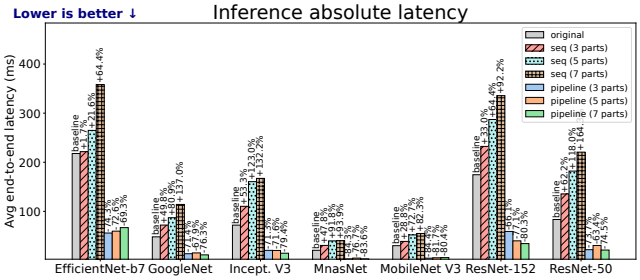




**Figure 9: Performance Impact of Random-Balanced Partitioning**

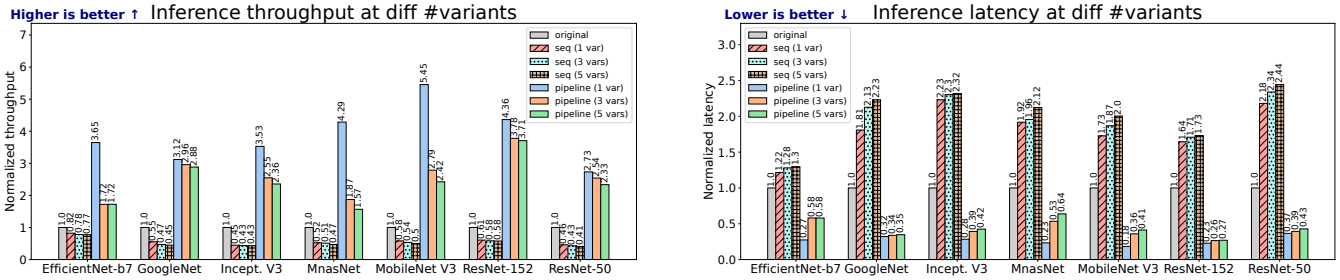



**Figure 11: Normalized Performance of Horizontal Variant Scaling Using Selective MVX**

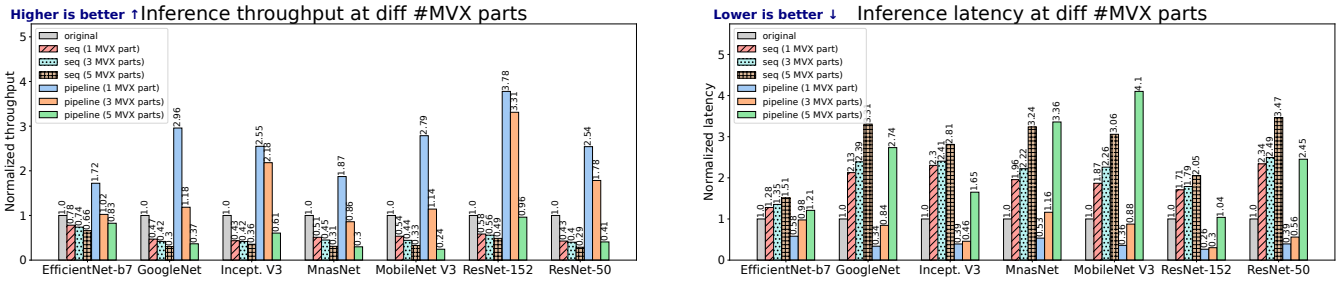



**Figure 12: Normalized Performance of Vertical Variant Scaling Using Selective MVX**

transmissions and cryptographic operations, while the verification computation typically completes quickly enough. However, we highlight that through MVTEE's fast path design, the overall overhead can be mitigated by up to 28.3% in sequential execution and up to 86.5% in pipelined execution. Additionally, while encryption overhead is inevitable, it can be optimized through more efficient cryptographic algorithms and implementations, or by limiting the size of data transferred between partitions.

## 6.3 Performance of Selective MVX

We analyze the performance of selective MVX by examining vertical and horizontal scaling of MVX configurations. We keep the original model performance as our baseline.

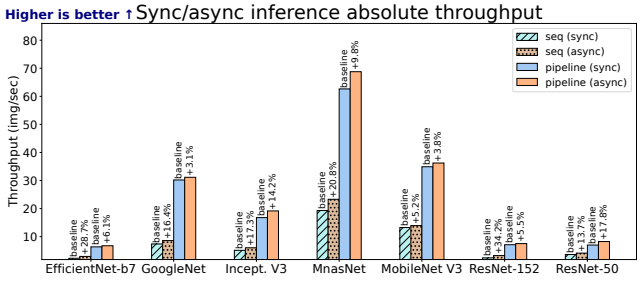
We evaluate horizontal scaling in a 5-partition setup, specifically scaling the 3$^{rd}$ partition with varying numbers (1, 3, and 5) of variants. Figure 11 illustrates the normalized performance outcomes. In sequential execution, the overhead from horizontal variant scaling is negligible compared to the partitioning-caused overhead (the difference between *seq* (1 *var*) and *original*). In pipelined execution, the initial MVX activation (from 1 to 3 variants) imposes noticeable overheads across models. Interestingly, adding more variants (from 3 to 5) has a lesser impact. We attribute this to the transition from the fast path to the slow path in the MVX-enabled partition, which requires additional synchronization and consistency checks. It can further create bottlenecks in the MVX-enabled pipeline stage, causing subsequent stages to wait and slow down the overall pipeline. Fine-grained pipeline optimizations could potentially alleviate this issue. Note that all pipelined executions significantly outperform the original models, achieving at least 1.6x throughput and less than 0.7x latency across all scaling settings.

Figure 12 presents the normalized performance statistics for vertical variant scaling, where we test varying numbers of partitions to enable MVX, each running 3 variants. Specifically, we enable MVX on the 3$^{rd}$ partition for 1-MVX configuration and on the 3$^{rd}$, 4$^{th}$, and 5$^{th}$ partitions for a 3-MVX configuration. In sequential execution, throughput is maintained at a minimum of 0.4x and latency below 2.5x for both 1- and 3-MVX-enabled partitions. However, expanding to a full 5-MVX configuration results in a significant performance reduction, with throughput dropping to about 0.3x and latency exceeding 3x for most models. In pipelined execution, the performance decline is mitigated by concurrent processing benefits, in particular in the case of 1- and 3-MVX-enabled partitions. In these configurations, we highlight that the pipelined execution generally outperforms the original models. Specifically, with 1-MVX-enabled partition, we achieve throughput ranged from 1.7x to 3.8x and average latency from 0.3x to 0.6x; with 3-MVX-enabled partitions, throughput is 0.9x to 3.3x and latency from 0.3x to 1.2x. However, retaining the original performance under full MVX pipelined execution is challenging, with throughput varying between 0.2x and 1.0x and latency ranging from 1.0x to 4.1x. This is attributed to early synchronization in the full MVX setup, which stalls pipelines, delays data progression to subsequent stages, and reduces overall parallelism. We consider this as a lesser concern in general as this setup is specifically reserved for scenarios where exhaustive hardening is prioritized over performance considerations.

## 6.4 Performance of MVTEE in Real Setup

We test variants running on ORT or TVM graph executor, with diversification at different levels to reflect real-world deployments.

First, we assess our asynchronous cross-validation execution mode in a 5-partition MVTEE setup, activating MVX on the 2$^{nd}$ and 3$^{rd}$ partitions, each with 3 variants. In this test, we specifically apply a TVM variant with complex diversification for targeted security checks, which leads to lagging performance. Figure 13 highlights that, compared to synchronous execution, our asynchronous approach achieves a throughput increase of 5.2% to 34.2% in sequential execution and 3.1% to 17.8% in pipelined execution. It also results in an average latency reduction of 5% to 25.6% in sequential execution and 3.1% to 15.2% in pipelined execution.



