Sentry: Authenticating Machine Learning Artifacts on the Fly

Andrew Gan
gan35@purdue.edu
Purdue University
West Lafayette, IN, USA

Zahra Ghodsi
zahra@purdue.edu
Purdue University
West Lafayette, IN, USA

Abstract

Machine learning systems increasingly rely on open-source artifacts such as datasets and models that are created or hosted by other parties. The reliance on external datasets and pre-trained models exposes the system to supply chain attacks where an artifact can be poisoned before it is delivered to the end-user. Such attacks are possible due to the lack of any authenticity verification in existing machine learning systems. Incorporating cryptographic solutions such as hashing and signing can mitigate the risk of supply chain attacks. However, existing frameworks for integrity verification based on cryptographic techniques can incur significant overhead when applied to state-of-the-art machine learning artifacts due to their scale, and are not compatible with GPU platforms. In this paper, we develop Sentry, a novel GPU-based framework that verifies the authenticity of machine learning artifacts by implementing cryptographic signing and verification for datasets and models. Sentry ties developer identities to signatures and performs authentication on the fly as artifacts are loaded on GPU memory, making it compatible with GPU data movement solutions such as NVIDIA GPUDirect that bypass the CPU. Sentry incorporates GPU acceleration of cryptographic hash constructions such as Merkle tree and lattice hashing, implementing memory optimizations and resource partitioning schemes for a high throughput performance. Our evaluations show that Sentry is a practical solution to bring authenticity to machine learning systems, achieving orders of magnitude speedup over a CPU-based baseline.

CCS Concepts

• **Computer systems organization → Parallel architectures**; • **Security and privacy → Symmetric cryptography and hash functions**; • **Theory of computation → Parallel algorithms**; • **Computing methodologies → Machine learning**.

Keywords

Machine Learning Security, GPU Acceleration, Supply Chains

ACM Reference Format:
Andrew Gan and Zahra Ghodsi. 2025. Sentry: Authenticating Machine Learning Artifacts on the Fly. In *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security (CCS '25), October 13–17, 2025, Taipei, Taiwan.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3719027.3765070

1 Introduction

Machine learning (ML) supply chains describe the complex series of steps carried out by multiple parties which characterize dataset generation, training, fine-tuning, testing, and distribution of ML products. Similar to software supply chains, ensuring the security of ML supply chains is critical to the overall security of an ML product. Unfortunately, unlike software supply chain security which has been the subject of much prior work [21, 41, 64], ML supply chain security has only recently gained attention. Recent work has identified ML supply chain attacks that can compromise the security of ML systems. Carlini et al. [11] discovered that datasets published online and hosted through a URL can be poisoned when an adversary obtains an expired domain name and gains the ability to modify the hosted data arbitrarily. Gu et al. [23] introduced backdoored neural networks which behave maliciously on inputs with an attacker chosen backdoor trigger, and can be covertly substituted with benign models in ML supply chains. Similarly, maliciously modifying a pre-trained model can enable an adversary to attack users fine-tuning the model for specific applications and fully compromise the privacy of the fine-tuning data [17]. Similar supply chain attacks have already occurred in the real world [22] with a recent example of an insider attack where an intern sabotaged internal ML models at ByteDance [6]. In the examples above, the attacks are possible due to a lack of authenticity verification of the artifacts (i.e., datasets or models) in existing ML systems.

A large body of work exists to secure generic software supply chains [41, 52, 64], allowing developers to verify certificates of operations and artifacts across the supply chain steps. In practice, the unique characteristics of ML supply chains including involved artifacts (datasets and models) and backend computing platforms (GPU servers) introduces challenges in applying software supply chain security solutions to ML supply chains. For example, when using Sigstore [41] to sign a model artifact, the model files should be serialized into a stream of bytes and passed sequentially through a cryptographic hash function to obtain a short digest which is then incorporated in a payload and cryptographically signed. This approach has several drawbacks. The feasibility of insider attacks demonstrates the need for verifying the integrity of artifacts regularly and ideally every time before use. However, sequential CPU hashing can incur significant overhead when applied to state-of-the-art ML models with millions or billions of parameters [40]. For example, the GPT2-XL has over 1.6 billion parameters and takes 34 seconds to be saved as a PyTorch pth file and up to 26 seconds to hash depending on the hashing algorithm. This incurred overhead negatively impacts the throughput of ML systems and can create scenarios where users have to choose between efficiency and security. Additionally, using CPU-based solutions requires the ML artifacts (e.g., the model) to be loaded into CPU memory to be hashed. This requirement is not compatible with efficient GPU data movement solutions such as NVIDIA GPUDirect [44] which bypasses the CPU and can load models from storage or remote GPUs directly into GPU memory. As such, a solution to authenticate the integrity of ML artifacts on the GPU is required.

In this paper, we present Sentry, the first framework to authenticate the integrity of machine learning artifacts on the fly when they are loaded into GPU memory. Sentry closes up the gaps in securing ML supply chains by providing an efficient solution to artifact authentication which is scalable to state-of-the-art models and datasets. Sentry uses the Compute Unified Device Architecture (CUDA) to implement novel GPU architectures for cryptographic hash constructions, incorporating dedicated memory optimizations and new adaptive resource partitioning schemes for high throughput performance. Sentry proposes new design strategies to enable granular integrity checks for ML artifacts (such as per-layer or per-model). We instantiate Sentry in Python for ease of use by ML practitioners as a generic framework for ML artifact trust on the GPU for ML supply chain security solutions.