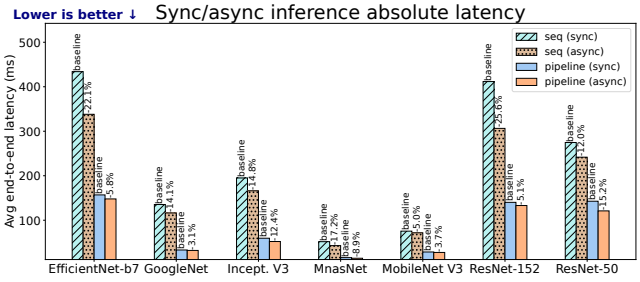
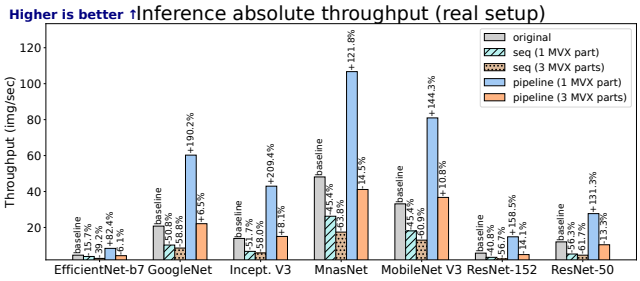




**Figure 13: Performance of Asynchronous Cross-validation Execution Mode**



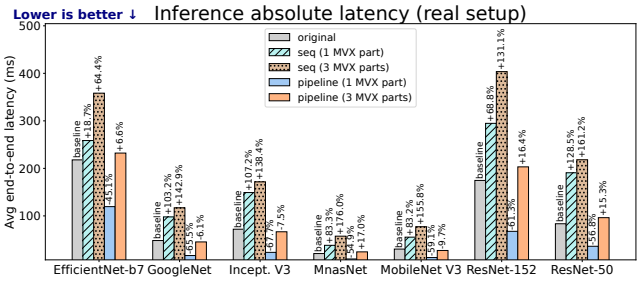




**Figure 14: MVTEE Performance in Real-World Setup**

Further, we evaluate the real-world performance of MVTEE deployment against the original inference baseline. We enable 3-variant MVX execution on one partition (the 3rd partition) and across three partitions (the 3rd, 4th, and 5th partitions), with by default asynchronous execution. Figure 14 illustrates that in sequential execution, throughput is sustained at 0.4x to 0.8x for 1 MVX partition and 0.4x to 0.6x for 3 MVX partitions. Latency overhead is recorded at 18.7% to 128.5% for 1 MVX partition and 64.4% to 176% for 3 MVX partitions. Given MVTEE's focus on security and reliability, this performance level is considered acceptable. In pipelined execution, performance improvements are observed: a throughput increase of 82.4% to 209.4% and a latency reduction of 45.1% to 67.7% when 1 MVX partition is selectively activated; and an 85.5% to 110.8% throughput with latency between -9.7% to 17% is maintained when 3 MVX partitions are enabled, covering the majority of the model. Since (i) not all parts of a model necessitate hardening and (ii) mainstream managed cloud inference platforms like Amazon SageMaker and Azure AI services provide built-in support for streaming inference targeting real-time scenarios and continuous large-volume data analysis, we highlight the practicality of MVTEE's pipelined execution. We can achieve superior performance by hardening only the most vulnerable subset of a model and maintain comparable performance when the majority of partitions are hardened under MVX.

## 6.5 Security Analysis

MVTEE comprises the following core components at runtime: monitor, *init-variant*, and variant, each linked to a manifest and all running on the TEE OS. We first describe the security properties that these components must uphold: (i) the monitor, *init-variant*, and TEE OS code must be integrity-protected, (ii) the code and data (including manifests) of the variant and the internal state of TEE OS must be confidentiality- and integrity-protected, (iii) all network I/O must be confidentiality- and integrity-protected, (iv) file I/O must be integrity-protected for the monitor and *init-variant*, and confidentiality-protected for the variant, (v) the TEE OS must not be vulnerable to privileged attacks, (vi) the monitor and *init-variant* must be hardened against software and fault vulnerabilities, (vii) tampering with all manifest files must be detectable, (viii) the chain of trust of TEE attestation must reflect all loaded components and remain immutable. In the following, we systematically study the possible attack vectors and explain their mitigations.

**Software vulnerabilities in variants.** Attackers with access to public APIs can send maliciously-crafted inputs to exploit memory-safety or runtime errors in ML frameworks and libraries. Our empirical analysis of various TensorFlow CVEs shows that MVTEE can mitigate such attacks through the variants exemplified in Table 1. Variants generated from inference-instance-level transformations and traditional MVX diversification are the most effective against these attacks. Certain model-graph-level transformations, e.g., equivalent operator replacement, can also help by nullifying vulnerabilities specific to certain operator implementations, but we don't highlight this as models are mathematical representations and not directly tied to software vulnerabilities. Intuitively, variants seem unnecessary since existing defenses or their combinations can also mitigate these vulnerabilities. However, a single defense is often inadequate due to attack diversity, while trivial combined protection on one target is prohibitively expensive and often conflicting [133]. MVTEE is orthogonal and can be applied on top to effectively detect broad classes of attacks.

**Faults in variants.** Attackers can induce runtime faults to ML inference locally or remotely (e.g., through a co-located container which has access to vulnerable interfaces). For model-targeted attacks (e.g., on operators, weights), graph-level transformations can alter graph-level properties and thus change fault susceptibility, making targeted faults unreliable or invalid. Existing algorithm-level countermeasures can also be applied. To counter faults injected at the ML framework/library level, inference-instance-level transformations can be used. For instance, the FrameFlip attack targets fault-vulnerable bits in the OpenBLAS [13] linear algebra backend,

**Table 1: TensorFlow Vulnerabilities and Defending Variants. (OOB: Out of Bound Read/Write, UNP: Uninitialized/Null Pointers, FPE: Floating Point Exception, IO: Integer Overflow, UAF: Use After Free, ACF: Assertion Check Failures, RT: Runtime)**

| Type | Example CVE | Impact | Variants e.g. |
|---|---|---|---|
| OOB | CVE-2021-41226<br>CVE-2022-41883<br>CVE-2022-41900<br>CVE-2023-25668 | DoS<br>Data corruption<br>R/W primitives<br>Code execution | Different RT<br>Bounds check<br>Sanitizers<br>ASLR |
| UNP | CVE-2022-21739<br>CVE-2023-25672 | DoS<br>Incorrect results | Different RT<br>Sanitizers |
| FPE | CVE-2022-21725 | DoS<br>Incorrect results | Different RT<br>Error handling<br>Compiler |
| IO | CVE-2022-21727<br>CVE-2022-21733 | DoS<br>Data corruption<br>Incorrect results | Different RT<br>Sanitizers<br>Compiler |
| UAF | CVE-2021-37652 | DoS<br>Data corruption<br>Code execution | Different RT<br>Sanitizers |
| ACF | CVE-2022-35935 | DoS | Different RT<br>Error handling |

but is ineffective against a variant using a different BLAS implementation (e.g., Eigen [2] or Intel MKL [124]). Certain fault attacks are only possible on specific TEE hardware [137]. MVTEE supports TEE-level variants to help mitigate these attacks.