In summary, this paper makes the following contributions:

- GPU implementation of cryptographic hash constructions based on *Merkle tree* and *lattice hashing* supporting various underlying hash functions and incorporating GPU optimization strategies such as shared memory computation and streaming for pipelining operations.
- Model and data authentication modules implemented on GPU and specifically tailored to ML systems with memory fragmentation compatibility and support for mixed-source datasets. Our on-the-fly artifact authentication achieves up to 269× and 180× speedup compared to a CPU-based baseline for model and datasets respectively.
- A Python-based and modular library, namely Sentry, for *ML artifact authentication on GPU* with support for efficient GPU data movement solutions such as GPUDirect for both model loading and data processing pipeline.

2 Background

The machine learning supply chain (MLSC) describes the operations from dataset collection and preparation, to model training, model evaluation, and service deployment. An MLSC can be described by *artifacts* including datasets, algorithms, models and configurations, and ML frameworks which are used during its life cycle. Due to the scale and complexity of curating datasets and training models, ML hubs (such as ONNX Model Zoo [50], Hugging Face [15], Kaggle [31], ModelScope [39], etc.) which store and distribute ML artifacts have gained popularity. These ML hubs provide inputs to MLSCs at different steps or host products generated from them.

ML artifacts can be attacked at different stages leading to system security compromises which necessitate integrity verification at every step of the pipeline. In this work, we focus on authentication of datasets and models which we refer to as "ML artifacts".

2.1 Software Artifact Signing

Software artifact signing has emerged as a promising mitigation for software supply chain attacks. A signature over a software package or artifact allows a user to verify the integrity of the artifact they receive and ensure that it has not been tampered with en route. Sigstore [41] is a framework for signing software to achieve the aforementioned goal. Sigstore uses OpenID Connect (OIDC) with a public-key infrastructure (PKI) for user authentication, incorporates digital signatures for artifacts, and includes logging infrastructure (i.e., transparency log) for identities and artifact signatures which are accessed during verification process. During signing and verification, information regarding how an artifact is signed is inserted into an attestation payload. Figure 1 illustrates the structure of Sigstore attestation payload that we adopt in this work. The payload is a JSON formatted data structure where the innermost layer is an in-toto Statement [28, 64] containing Subject arrays with information about the artifacts the attestation applies to and their digest values. The next layer wraps the in-toto Statement with the signature using a DSSE Envelope [1]. The outermost layer constitutes the Sigstore Bundle including signature verification material such as certificate chains or a public key together with the DSSE Envelope. Combined with the transparency log, the attestation payload enables secure verification by any party who wishes to use the signed software and ensure its authenticity. To sign an artifact, Sigstore computes the hash of the (serialized) content to obtain a short digest, which it then signs (for example using the ECDSA [62] algorithm). The end-user obtaining the artifact repeats the same steps of computing the content hash and verifies the signature. To accommodate the scale of ML artifacts, we focus on artifact hash computation and provide a brief primer on cryptographic hashing algorithms next.

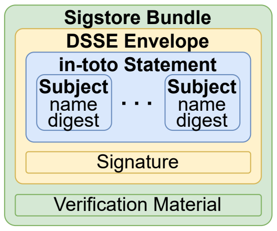


Figure 1: Sigstore payload structure.

2.2 Cryptographic Hash Constructions

Cryptographic hash functions (CHF) map arbitrary length data to a short fixed-length digest which satisfy specific security properties such as collision resistance (i.e., it is difficult to find two inputs that produce the same output) and preimage resistance (i.e., it is difficult to find an input for a given output) [53]. CHFs are constructed from *one-way compression functions* which map a fixed-length input to a fixed-length digest. Compression functions are designed to be easy to compute but hard to reverse. Examples of compression functions include SHA-2 [42], SHA-3 [9], and Blake2 [5]. Several construction exist which create a CHF from a compression function as described below and summarized in Figure 2.

Merkle–Damgård: In Merkle-Damgård construction [13, 38], a padding function is applied to the message to be hashed in order to create an input with size which is a multiple of a fixed number, namely the block size. The compression function is then applied sequentially to each data block, each time combining a block of input with the output from the previous round (set as an initial fixed value for the first block). The final hash output is the message digest.

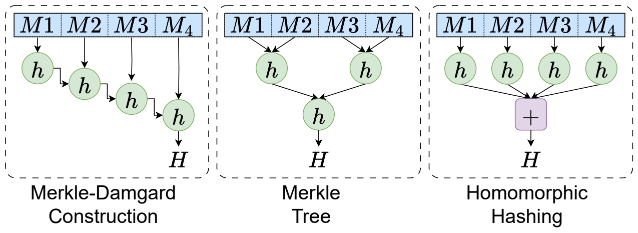


Figure 2: Cryptographic hash constructions based on Merkle-Damgård, Merkle tree, and homomorphic hashing from one-way compression function h.

Merkle Tree: The Merkle tree [37] is a tree-based hashing algorithm which starts with dividing the input message into blocks and computing the digest of each block independently using the compression function. The resulting digests are then reduced to one in a tree structure. At each tree level, two neighboring nodes are processed by the compression function to obtain one output node. If a layer has an odd number of nodes, a padding node is added to the layer. Finally, the root node is the message digest. The Merkle tree construction provides several benefits. First, it eliminates the sequential dependency of the digest computation in Merkle-Damgård and is more suitable for parallelization. Second, if a message block is updated, only the path from that block to the root node has to be redone instead of the entire process. This is specially important for database applications where some entries or data points may be updated at a time, yielding a more efficient final digest update computation.