**Additional variant hardening.** We harden the variants' TEE runtime/OS against privileged attacks, such as malicious exceptions and signals [105], by cross-verifying host-reported signals against TEE-reported exceptions. The TEE OS maintains the state of the application's requests and proactively cross-checks, e.g., the opened files and the statue of network connections. We further shield pipes, network and file system I/O by implementing automatic encryption and decryption at the TEE OS level to ensure all data leaving the TEE boundary remains confidential. This guarantees that only the monitor and the specific variant itself, which hold the decryption key, can access the data. The variant manifest by default blocks all command-line arguments and environment variables from the untrusted host and only enables through a controlled allow-list when necessary. It is impossible for attackers to abuse the variant manifest installation feature at the application level, as we enforce a one-time and one-way manifest installation by *init-variant*, and the interface is disabled during the variant execution stage.

**Attacks on *init-variant* and initialization/updates.** MVTEE monitor must perform attestation to establish trust in *init-variant* before distributing any variant-specific keys and updating secure bindings. TEE reports that include measurements of the entire software stack of *init-variant* are sent to the monitor. Any discrepancies, such as a malformed manifest or tempered code, can be detected due to unexpected measurements. To defend replay attacks from untrusted environments, we use a nonce check to ensure freshness.

Attackers may attempt to inspect the manifest and files of a variant, but this would simply fail as they are encrypted with a variant-specific key. Key rotation can be conducted on a regular basis for proactive defense. Note that the variant-specific key acts as a key derivation key for the TEE OS's encrypted filesystem, while actual file encryption uses one-time keys. Therefore, it results in much less ciphertext compared to use it for direct data encryption. This prolongs the time to reach NIST recommended key usage thresholds and lessens the burden of key rotation.

The hashes of trusted files used by *init-variant* are generated and stored in its manifest at build time. TEE OS verifies all opened files in this set against their reference hashes at runtime. MVTEE's encrypted files can suffer from rollback/replay attacks [79], where an attacker reverts files to an older state. We partially mitigate this by maintaining freshness metadata at runtime but a complete defense requires independent monotonic counters. Fork attacks [15] occur when an attacker creates diverging instances of the same variant. MVTEE mitigates this by disallowing reuse or migration, with its monitor ensuring secure variant binding before execution.

**Monitor security.** The monitor acts as the trust anchor and is designed with minimal attack surfaces and a very low TCB. The monitor is also hardened against any untrusted inputs. It interacts with model owners and variants, ensuring that variant initialization and updates adhere to the protocol. Further, the distributed nature of cross-process monitoring allows for independent hardening of the minimalistic monitor. Instead of running in a large-memory TEE (e.g., Intel SGX2 or TDX) that sacrifices hardware-level memory integrity checks, the monitor can be hosted in a small integrity-enhanced TEE (e.g., Intel SGX1, using Message Authentication Code and an integrity tree) to mitigate RAM corruption and replacement without incurring additional secure memory swapping overheads.

# 7 Related Work and Discussions

## 7.1 TEE-based Secure Model Inference

TEEs have been applied in secure model inference through two primary methods. The first is comprehensive TEE-shielding, where the entire model is secured within TEEs [58, 63, 78]. While this maintains confidentiality and accuracy same as the original model, it comes with performance penalties mainly due to EPC constraints. To tackle this, previous studies have proposed using lightweight models [94], reusing a shared memory pool for model weights and pipelining loading for layered processing [53, 65, 74], as well as through model partitioning [65, 66]. MVTEE also leverages model partitioning, but with a distinct focus on security and reliability through MVX. It requires randomized partitioning and prioritizes balanced checkpoint insertion for effective early threat detection and response. Besides, MVTEE aims to support newer TEEs like Intel SGX2, Intel TDX, and AMD SEV with large secure memory capacities, in which case limited secure memory is less of a concern.

A second method is TEE-shielded DNN partition, which places only a portion of the DNN model within TEEs while offloading the rest to non-TEEs (e.g., GPUs) for computational efficiency [30, 41, 84, 102, 107]. Existing studies broadly assume that the offloaded segments do not reveal critical private information of models. Yet, this assumption has been challenged under practical threat models. Though a recent approach suggests partition-before-training to deliver security equivalent to full TEE-shielding [140], it does not apply to post-training models – the focus of MVTEE. MVTEE strives to offer baseline protection equivalent to full TEE-shielding, even with selective MVX. MVTEE further considers stronger threat model including software vulnerabilities and faults.

## 7.2 Defenses Against Relevant Attacks

Recent research has explored vulnerabilities in ML frameworks and libraries, examining their security impacts, exploitability, root causes, and suggesting remediation solutions [29, 42, 44, 56, 101, 139]. To counter memory corruptions within TEEs, one approach is using memory-safe programming languages like Rust [4, 125] or running in restricted WebAssembly sandboxes [3]. Additionally, hardening techniques including ASLR [99], bounds checking [59], fuzzing [21, 26] and symbolic execution [129] have been proposed. MVTEE is orthogonal and can provide comprehensive protection based on them. Moreover, research proposes Control-Flow Attestation [85] and provenance analysis [109] to detect runtime exploits, addressing limitations in the built-in TEE attestation. Different from these solutions, MVTEE not only detects anomalous executions but also responds through termination, recovery, or updates.

To protect DNN models against faults, researchers have proposed using hardware-level hardening [67] and model-level fault-tolerance [37, 69] or fault detection [47, 68, 72, 73]. However, these methods often suffer from limited detection capabilities – model-level hardening fails to address faults induced at lower levels, e.g., within ML frameworks or libraries [70]. Some proposals suggest relying on TEEs to protect DNNs from fault injections [70, 132], but these can be broken by new types of attacks [86, 93, 108]. General defenses against TEE-targeted faults include hardware-level mitigations [138], restricting access to vulnerable interfaces [86], and software-based hardening [55]. MVTEE offers a generic system-level defense which can be applied to existing TEEs with minimal effort. With its distributed architecture, MVTEE can fully leverage any available defenses by e.g., running variants on hardware implemented with specific hardening or constructing variants with software- or algorithm-level protection. Research shows that neurons and weights in a DNN exhibit varying fault sensitivities [38, 73, 75], and more generally, instructions can have different susceptibilities to faults and impacts on inference tasks [55, 70]. The effectiveness of MVTEE's selective MVX is based on this rationale.

## 7.3 MVX Design and Applications

A range of MVX system designs have been explored, with distinct choices on monitoring architectures and checkpoint strategies [16, 28, 33, 40, 77, 97, 119, 120]. Researchers have also proposed MVX systems using hardware features [57, 128], or running in a distributed setting [96, 121]. MVTEE is specifically designed for TEEs. To our knowledge, no TEE-based MVX design has been proposed. We adopt cross-process monitoring and checkpointing on outputs to align with our design goals. In particular, model inference is typically multithreaded where non-deterministic scheduling can complicate syscall-level synchronization and cross-checks [118]. MVTEE uses output-level checkpointing, enabling multithreaded execution without implementing complex synchronization, which also contributes to minimized TCB and reduced overhead. Other monitoring solutions (e.g., in-process [134] or hybrid [119]) and checkpointing options (e.g., on syscalls [45, 92, 135]) can be adapted to MVTEE, but this involves non-trivial effort to conform to the TEE threat model and its peculiarities.

Research on variant generation implements diversification at different stages of the software lifecycle, including development, compilation, linking, installation, loading, and execution [19, 61, 110, 130]. Some researchers also suggest using heterogeneous ABIs or ISAs [123, 127]. MVX has been proposed to harden user-space applications [31], OS kernels [90], cloud microservices [32], and in edge scenarios [20]. MVTEE aims to apply MVX in DNN inference, an area less covered by prior research. It is designed with ML-native characteristics in mind, particularly in its variant generation.

Resource overhead is a known MVX trade-off, providing more comprehensive protection at the cost of increased resource usage. Compared to the same level of single-target combined protection, MVX trades resource for execution time. Different MVX settings result in varying security/performance/resource trade-offs. In MVTEE, we target the protection of critical inference services that prioritize security and safety. Through selective MVX, we allow replicating only the subset of models more susceptible to threats, reducing the overheads of full static replication. Integrating MVTEE into inherent cloud redundancy patterns like Kubernetes *ReplicaSets* [8] or leveraging underutilized hardware on demand can further mitigate costs. Selective and asynchronous execution are common optimization mechanisms, also implemented in some other MVX systems [122, 136]. MVTEE incorporates unique adaptations, such as runtime variant initialization and updates, along with asynchronous cross-validation designs, aligned with its specific objectives.

## 7.4 Future Work

MVTEE currently targets secure inference using CPU TEEs, which are more accessible, widely deployed, and well-studied targets of advanced attacks. Adapting MVTEE to domain-specific (xPU) TEEs requires further evaluation and non-trivial porting efforts, which we plan to work on. While this work focuses on DNNs, running large Foundation Models within CPU TEEs is also practical, offering flexibility and cost-effectiveness. We may explore MVTEE's applicability to these models and address their specific challenges. Additionally, investigating the trade-offs between security, performance, and resource utilization introduced by different MVX strategies is an interesting topic for future research.

# 8 Conclusion

Under a practical threat model, TEE-shielded model inference remains susceptible to a variety of software vulnerabilities and faults, which could break its security and reliability promises. We present MVTEE, a TEE-based model inference system applying MVX for elevated confidentiality and integrity guarantees. MVTEE leverages model partitioning and the heterogeneous nature of the inference stack to generate MVX checkpoints and variants. Based on cross-process monitoring and two-stage variant bootstrapping, MVTEE can securely run multiple, diversified inference variants concurrently, enabling timely threat detection and response. With efficiency optimizations including asynchronous selective execution, we show that MVTEE provides a practical deployment model well-suited for security- and safety-critical model inference.

# Acknowledgments

We thank our shepherd Dr. Davide Frey and the anonymous reviewers for their helpful comments. This work was supported by National Science and Technology Major Project (No. 2022ZD0120304).

## References


[1] 2024. Amazon Elastic Inference. https://docs.aws.amazon.com/elastic-inference.

[2] 2024. Eigen: a C++ template library for linear algebra: matrices, vectors, numerical solvers, and related algorithms. https://eigen.tuxfamily.org.

[3] 2024. Enarx: Confidential Computing with WebAssembly. https://github.com/enarx/enarx.

[4] 2024. Fortanix EDP: Enclave Development Platform. https://edp.fortanix.com/.

[5] 2024. Google Tensorflow: Product details, threats and statistics. https://www.cvedetails.com/product/53738/Google-Tensorflow.html.

[6] 2024. Intel® Trust Domain Extensions (Intel® TDX). https://www.intel.com/content/www/us/en/developer/articles/technical/intel-trust-domainextensions.html.

[7] 2024. Kubernetes Init Containers. https://kubernetes.io/docs/concepts/workloads/pods/init-containers.

[8] 2024. Kubernetes ReplicaSet. https://kubernetes.io/docs/concepts/workloads/controllers/replicaset.

[9] 2024. Model inference overview. https://cloud.google.com/bigquery/docs/inference-overview.

[10] 2024. NVIDIA Confidential Computing. https://www.nvidia.com/en-us/data-center/solutions/confidential-computing.

[11] 2024. ONNX: Open standard for machine learning interoperability. https://github.com/onnx/onnx.

[12] 2024. ONNX Runtime: cross-platform, high performance ML inferencing and training accelerator. https://github.com/microsoft/onnxruntime.

[13] 2024. OpenBLAS: an optimized BLAS library based on GotoBLAS2 1.13 BSD version. https://github.com/OpenMathLib/OpenBLAS.

[14] Andrew Adams, Karima Ma, Luke Anderson, Riyadh Baghdadi, Tzu-Mao Li, Michaël Gharbi, Benoit Steiner, Steven Johnson, Kayvon Fatahalian, Frédo Durand, et al. 2019. Learning to optimize halide with tree search and random programs. *ACM Transactions on Graphics (TOG)* 38, 4 (2019), 1–12.

[15] Fritz Alder, Arseny Kurnikov, Andrew Paverd, and N Asokan. 2018. Migrating SGX enclaves with persistent state. In *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 195–206.

[16] Emery D Berger and Benjamin G Zorn. 2006. DieHard: Probabilistic memory safety for unsafe languages. *Acm sigplan notices* 41, 6 (2006), 158–168.

[17] Andrea Biondo, Mauro Conti, Lucas Davi, Tommaso Frassetto, and Ahmad-Reza Sadeghi. 2018. The Guard's Dilemma: Efficient {Code-Reuse} Attacks Against Intel {SGX}. In *27th USENIX Security Symposium (USENIX Security 18)*. 1213–1227.

[18] Jakub Breier, Xiaolu Hou, Dirmanto Jap, Lei Ma, Shivam Bhasin, and Yang Liu. 2018. Practical fault attack on deep neural networks. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 2204–2206.

[19] Javier Cabrera Arteaga. 2022. *Artificial Software Diversification for WebAssembly*. Ph. D. Dissertation. KTH Royal Institute of Technology.

[20] Javier Cabrera Arteaga, Pierre Laperdrix, Martin Monperrus, and Benoit Baudry. 2022. Multi-variant Execution at the Edge. In *Proceedings of the 9th ACM Workshop on Moving Target Defense*. 11–22.

[21] Liheng Chen, Zheming Li, Zheyu Ma, Yuan Li, Baojian Chen, and Chao Zhang. 2024. EnclaveFuzz: Finding Vulnerabilities in SGX Applications. In *NDSS*.

[22] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. 2018. {TVM}: An automated {End-to-End} optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 578–594.

[23] Tianqi Chen, Lianmin Zheng, Eddie Yan, Ziheng Jiang, Thierry Moreau, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. Learning to optimize tensor programs. *Advances in Neural Information Processing Systems* 31 (2018).

[24] Neophytos Christou, Di Jin, Vaggelis Atlidakis, Baishakhi Ray, and Vasileios P Kemerlis. 2023. {IvySyn}: Automated Vulnerability Discovery in Deep Learning Frameworks. In *32nd USENIX Security Symposium (USENIX Security 23)*. 2383–2400.

[25] Tobias Cloosters, Michael Rodler, and Lucas Davi. 2020. {TeeRex}: Discovery and exploitation of memory corruption vulnerabilities in {SGX} enclaves. In *29th USENIX Security Symposium (USENIX Security 20)*. 841–858.

[26] Tobias Cloosters, Johannes Willbold, Thorsten Holz, and Lucas Davi. 2022. {SGXFuzz}: Efficiently synthesizing nested structures for {SGX} enclave fuzzing. In *31st USENIX Security Symposium (USENIX Security 22)*. 3147–3164.