Homomorphic Hashing: Homomorphic hashing constructions [7] extend the idea of efficient digest update when a small change in the input message occurs. Bellare and Micciancio [8] proposed specific constructions under this paradigm based on the hardness of specific computational problems in groups. One of their proposed constructions was the *lattice hash* which was later formalized in subsequent work [34]. Lattice hash applies a compression function to message blocks, and combines the resulting digests with a component-wise vector modular addition. An important property in lattice hash is set homomorphism, meaning that for two disjoint sets S_1 and S_2 we have $\texttt{LtHash}(S_1) + \texttt{LtHash}(S_2) = \texttt{LtHash}(S_1 \cup S_2)$. It is also true that $\texttt{LtHash}(S_1 \cup S_2) - \texttt{LtHash}(S_1) = \texttt{LtHash}(S_2)$. This set homomorphism property results in very efficient digest update computation. Additionally, similar to the Merkle tree construction, lattice hash computation is highly parallelizable. The lattice addition of k inputs where M_i is the i-th input, with each input broken into n partitions of d bits, can be expressed as:

$$\texttt{LtHash}_{n,d}(\{M_1, ..., M_k\}) = \sum_{i=1}^{k} h(M_i) \mod q \tag{1}$$

where q is the modulo and h is the compression function. Lewi et al. [34] set $q = 2^d$ which results in a simple modular operation by discarding any carry-over values after addition.

The addition operation in lattice hash is *order invariant*, which means that the order in which individual digests are accumulated into the final value does not change the result. Therefore, message block indices are incorporated into the digests before addition for security. Nevertheless, the order invariant property of lattice hash provides optimization opportunities on GPU which we will detail later.

2.3 GPU Architecture

We provide a brief summary of the main GPU features, and refer the readers to other references for more detail [46]. **GPU Execution Model:** GPUs follow a single instruction multiple thread execution model, enabling massive thread-level parallelism. Tasks offloaded to the GPU are initiated by the CPU through asynchronous kernel calls. Any data dependency in a sequential step then requires synchronization with the CPU. In NVIDIA GPUs, threads are organized hierarchically into warps, blocks, and grids. Warps are the smallest execution unit consisting of 32 threads, and should access coalesced memory with minimal control divergence for optimal performance. A thread block consists of up to 1024 threads assigned to a streaming multiprocessor. Multiple blocks are combined to form a grid. A program performance can be improved through the use of *CUDA streams*. A stream is a queue of GPU operations which are executed in a specific order. Different streams can be executed concurrently, allowing for overlapping of memory transfers and kernel executions for an improved performance.

GPU Memory: A GPU is composed of a memory hierarchy with different access times and capacities. Those are the registers, shared memory, global memory, and constant memory. Registers are local to each thread and quickest to access. Shared memory is shared between threads in a block and are slower to access than registers. Global memory is the largest memory and stores data for the lifetime of the program. It has the longest access time. Constant memory is a read-only memory which allows multiple threads to read from the same address through broadcasting its value to all reading threads at once.

GPUDirect: To enhance data movement for GPUs and mitigate the increasing time spent loading data, NVIDIA has developed a family of technologies called GPUDirect [44]. This technology provides a direct path between storage and GPU memory (GPUDirect Storage) as well as between GPUs and a third party device through Remote Direct Memory Access (GPUDirect RDMA). Therefore, data movement can bypass the CPU and eliminate required buffer copies of data, improving performance.

3 System and Threat Model

In this section, we detail our setting including the parties involved and describe attacker capabilities and goals.

3.1 System Model

We consider the following roles:

- **ML Hubs** are systems that host and distribute ML artifacts such as models and datasets.
- **Dataset Providers** are parties collecting or preparing datasets for ML tasks (training or inference) which they can upload to ML hubs.
- **Model Providers** are parties training models which they upload as pre-trained models to ML hubs.

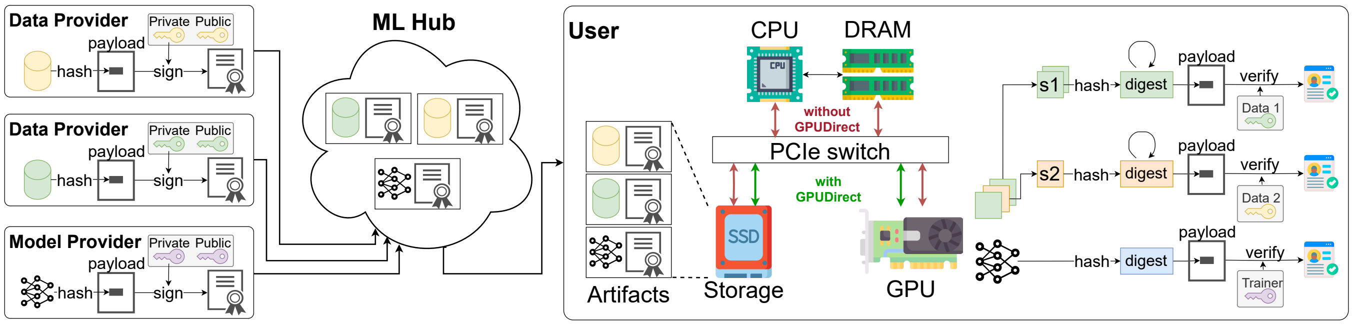


Figure 3: High-level description Sentry **with data provider, model provider, ML hub, and user roles. Model and data providers augment their artifact with an authenticated payload before uploading them to the hub. Any user accessing artifacts on the hub can verify their authentication on the fly as artifacts are loaded on GPU memory for ML tasks.**

- **Users** are parties that obtain an ML artifact from a hub (dataset or model) and use it for ML tasks (inference or fine-tuning).