[27] Victor Costan and Srinivas Devadas. 2016. Intel SGX explained. *Cryptology ePrint Archive* (2016).

[28] Benjamin Cox, David Evans, Adrian Filipi, Jonathan Rowanhill, Wei Hu, Jack Davidson, John Knight, Anh Nguyen-Tuong, and Jason Hiser. 2006. N-Variant Systems: A Secretless Framework for Security through Diversity.. In *USENIX Security Symposium*, Vol. 114. 114.

[29] Zizhuang Deng, Guozhu Meng, Kai Chen, Tong Liu, Lu Xiang, and Chunyang Chen. 2023. Differential Testing of Cross Deep Learning Framework {APIs}: Revealing Inconsistencies and Vulnerabilities. In *32nd USENIX Security Symposium (USENIX Security 23)*. 7393–7410.

[30] Tarek Elgamal and Klara Nahrstedt. 2020. Serdab: An IoT framework for partitioning neural networks computation across multiple enclaves. In *2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*. IEEE, 519–528.

[31] Shuhei Enomoto and Hiroshi Yamada. 2022. A Multi-variant Execution Environment for Securing In-memory KVSes. In *2022 18th European Dependable Computing Conference (EDCC)*. IEEE, 9–16.

[32] Antonio M Espinoza, Riley Wood, Stephanie Forrest, and Mohit Tiwari. 2022. Back to the future: N-Versioning of Microservices. In *2022 52nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 415–427.

[33] Robert Gawlik, Philipp Koppe, Benjamin Kollenda, Andre Pawlowski, Behrad Garmany, and Thorsten Holz. 2016. Detile: Fine-grained information leak detection in script engines. In *Detection of Intrusions and Malware, and Vulnerability Assessment: 13th International Conference, DIMVA 2016, San Sebastián, Spain, July 7-8, 2016, Proceedings 13*. Springer, 322–342.

[34] Ran Gilad-Bachrach, Nathan Dowlin, Kim Laine, Kristin Lauter, Michael Naehrig, and John Wernsing. 2016. Cryptonets: Applying neural networks to encrypted data with high throughput and accuracy. In *International conference on machine learning*. PMLR, 201–210.

[35] Udit Gupta, Carole-Jean Wu, Xiaodong Wang, Maxim Naumov, Brandon Reagen, David Brooks, Bradford Cottel, Kim Hazelwood, Mark Hempstead, Bill Jia, et al. 2020. The architectural implications of facebook's dnn-based personalized recommendation. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 488–501.

[36] Kim Hazelwood, Sarah Bird, David Brooks, Soumith Chintala, Utku Diril, Dmytro Dzhulgakov, Mohamed Fawzy, Bill Jia, Yangqing Jia, Aditya Kalro, et al. 2018. Applied machine learning at facebook: A datacenter infrastructure perspective. In *2018 IEEE international symposium on high performance computer architecture (HPCA)*. IEEE, 620–629.

[37] Zhezhi He, Adnan Siraj Rakin, Jingtao Li, Chaitali Chakrabarti, and Deliang Fan. 2020. Defending and harnessing the bit-flip based adversarial weight attack. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 14095–14103.

[38] Sanghyun Hong, Pietro Frigo, Yiğitcan Kaya, Cristiano Giuffrida, and Tudor Dumitraş. 2019. Terminal brain damage: Exposing the graceless degradation in deep neural networks under hardware fault attacks. In *28th USENIX Security Symposium (USENIX Security 19)*. 497–514.

[39] Petr Hosek and Cristian Cadar. 2013. Safe software updates via multi-version execution. In *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 612–621.

[40] Petr Hosek and Cristian Cadar. 2015. Varan the unbelievable: An efficient n-version execution framework. *ACM SIGARCH Computer Architecture News* 43, 1 (2015), 339–353.

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

[42] Nargiz Humbatova, Gunel Jahangirova, Gabriele Bavota, Vincenzo Riccio, Andrea Stocco, and Paolo Tonella. 2020. Taxonomy of real faults in deep learning systems. In *Proceedings of the ACM/IEEE 42nd international conference on software engineering*. 1110–1121.

[43] Gadi Hutt, Vibhav Viswanathan, and Adam Nadolski. 2019. Deliver high performance ML inference with AWS Inferentia. (2019). https://d1.awsstatic.com/events/reinvent/2019/REPEAT_1_Deliver_high_performance_ML_inference_with_AWS_Inferentia_CMP324-R1.pdf

[44] Md Johirul Islam, Giang Nguyen, Rangeet Pan, and Hridesh Rajan. 2019. A comprehensive study on deep learning bug characteristics. In *Proceedings of the 2019 27th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering*. 510–520.

[45] Adriaan Jacobs, Merve Gülmez, Alicia Andries, Stijn Volckaert, and Alexios Voulimeneas. 2024. System Call Interposition Without Compromise. In *2024 54th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 183–194.

[46] Yeongjin Jang, Jaehyuk Lee, Sangho Lee, and Taesoo Kim. 2017. SGX-Bomb: Locking down the processor via Rowhammer attack. In *Proceedings of the 2nd Workshop on System Software for Trusted Execution*. 1–6.

[47] Mojan Javaheripi and Farinaz Koushanfar. 2021. Hashtag: Hash signatures for online detection of fault-injection attacks on deep neural networks. In *2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*. IEEE, 1–9.

[48] Junchen Jiang, Ganesh Ananthanarayanan, Peter Bodik, Siddhartha Sen, and Ion Stoica. 2018. Chameleon: scalable adaptation of video analytics. In *Proceedings of the 2018 conference of the ACM special interest group on data communication*. 253–266.

[49] Eric Jonas, Qifan Pu, Shivaram Venkataraman, Ion Stoica, and Benjamin Recht. 2017. Occupy the cloud: Distributed computing for the 99%. In *Proceedings of the 2017 symposium on cloud computing*. 445–451.

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

[51] David R Karger. 1993. Global Min-cuts in RNC, and Other Ramifications of a Simple Min-Cut Algorithm.. In *Soda*, Vol. 93. Citeseer, 21–30.

[52] Zijo Kenjar, Tommaso Frassetto, David Gens, Michael Franz, and Ahmad-Reza Sadeghi. 2020. {V0LTpwn}: Attacking x86 processor integrity from software. In *29th USENIX Security Symposium (USENIX Security 20)*. 1445–1461.

[53] Kyungtae Kim, Chung Hwan Kim, Junghwan" John" Rhee, Xiao Yu, Haifeng Chen, Dave Tian, and Byoungyoung Lee. 2020. Vessels: Efficient and scalable deep learning prediction on trusted processors. In *Proceedings of the 11th ACM Symposium on Cloud Computing*. 462–476.

[54] Thomas Knauth, Michael Steiner, Somnath Chakrabarti, Li Lei, Cedric Xing, and Mona Vij. 2018. Integrating remote attestation with transport layer security. *arXiv preprint arXiv:1801.05863* (2018).

[55] Andreas Kogler, Daniel Gruss, and Michael Schwarz. 2022. Minefield: A Software-only Protection for {SGX} Enclaves against {DVFS} Attacks. In *31st USENIX Security Symposium (USENIX Security 22)*. 4147–4164.

[56] Denis Kolegov and Anton Nikolaev. [n. d.].

[57] Koen Koning, Herbert Bos, and Cristiano Giuffrida. 2016. Secure and efficient multi-variant execution using hardware-assisted process virtualization. In *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 431–442.

[58] Roland Kunkel, Do Le Quoc, Franz Gregor, Sergei Arnautov, Pramod Bhatotia, and Christof Fetzer. 2019. Tensorscone: A secure tensorflow framework using intel sgx. *arXiv preprint arXiv:1902.04413* (2019).

[59] Dmitrii Kuvaiskii, Oleksii Oleksenko, Sergei Arnautov, Bohdan Trach, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. 2017. SGXBOUNDS: Memory safety for shielded execution. In *Proceedings of the Twelfth European Conference on Computer Systems*. 205–221.

[60] Dmitrii Kuvaiskii, Dimitrios Stavrakakis, Kailun Qin, Cedric Xing, Pramod Bhatotia, and Mona Vij. 2024. Gramine-TDX: A Lightweight OS Kernel for Confidential VMs. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*. 4598–4612.

[61] Per Larsen, Andrei Homescu, Stefan Brunthaler, and Michael Franz. 2014. SoK: Automated software diversity. In *2014 IEEE Symposium on Security and Privacy*. IEEE, 276–291.

[62] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. 2015. Deep learning. *nature* 521, 7553 (2015), 436–444.

[63] Dayeol Lee, Dmitrii Kuvaiskii, Anjo Vahldiek-Oberwagner, and Mona Vij. 2020. Privacy-Preserving Machine Learning in Untrusted Clouds Made Simple. *arXiv preprint arXiv:2009.04390* (2020).

[64] Jaehyuk Lee, Jinsoo Jang, Yeongjin Jang, Nohyun Kwak, Yeseul Choi, Changho Choi, Taesoo Kim, Marcus Peinado, and Brent ByungHoon Kang. 2017. Hacking in darkness: Return-oriented programming against secure enclaves. In *26th USENIX Security Symposium (USENIX Security 17)*. 523–539.

[65] Taegyeong Lee, Zhiqi Lin, Saumay Pushp, Caihua Li, Yunxin Liu, Youngki Lee, Fengyuan Xu, Chenren Xu, Lintao Zhang, and Junehwa Song. 2019. Occlumency: Privacy-preserving remote deep-learning inference using SGX. In *The 25th Annual International Conference on Mobile Computing and Networking*. 1–17.

[66] Fabing Li, Xiang Li, and Mingyu Gao. 2023. Secure MLaaS with Temper: Trusted and Efficient Model Partitioning and Enclave Reuse. In *Proceedings of the 39th Annual Computer Security Applications Conference*. 621–635.

[67] Guanpeng Li, Siva Kumar Sastry Hari, Michael Sullivan, Timothy Tsai, Karthik Pattabiraman, Joel Emer, and Stephen W Keckler. 2017. Understanding error propagation in deep learning neural network (DNN) accelerators and applications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–12.

[68] Jingtao Li, Adnan Siraj Rakin, Zhezhi He, Deliang Fan, and Chaitali Chakrabarti. 2021. Radar: Run-time adversarial weight attack detection and accuracy recovery. In *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 790–795.

[69] Jingtao Li, Adnan Siraj Rakin, Yan Xiong, Liangliang Chang, Zhezhi He, Deliang Fan, and Chaitali Chakrabarti. 2020. Defending bit-flip attack through dnn weight reconstruction. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 1–6.

[70] Shaofeng Li, Xinyu Wang, Minhui Xue, Haojin Zhu, Zhi Zhang, Yansong Gao, Wen Wu, and Xuemin Sherman Shen. 2024. Yes, One-Bit-Flip Matters! Universal DNN Model Inference Depletion with Runtime Code Fault Injection. In *Proceedings of the 33th USENIX Security Symposium*.

[71] Xupeng Li, Xuheng Li, Christoffer Dall, Ronghui Gu, Jason Nieh, Yousuf Sait, and Gareth Stockwell. 2022. Design and verification of the arm confidential compute architecture. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. 465–484.

[72] Yu Li, Min Li, Bo Luo, Ye Tian, and Qiang Xu. 2020. Deepdyve: Dynamic verification for deep neural networks. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. 101–112.

[73] Yu Li, Yannan Liu, Min Li, Ye Tian, Bo Luo, and Qiang Xu. 2019. D2nn: a fine-grained dual modular redundancy framework for deep neural networks. In *Proceedings of the 35th Annual Computer Security Applications Conference*. 138–147.

[74] Yuepeng Li, Deze Zeng, Lin Gu, Quan Chen, Song Guo, Albert Zomaya, and Minyi Guo. 2021. Lasagna: Accelerating secure deep learning inference in sgx-enabled edge cloud. In *Proceedings of the ACM Symposium on Cloud Computing*. 533–545.

[75] Qi Liu, Wujie Wen, and Yanzhi Wang. 2020. Concurrent weight encoding-based detection for bit-flip attack on neural network accelerators. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD), 2020*.

[76] Yannan Liu, Lingxiao Wei, Bo Luo, and Qiang Xu. 2017. Fault injection attack on deep neural network. In *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 131–138.

[77] Kangjie Lu, Meng Xu, Chengyu Song, Taesoo Kim, and Wenke Lee. 2018. Stopping memory disclosures via diversification and replicated execution. *IEEE Transactions on Dependable and Secure Computing* 18, 1 (2018), 160–173.

[78] Junming Ma, Chaofan Yu, Aihui Zhou, Bingzhe Wu, Xibin Wu, Xingyu Chen, Xiangqun Chen, Lei Wang, and Donggang Cao. 2020. S3ML: A secure serving system for machine learning inference. *arXiv preprint arXiv:2010.06212* (2020).

[79] Sinisa Matetic, Mansoor Ahmed, Kari Kostiainen, Aritra Dhar, David Sommer, Arthur Gervais, Ari Juels, and Srdjan Capkun. 2017. {ROTE}: Rollback protection for trusted execution. In *26th USENIX Security Symposium (USENIX Security 17)*. 1289–1306.

[80] Matthew Maurer and David Brumley. 2012. TACHYON: Tandem execution for efficient live patch testing. In *21st USENIX Security Symposium (USENIX Security 12)*. 617–630.

[81] Frank McKeen, Ilya Alexandrovich, Ittai Anati, Dror Caspi, Simon Johnson, Rebekah Leslie-Hurd, and Carlos Rozas. 2016. Intel® software guard extensions (intel® sgx) support for dynamic memory management inside an enclave. In *Proceedings of the Hardware and Architectural Support for Security and Privacy 2016*. 1–9.

[82] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R Savagaonkar. 2013. Innovative instructions and software model for isolated execution. *Hasp@ isca* 10, 1 (2013).

[83] Jämes Ménétrey, Christian Göttel, Anum Khurshid, Marcelo Pasin, Pascal Felber, Valerio Schiavoni, and Shahid Raza. 2022. Attestation mechanisms for trusted execution environments demystified. In *IFIP International Conference on Distributed Applications and Interoperable Systems*. Springer, 95–113.

[84] Fan Mo, Ali Shahin Shamsabadi, Kleomenis Katevas, Soteris Demetriou, Ilias Leontiadis, Andrea Cavallaro, and Hamed Haddadi. 2020. Darknetz: towards model privacy at the edge using trusted execution environments. In *Proceedings of the 18th International Conference on Mobile Systems, Applications, and Services*. 161–174.

[85] Mathias Morbitzer, Benedikt Kopf, and Philipp Zieris. 2023. GuaranTEE: Introducing control-flow attestation for trusted execution environments. In *2023 IEEE 16th International Conference on Cloud Computing (CLOUD)*. IEEE, 547–553.