We simplify the system description by abstracting out the parties that provide identity or monitoring services such as certificate authorities, identity and artifact logs, and log monitors as described in Section 2.1, and refer the readers to corresponding systems [41] for more detail.

3.2 Threat Model

The adversary's goal is to implant a tampered ML artifact into the supply chain, such as backdoored models [24] or poisoned datasets [10] which can be carried out by internal [6] or external [24] attackers. We consider a scenario where the adversary tampers with the ML artifacts at any point before they are loaded into GPU memory, but assume secure Sentry users and providers running signing and verification as well as existence of a secure public key infrastructure. We envision the following types of compromise:

(1) An adversary can perform man-in-the-middle attack targeting an artifact in transit between an artifact provider and a hub, or between a hub and the user.
(2) An adversary can tamper with the artifacts residing on user storage.
(3) A hub can be compromised by an adversary.

ML artifact compromises can sabotage the downstream ML system performance and undermine security guarantees. Sentry lays the groundwork for a paradigm where authentication is always performed at runtime which is practical due to Sentry 's small overhead. We note that the assumptions on the secure user can be further relaxed using GPU-based secure enclaves [65, 67] as we discuss in Section 6.

4 Sentry Framework

4.1 Overview

In this section, we provide a high-level overview of Sentry design as depicted in Figure 3. The signing ecosystem in Sentry is implemented through Sigstore [41] which provides mechanisms to bind signatures to identities and incorporates signing procedure attestation as described in Section 2.1. Accordingly, Sentry creates authenticated payloads for each ML artifact. Sentry consists of two main components for model and dataset authentication. We describe the signing and verification flow for each component below.

4.1.1 Model Authentication. As previously mentioned, existing solutions for software signing require the artifact to be loaded into CPU memory for hash and signature computation which incurs significant overhead for state-of-the-art ML systems as we demonstrate later in Section 5.1. Sentry overcomes this limitation by performing authentication operations on ML artifacts directly on the GPU.

We start by describing the signing flow for a model provider, assuming model training is completed or model provider is saving a checkpoint of the model currently residing on GPU memory. Sentry generates the attestation payload associated with the model on the CPU with dummy digests. Then, Sentry moves the payload into GPU memory, computes model digests on GPU, and inserts the digests into the payload. The private key associated with the signing key-pair is also copied to GPU memory. Afterwards, Sentry hashes the payload and signs the payload digest on GPU using the key, and obtains the signature.

Sentry implements several hashing constructions and provides model digest computation at two granularity levels. The first granularity level only computes a single digest for the entire model. The second granularity level computes per-layer digest computation in addition to the final digest which will be included in the authenticated payload. This flexibility enables Sentry to be applicable in settings where an audit trail is required to verify computations such as fine-tuning where only a subset of model layers are modified and the rest are frozen. In order to verify a pre-trained model obtained from a third party or a hub, a user applies Sentry to recompute model digests on the GPU and verify the signature on the associated payload. Sentry incorporates several GPU optimizations by performing efficient memory accesses when hashing an ML model loaded in fragmented GPU memory, in addition to carefully designing concurrent computations for a highly efficient runtime. In addition to runtime benefits, Sentry can authenticate models within ML systems that integrate efficient GPU data movement solutions such as GPUDirect Storage which bypass the CPU as shown in Figure 3.

4.1.2 Dataset Authentication. Dataset authentication in ML systems faces additional challenges since datasets can originate from multiple sources and used within the same ML pipeline. For example, a training dataset can be aggregated by combining several datasets, or a server might perform inference computation on batched queries from several clients. Furthermore, training datasets (e.g., images) are often compressed before they are stored, requiring developers to build multi-stage data processing pipelines that load, decode, and perform transformations such as cropping or resizing on datasets. These data processing pipelines, typically executed on the CPU, have become a bottleneck. Including an authentication step into the data processing pipeline will exacerbate this problem and limit throughput further. Finally, ML training pipelines typically include a random shuffling of the data which in the case of dataset sourced from multiple data providers, means that each data batch loaded to the GPU could contain data points from multiple providers.

Sentry is designed to address the above challenges, providing the flexibility required to preserve the functionality of ML operations in addition to authentication guarantees with minimal overhead. For dataset authentication, Sentry augments each data point to include a digest in addition its value (raw data), label, and provider source ID. The digest is computed on raw data prior to any encoding step for storage. For example, the digest of an image is the hash computed over the 3D byte tensor. Each data provider will then generate an attestation which contains the final digest over the entire dataset and is signed by the data provider. Sentry incorporates lattice hashing for dataset digest computation which enables simple digest updates when individual data points are added or removed. Utilizing lattice hashing for data authentication will further enable Sentry to authenticate randomly shuffled datasets efficiently as detailed next.