[86] Kit Murdock, David Oswald, Flavio D Garcia, Jo Van Bulck, Daniel Gruss, and Frank Piessens. 2020. Plundervolt: Software-based fault injection attacks against Intel SGX. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1466–1482.

[87] Wei Niu, Jiexiong Guan, Yanzhi Wang, Gagan Agrawal, and Bin Ren. 2021. Dnnfusion: accelerating deep neural networks execution with advanced operator fusion. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 883–898.

[88] Augustus Odena, Catherine Olsson, David Andersen, and Ian Goodfellow. 2019. Tensorfuzz: Debugging neural networks with coverage-guided fuzzing. In *International Conference on Machine Learning*. PMLR, 4901–4911.

[89] Oleksii Oleksenko, Bohdan Trach, Robert Krahn, Mark Silberstein, and Christof Fetzer. 2018. Varys: Protecting {SGX} Enclaves from Practical {Side-Channel} Attacks. In *2018 Usenix Annual Technical Conference (USENIX ATC 18)*. 227–240.

[90] Sebastian Österlund, Koen Koning, Pierre Olivier, Antonio Barbalace, Herbert Bos, and Cristiano Giuffrida. 2019. kMVX: Detecting kernel information leaks with multi-variant execution. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. 559–572.

[91] Luís Pina, Anastasios Andronidis, Michael Hicks, and Cristian Cadar. 2019. Mvedsua: Higher availability dynamic software updates via multi-version execution. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. 573–585.

[92] Kailun Qin and Dawu Gu. 2024. One System Call Hook to Rule All TEE OSes in the Cloud. In *2024 IEEE 17th International Conference on Cloud Computing (CLOUD)*. IEEE, 205–216.

[93] Pengfei Qiu, Dongsheng Wang, Yongqiang Lyu, and Gang Qu. 2019. Voltjockey: Breaching trustzone by software-controlled voltage manipulation over multi-core frequencies. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. 195–209.

[94] Do Le Quoc, Franz Gregor, Sergei Arnautov, Roland Kunkel, Pramod Bhatotia, and Christof Fetzer. 2020. Securetf: A secure tensorflow framework. In *Proceedings of the 21st International Middleware Conference*. 44–59.

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

[96] André Rösti, Stijn Volckaert, Michael Franz, and Alexios Voulimeneas. 2024. I'll Be There for You! Perpetual Availability in the A 8 MVX System. In *2024 Annual Computer Security Applications Conference (ACSAC)*. IEEE, 520–533.

[97] Babak Salamat, Todd Jackson, Andreas Gal, and Michael Franz. 2009. Orchestra: intrusion detection using parallel execution and monitoring of program variants in user-space. In *Proceedings of the 4th ACM European conference on Computer systems*. 33–46.

[98] Babak Salamat, Todd Jackson, Gregor Wagner, Christian Wimmer, and Michael Franz. 2011. Runtime defense against code injection attacks using replicated execution. *IEEE Transactions on Dependable and Secure Computing* 8, 4 (2011), 588–601.

[99] Jaebaek Seo, Byoungyoung Lee, Seong Min Kim, Ming-Wei Shih, Insik Shin, Dongsu Han, and Taesoo Kim. 2017. SGX-shield: Enabling address space layout randomization for SGX programs.. In *NDSS*.

[100] AMD Sev-Snp. 2020. Strengthening VM isolation with integrity protection and more. *White Paper, January* 53 (2020), 1450–1465.

[101] Qingchao Shen, Haoyang Ma, Junjie Chen, Yongqiang Tian, Shing-Chi Cheung, and Xiang Chen. 2021. A comprehensive study of deep learning compiler bugs. In *Proceedings of the 29th ACM Joint meeting on european software engineering conference and symposium on the foundations of software engineering*. 968–980.

[102] Tianxiang Shen, Ji Qi, Jianyu Jiang, Xian Wang, Siyuan Wen, Xusheng Chen, Shixiong Zhao, Sen Wang, Li Chen, Xiapu Luo, et al. 2022. {SOTER}: Guarding Black-box Inference for General Neural Networks at the Edge. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. 723–738.

[103] Sandra Siby, Sina Abdollahi, Mohammad Maheri, Marios Kogias, and Hamed Haddadi. 2024. GuaranTEE: Towards Attestable and Private ML with CCA. In *Proceedings of the 4th Workshop on Machine Learning and Systems*. 1–9.

[104] Jonathan Soifer, Jason Li, Mingqin Li, Jeffrey Zhu, Yingnan Li, Yuxiong He, Elton Zheng, Adi Oltean, Maya Mosyak, Chris Barnes, et al. 2019. Deep learning inference service at microsoft. In *2019 USENIX Conference on Operational Machine Learning (OpML 19)*. 15–17.

[105] Supraja Sridhara, Andrin Bertschi, Benedict Schlüter, and Shweta Shinde. 2024. SIGY: Breaking Intel SGX Enclaves with Malicious Exceptions & Signals. *arXiv preprint arXiv:2404.13998* (2024).

[106] Frederic Stumpf, Omid Tafreschi, Patrick Röder, Claudia Eckert, et al. 2006. A robust integrity reporting protocol for remote attestation. In *Proceedings of the Workshop on Advances in Trusted Computing (WATC)*. 65.

[107] Zhichuang Sun, Ruimin Sun, Changming Liu, Amrita Roy Chowdhury, Long Lu, and Somesh Jha. 2023. Shadownet: A secure and efficient on-device model inference system for convolutional neural networks. In *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1596–1612.

[108] Adrian Tang, Simha Sethumadhavan, and Salvatore Stolfo. 2017. {CLKSCREW}: Exposing the perils of {Security-Oblivious} energy management. In *26th USENIX Security Symposium (USENIX Security 17)*. 1057–1074.

[109] Flavio Toffalini, Mathias Payer, Jianying Zhou, and Lorenzo Cavallaro. 2022. Designing a provenance analysis for SGX enclaves. In *Proceedings of the 38th Annual Computer Security Applications Conference*. 102–116.

[110] Dominik Töllner, Christian Dietrich, Illia Ostapyshyn, Florian Rommel, and Daniel Lohmann. 2023. {MELF}: Multivariant Executables for a Heterogeneous World. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*. 257–273.

[111] Lisa Torrey and Jude Shavlik. 2010. Transfer learning. In *Handbook of research on machine learning applications and trends: algorithms, methods, and techniques*. IGI global, 242–264.

[112] Chia-Che Tsai, Donald E Porter, and Mona Vij. 2017. {Graphene-SGX}: A Practical Library {OS} for Unmodified Applications on {SGX}. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. 645–658.

[113] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F Wenisch, Yuval Yarom, and Raoul Strackx. 2018. Foreshadow: Extracting the keys to the intel {SGX} kingdom with transient {Out-of-Order} execution. In *27th USENIX Security Symposium (USENIX Security 18)*. 991–1008.

[114] Jo Van Bulck, David Oswald, Eduard Marin, Abdulla Aldoseri, Flavio D Garcia, and Frank Piessens. 2019. A tale of two worlds: Assessing the vulnerability of enclave shielding runtimes. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. 1741–1758.

[115] Stephan Van Schaik, Alex Seto, Thomas Yurek, Adam Batori, Bader AlBassam, Daniel Genkin, Andrew Miller, Eyal Ronen, Yuval Yarom, and Christina Garman. 2024. Sok: Sgx. fail: How stuff gets exposed. In *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE, 4143–4162.