To enable efficient data processing and mitigate the added overhead of data authentication, Sentry integrates NVIDIA Data Loading Library (DALI) [48] which accelerates data processing operations including loading, decoding, and transformations by offloading them to the GPU, resulting in improved performance and scalability for training and inference. Within DALI, data authentication is performed as an additional step in the data processing pipeline as depicted in Figure 3. Specifically, Sentry keeps track of a running sum of digests for each data source using lattice hash as batches of data are loaded into the GPU. For each batch, Sentry uses a dictionary to map each data source to its data points in the batch, and computes per-source digests before adding them to their corresponding digest running sum buffers. Afterwards, the data batch goes through the rest of the pipeline which implements the remaining data processing operations and performs a training or inference round. Once all batches from the dataset are processed, the digest buffers contain the final dataset digests and are used in signature verification.

In the next section, we detail the implementation of Sentry digest computation on GPU and discuss incorporated design optimizations. We start by describing algorithms for hashing specifically tailored to ML model artifacts, followed by Sentry data authenticator component.

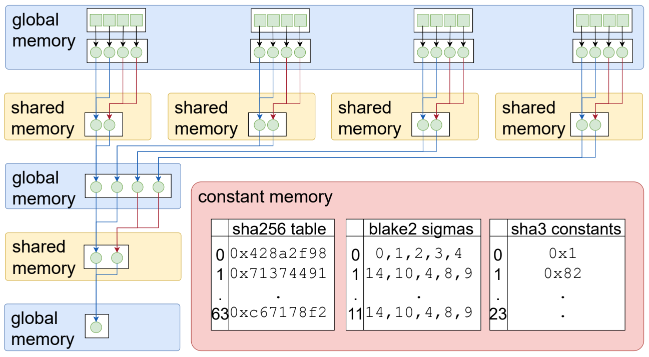


Figure 4: Depiction of block hashing and tree reduction kernels. Shared memory is used for storing intermediate values within a thread block, global memory for collecting results from different blocks, and constant memory for storing lookup tables with hard-coded values. The input read from global memory is reduced in shared memory, whereas the last intermediate results in shared memory are reduced to the final result in global memory to minimize the number of memory transfers.

4.2 Model Hash Computation on GPU

Efficient hash computation of large ML model artifacts requires exploiting parallelization opportunities on the GPU. To do so, we focus on two hash constructions based on Merkle tree and lattice hash introduced in Section 2.2 which are amenable to parallel execution.

We start by describing how ML models are defined and loaded into GPU memory. In PyTorch, models are described using a Python dictionary object called a `state_dict` which maps each layer to its parameter tensor. PyTorch implements dynamic memory allocation optimizations on the backend with CUDA which uses the smallest available free blocks for allocation requests or allocates a new block. As a consequence, when a model is loaded into GPU memory, each parameter tensor is stored in a contiguous block of memory, but different layers could be fragmented across the memory space. We note that this behavior is not unique to PyTorch, and is implemented in other ML frameworks as well. For example, in Tensorflow each model layer is allocated a Tensorflow tensor, resulting in the scattering of model weights across GPU memory. Based on this insight, we propose different designs for hashing ML models on the GPU with different memory and runtime trade-offs. We start by describing our implementation of the Merkle tree hash on GPU, followed by our proposed architecture for lattice hash.

4.2.1 Merkle Tree Hashing on GPU. Sentry implementation of the Merkle tree hashing consists of two efficient CUDA kernels as depicted in Figure 4. The first kernel, namely the *block hashing kernel*, hashes fixed size data blocks in global memory into digests. In this kernel, each thread is assigned a single data block (with default size of 8192 bytes) to hash. The resulting digests are written to an output buffer in global memory. The second kernel, namely the *reduction kernel*, takes the output buffer of the hashing kernel

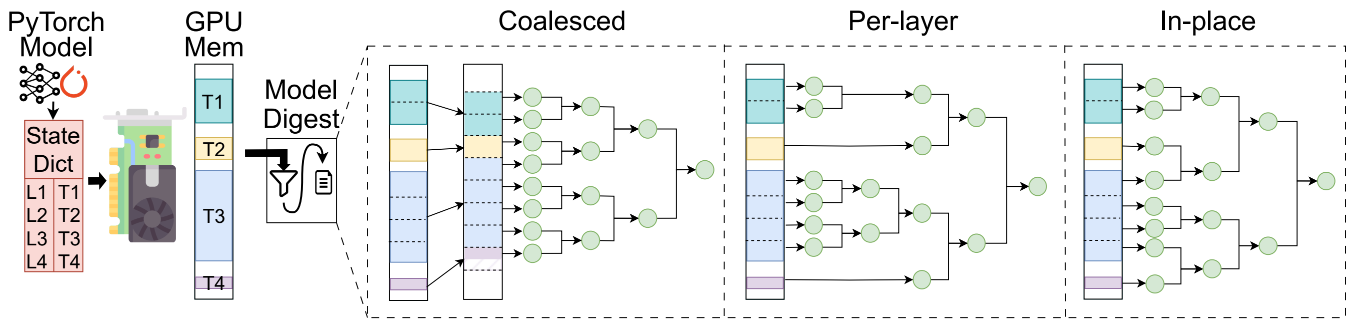


Figure 5: Depiction of three model hashing implementations. The model is defined in PyTorch using a dictionary `state_dict`, which maps each layer to the corresponding weight tensor. After the model is loaded on GPU memory, the layers are fragmented across GPU memory space. We propose coalesced, per-layer, and in-place implementations for computing the model hash.

as input, and in each step reduces the size in half through a Merkle tree layer in shared memory, finally generating the output digest as shown in Figure 4. We describe the details of the reduction kernel below.

The reduction kernel takes as input the digests to be reduced and outputs the reduced digests computed by each thread block. In other words, if a thread block has k threads, each thread block reduces $2k$ digests into one digest. With n digests as input to a reduction kernel, the output consists of $n/2k$ digests. The reduction kernel is repeatedly called until a single digest remains. To prevent unnecessary memory transfers, we use two temporary memory buffers and designate one as the input buffer and the other as the output buffer. At the end of each reduction kernel call, we switch the designation of buffers such that the output buffer (containing the layer digests) is designated as the input buffer and the previous input buffer is now the output buffer. As a result, the next call to the reduction kernel can start immediately without the need for any memory copy operations.

Within a thread block, each thread hashes two neighboring digests and stores the resulting digest into shared memory. As mentioned in Section 2.3, shared memory allows for quicker access and resource sharing between threads in a thread block. Additionally, we use the constant memory to store the lookup tables for hard-coded values (such as the key values for SHA256 and sigma values for BLAKE2) used in hashing algorithms as it allows read-only access and can broadcast a value to multiple threads with one read. Sentry supports SHA256, BLAKE2B, and SHA3 (based on [69]) as compression functions for Merkle tree hashing on the GPU. In each reduction step, the number of threads within a thread block are reduced by half. Afterwards, we perform an explicit synchronization of all threads in the thread block to prevent race conditions between tree layer reductions. Once the number of active threads falls below the warp size of 32, thread block synchronization is not needed, since threads in a warp execute instructions in lockstep, offering implicit synchronization. The final call to the reduction kernel generates the output digest of the entire Merkle tree.

Next, we detail hash computation of ML models based on our Merkle tree hashing kernel. As previously discussed, PyTorch optimizations for memory allocation results in model layers being fragmented across the memory space after being loaded on the

Algorithm 1 Sentry Coalesced Hashing

```
Input: Model state_dict = $\{L_1 : T_1, \ldots, L_n : T_n\}$
Output: Model hash value
buffer $\leftarrow T_1 \| \ldots \| T_n$
$m \leftarrow$ number of blocks in buffer
for $j = 1$ in $m$ do
    hash_blocks[$j$] $\leftarrow$ BlockHash(buffer[$j$])
end for
hash $\leftarrow$ HashReduce(hash_blocks)
return hash
```

Algorithm 2 Sentry Per-layer Hashing

```
Input: Model state_dict = $\{L_1 : T_1, \ldots, L_n : T_n\}$
Output: Model hash value
for $i = 1$ to $n$ do
    $m \leftarrow$ number of blocks in $T_i$
    for $j = 1$ in $m$ do
        hash_blocks[$i$][$j$] $\leftarrow$ BlockHash($T_i$[$j$])
    end for
    layers_hash[$i$] $\leftarrow$ HashReduce(hash_blocks[i])
end for
hash $\leftarrow$ HashReduce(layers_hash)
return hash, layers_hash
```

GPU, making model digest computation and efficient memory accesses more challenging. We propose three architectures (namely coalesced, per-layer, and in-place) using Merkle hashing which address this issue as depicted in Figure 5 and described next.

Coalesced Hashing The coalesced hashing scheme first coalesces the model layers into a contiguous block of memory by iterating through the model `state_dict` and copying layer tensors into a contiguous memory block. The size of the contiguous memory block is a multiple of the data block size, and is padded at the end if the total model size is smaller. The details of the above steps for this hashing scheme are described in Algorithm 1, where buffer represents the contiguous memory block. Afterwards, the Merkle tree hashing kernel (described previously) operates on the contiguous memory block to obtain the final model digest. The coalesced scheme provides a baseline implementation and requires additional memory allocation to store the coalesced model, resulting in a 2× memory requirement with respect to the model size. Furthermore, the memory copy operations to move layer tensors into the coalesced block impose additional runtime overhead. However, we note that GPU to GPU memory copy can be performed quite efficiently in modern GPUs which have a memory bandwidth above 700 GB/s.

Per-layer Hashing The per-layer hashing scheme operates on each layer tensor (stored in a contiguous block of memory by PyTorch) separately to obtain per-layer digest values, which are then reduced into a single model digest. We outline the operations in per-layer hashing in Algorithm 2. Each layer tensor uses a Merkle tree hashing kernel to generate the corresponding layer digest. We assign a dedicated CUDA stream to each layer, executing per-layer Merkle tree kernels concurrently. After waiting for all layer digests to be computed by synchronizing the created CUDA streams, a final Merkle tree reduction kernel computes the model digest The hashing scheme returns the per-layer digests and the final model digest, providing two granularity levels as discussed in Section 4.1.1. Furthermore, the per-layer hashing scheme mitigates the additional memory requirements of the coalesced scheme.

In-place Hashing The in-place hashing scheme modifies the block hashing kernel to operate on fragmented layer tensors at once as specified in Algorithm 3. To do so, we design a lookup table which stores, for each range of threads assigned to work on the same weight tensor, the starting address, block size, and tensor size for computing corresponding digests. Accordingly, when the block hashing kernel is launched, each thread within the kernel retrieves its input data by accessing the lookup table (at the row where the index range includes the thread index) and obtaining the starting address and size of input. The input size for each thread within the kernel is the same except for the threads working on the last block of every weight tensor, whereby the input data range is the starting address of the thread to the end of the tensor, which is typically less than the fixed block size. This method is used instead of simply padding the last block of data up to the block size because in-place hashing reads data directly from the state dictionary, which should be kept unchanged. After the block hashing kernel computes all the digests, the Merkle tree reduction kernel operates on the digests to obtain the final model digest.