[116] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, William S Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. 2018. Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions. *arXiv preprint arXiv:1802.04730* (2018).

[117] Jonas Vinck, Bert Abrath, Bart Coppens, Alexios Voulimeneas, Bjorn De Sutter, and Stijn Volckaert. 2022. Sharing is caring: Secure and efficient shared memory support for mvees. In *Proceedings of the Seventeenth European Conference on Computer Systems*. 99–116.

[118] Stijn Volckaert, Bart Coppens, Bjorn De Sutter, Koen De Bosschere, Per Larsen, and Michael Franz. 2017. Taming parallelism in a multi-variant execution environment. In *Proceedings of the Twelfth European Conference on Computer Systems*. 270–285.

[119] Stijn Volckaert, Bart Coppens, Alexios Voulimeneas, Andrei Homescu, Per Larsen, Bjorn De Sutter, and Michael Franz. 2016. Secure and efficient application monitoring and replication. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*. 167–179.

[120] Stijn Volckaert, Bjorn De Sutter, Tim De Baets, and Koen De Bosschere. 2013. GHUMVEE: efficient, effective, and flexible replication. In *Foundations and Practice of Security: 5th International Symposium, FPS 2012, Montreal, QC, Canada, October 25-26, 2012, Revised Selected Papers 5*. Springer, 261–277.

[121] Alexios Voulimeneas, Dokyung Song, Per Larsen, Michael Franz, and Stijn Volckaert. 2021. dmvx: Secure and efficient multi-variant execution in a distributed setting. In *Proceedings of the 14th European Workshop on Systems Security*. 41–47.

[122] Alexios Voulimeneas, Dokyung Song, Fabian Parzefall, Yeoul Na, Per Larsen, Michael Franz, and Stijn Volckaert. 2019. DMON: A distributed heterogeneous n-variant system. *arXiv preprint arXiv:1903.03643* (2019).

[123] Alexios Voulimeneas, Dokyung Song, Fabian Parzefall, Yeoul Na, Per Larsen, Michael Franz, and Stijn Volckaert. 2020. Distributed heterogeneous n-variant execution. In *Detection of Intrusions and Malware, and Vulnerability Assessment: 17th International Conference, DIMVA 2020, Lisbon, Portugal, June 24–26, 2020, Proceedings 17*. Springer, 217–237.

[124] Endong Wang, Qing Zhang, Bo Shen, Guangyong Zhang, Xiaowei Lu, Qing Wu, Yajuan Wang, Endong Wang, Qing Zhang, Bo Shen, et al. 2014. Intel math kernel library. *High-Performance Computing on the Intel® Xeon Phi™: How to Fully Exploit MIC Architectures* (2014), 167–188.

[125] Huibo Wang, Pei Wang, Yu Ding, Mingshen Sun, Yiming Jing, Ran Duan, Long Li, Yulong Zhang, Tao Wei, and Zhiqiang Lin. 2019. Towards memory safe enclave programming with rust-sgx. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. 2333–2350.

[126] Wenhao Wang, Guoxing Chen, Xiaorui Pan, Yinqian Zhang, XiaoFeng Wang, Vincent Bindschaedler, Haixu Tang, and Carl A Gunter. 2017. Leaky cauldron on the dark land: Understanding memory side-channel hazards in SGX. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 2421–2434.

[127] Xiaoguang Wang, SengMing Yeoh, Robert Lyerly, Pierre Olivier, Sang-Hoon Kim, and Binoy Ravindran. 2020. A framework for software diversification with {ISA} heterogeneity. In *23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020)*. 427–442.

[128] Xiaoguang Wang, SengMing Yeoh, Pierre Olivier, and Binoy Ravindran. 2020. Secure and efficient in-process monitor (and library) protection with Intel MPK. In *Proceedings of the 13th European workshop on Systems Security*. 7–12.

[129] Yuanpeng Wang, Ziqi Zhang, Ningyu He, Zhineng Zhong, Shengjian Guo, Qinkun Bao, Ding Li, Yao Guo, and Xiangqun Chen. 2023. Symgx: Detecting cross-boundary pointer vulnerabilities of sgx applications via static symbolic execution. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*. 2710–2724.

[130] Richard Wartell, Vishwath Mohan, Kevin W Hamlen, and Zhiqiang Lin. 2012. Securing untrusted code via compiler-agnostic binary rewriting. In *Proceedings of the 28th Annual Computer Security Applications Conference*. 299–308.

[131] Jean-Luc Watson, Sameer Wagh, and Raluca Ada Popa. 2022. Piranha: A {GPU} platform for secure computation. In *31st USENIX Security Symposium (USENIX Security 22)*. 827–844.

[132] Yecheng Xiang, Yidi Wang, Hyunjong Choi, Mohsen Karimi, and Hyoseung Kim. 2021. Aegisdnn: Dependable and timely execution of dnn tasks with sgx. In *2021 IEEE Real-Time Systems Symposium (RTSS)*. IEEE, 68–81.

[133] Meng Xu, Kangjie Lu, Taesoo Kim, and Wenke Lee. 2017. Bunshin: compositing security mechanisms through diversification. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. 271–283.

[134] Fangfei Yang, Bumjin Im, Weijie Huang, Kelly Kaoudis, Anjo Vahldiek-Oberwagner, Chia-Che Tsai, and Nathan Dautenhahn. 2024. Endokernel: A Thread Safe Monitor for Lightweight Subprocess Isolation. In *33rd USENIX Security Symposium (USENIX Security 24)*. 145–162.

[135] Kenichi Yasukata, Hajime Tazaki, Pierre-Louis Aublin, and Kenta Ishiguro. 2023. zpoline: a system call hook mechanism based on binary rewriting. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*. 293–300.

[136] SengMing Yeoh, Xiaoguang Wang, Jae-Won Jang, and Binoy Ravindran. 2024. sMVX: Multi-Variant Execution on Selected Code Paths. In *Proceedings of the 25th International Middleware Conference*.

[137] Ruiyi Zhang, Lukas Gerlach, Daniel Weber, Lorenz Hetterich, Youheng Lü, Andreas Kogler, and Michael Schwarz. 2024. {CacheWarp}: Software-based

Fault Injection using Selective State Reset. In *33rd USENIX Security Symposium (USENIX Security 24)*. 1135–1151.

[138] Sheng Zhang, Adrian Tang, Zhewei Jiang, Simha Sethumadhavan, and Mingoo Seok. 2018. Blacklist core: Machine-learning based dynamic operating-performance-point blacklisting for mitigating power-management security attacks. In *Proceedings of the International Symposium on Low Power Electronics and Design*. 1–6.

[139] Yuhao Zhang, Yifan Chen, Shing-Chi Cheung, Yingfei Xiong, and Lu Zhang. 2018. An empirical study on tensorflow program bugs. In *Proceedings of the 27th ACM SIGSOFT international symposium on software testing and analysis*. 129–140.

[140] Ziqi Zhang, Chen Gong, Yifeng Cai, Yuanyuan Yuan, Bingyan Liu, Ding Li, Yao Guo, and Xiangqun Chen. 2023. No Privacy Left Outside: On the (In-) Security of TEE-Shielded DNN Partition for On-Device ML. *arXiv preprint arXiv:2310.07152* (2023).

[141] Size Zheng, Yun Liang, Shuo Wang, Renze Chen, and Kaiwen Sheng. 2020. Flextensor: An automatic schedule exploration and optimization framework for tensor computation on heterogeneous system. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 859–873.