4.2.2 Lattice Hashing on GPU. Sentry implementation of the lattice hashing follows the high level structure of the Merkle tree implementation by incorporating a block hashing kernel and a reduction kernel. However, the main operations in each kernel of lattice hash is different from Merkle tree. The block hashing kernel incorporates Blake2b based on the GPU implementation in [69] for digest computation, and the reduction kernel uses the lattice addition (with GPU implementation described later) as the reduction operation in a binary tree to perform summation to reduce a group of digests into one. Because of the order invariance property of lattice hash (Section 2.2), we concatenate each data block with the block index before computing the hash digest for that block to mitigate reordering attacks.

Similar to Merkle tree implementation, lattice hashing incorporates *coalesced*, *per-layer*, and *in-place* hashing schemes. While coalesced and in-place lattice implementations follow the structure of Merkle tree closely, the per-layer architecture incorporates an optimization opportunity based on the order invariant property of lattice hash as shown in Figure 6 and described in Algorithm 4. Specifically, after layer digests are computed, they can be accumulated into a buffer containing the running sum in any order to produce the final digest. We use this property to accumulate layer digests in ascending order of their size (represented by the OrderLayerIndices function in Algorithm 4), ensuring that smaller layer digests can be accumulated into the running sum buffer while larger layer digests are still being reduced. In contrast, Merkle tree per-layer hash computation requires all layer digests to be computed first, before they can be reduced to a single digest. This limitation forces smaller layers to wait for larger layer digest computation before their digest values can be reduced into the final model digest.

Algorithm 3 Sentry In-place Hashing

```
Input: Model state_dict = $\{L_1 : T_1, \ldots, L_n : T_n\}$
Output: Model hash value
$k \leftarrow 1$
for $i = 1$ to $n$ do
    $m \leftarrow$ number of blocks in $T_i$
    for $j = 1$ in $m$ do
        hash_blocks[$k$] $\leftarrow$ BlockHash($T_i[j]$)
        $k \leftarrow k + 1$
    end for
end for
hash $\leftarrow$ HashReduce(hash_blocks)
return hash
```

Algorithm 4 Sentry Optimized Lattice Per-layer

```
Input: Model state_dict = $\{L_1 : T_1, \ldots, L_n : T_n\}$
Output: Model hash value
for $i = 1$ to $n$ do
    $m \leftarrow$ number of blocks in $T_i$
    for $j = 1$ in $m$ do
        hash_blocks[$i$][$j$] $\leftarrow$ BlockHash($i \| T_i[j]$)
    end for
end for
$\{o_1, \ldots, o_n\} \leftarrow$ OrderLayerIndices($1, \ldots, n$)
for $i = o_1$ to $o_n$ do
    layers_hash[$i$] $\leftarrow$ HashReduce(hash_blocks[i])
end for
hash $\leftarrow$ HashReduce(layers_hash)
return hash, layers_hash
```

Incorporating the order-invariance optimization described above, the per-layer lattice hash computation starts by launching block hashing and reduction kernels for each layer tensor in ascending order of the layer size as depicted in Figure 5. The digest and reduction steps for each layer can be performed concurrently, and are scheduled into separate CUDA streams. To collect the results, the asynchronously launched kernels are synchronized with the host device in ascending order of the layer size.

With the high level description of the lattice hashing described above, we now present the details of lattice addition and our proposed GPU architecture as depicted in Figure 6 on the right. We

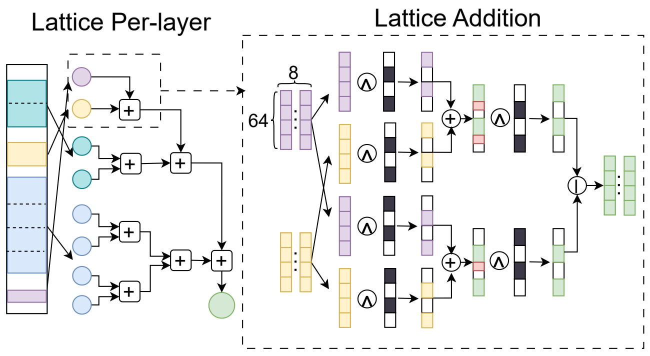


Figure 6: Details of lattice hash per-layer implementation on the left, incorporating the order-invariant property of lattice addition to sort weight tensors by size before reduction into per-layer digest and addition to running sum of model digest. The internals of lattice addition implementation for two 64-bit integers is depicted on the right.

follow the lattice hashing implementation in the Folly library [16] with digest length set to 64 bytes or 8 64-bit integers. Each 64-bit segment of a digest is further partitioned into 4 partitions of 16 bits.

To perform a lattice addition between two digest values (shown as purple and yellow 64×8 partitioned arrays in Figure 6), each partition of the first digest is added to the corresponding partition of the second digest. The addition operation is modular, with the modulo set as 2^{16} for 16-bit partitions. In other words, when two 16-bit partitions are added together, the carry over is discarded (performing the modular operation). The lattice addition kernel uses a group of 8 threads to operate on two digest values being added, with each thread working on a pair of 64-bit values (or 4 partitions) as depicted in Figure 6. Each thread performs the following operation. For each 64-bit value, the odd and even partitions are added together in parallel. This function is implemented through masking (bit-wise AND operation with zero in corresponding bits shown as black in Figure 6). The odd partitions are then added together and again masked to discard any overflow values shown in red, with the same operation repeated for even partitions. Finally, the odd and even partition results are combined using an OR operation to obtain the 64-bit output. The reduced digest from addition can be obtained by concatenating the output of 8 threads.

4.3 Dataset Hash Computation on GPU

For data authentication, Sentry incorporates a hashing step within the data processing pipeline on the GPU as previously detailed in Section 4.1.2 and shown in Figure 7. Incoming data batches are first loaded on GPU memory, then they are processed by Sentry data authenticator module which computes the data hashes (based on lattice hash), and finally they are passed through user defined transformations. The block size within the data hashing kernel is set as the size of a single sample, which are then reduced to a digest value and aggregated into a buffer holding the running sum. After all the data batches are processed, the running sum buffer holds the output digest value and is used with signature verification step.

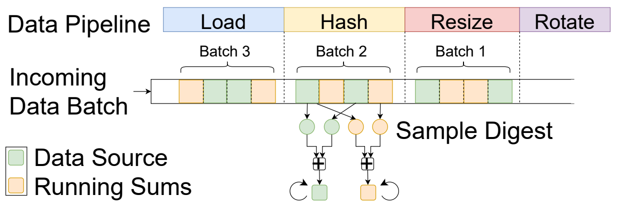


Figure 7: Sentry **pipeline for data processing for an incoming stream of data batches. After loading each batch to GPU memory,** Sentry **computes its lattice hash. Each batch is separated by data sources with corresponding digests accumulated into per-source digest sums. Afterwards, the data batch follows through the pipeline for other data transformations.**

If a dataset is created from multiple providers, random shuffling will result in batches with data points from different sources and authenticated by separate signatures. To address this setting, Sentry implements a dictionary to map each source represented in the batch to the samples in the batch that belong to the source. With a batch size of n, we spawn n threads for initial block hashing. Then, with n digests or $\frac{n}{2}$ digest pairs to sum up, we spawn 8 threads per digest pair to reduce two digests into one. A lattice hash reduction tree is spawned for each data source represented in the incoming batch. After the trees are fully reduced, we acquire the per-source digests of the data batch. Each per-source digest is added to the running sum of digests of each source.

4.4 Sentry **Library and End-to-End Operation**

Sentry is implemented as a Python library for ease of use by developers and machine learning practitioners. To incorporate the model and dataset authentication components described before in a modular way, Sentry uses the NVIDIA runtime compiler (NVRTC) to pre-compile the CUDA source code to callable modules from within Python during runtime. Under the hood, Sentry uses CUDA Python [43] to call CUDA runtime and driver APIs to manage GPU memory from within the Python code base.

We next describe an example scenario to demonstrate the end-to-end operation of Sentry. Assume a user has obtained an ML model and dataset (for example retrieved from a model hub) which currently reside in their storage. Sentry starts by loading the model to GPU memory which can be accelerated through GPUDirect Storage and bypassing the CPU. Sentry performs model authentication on the fly when the model is loaded on GPU memory. Sentry data pipeline incorporates user defined data transformations, and integrates data authentication as described in Section 4.3. The data loader for Sentry also supports reading data batches directly from storage to GPU with GPUDirect. Sentry incorporates model signing as well as model and dataset signature verification by incorporating a GPU-accelerated ECDSA algorithm (based on [18]) which signs or verifies the signature on each artifact authenticated payload on the GPU.

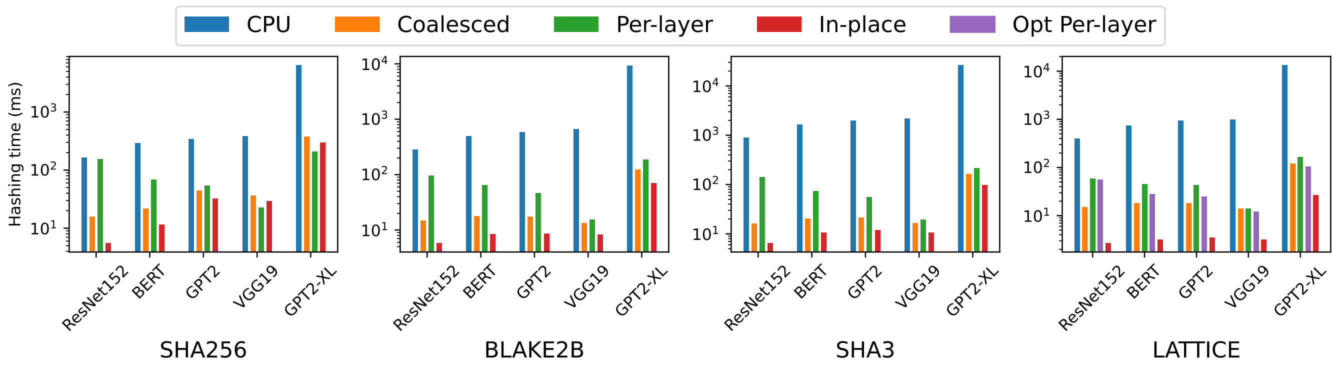


Figure 8: Runtime of Sentry **for hashing different ML models using proposed hashing architectures (coalesced, per-layer, and in-place) compared to a CPU baseline.**

5 Evaluation

In this section, we quantify the performance of Sentry and demonstrate its efficiency for ML artifact authentication.

Hardware Specifications We evaluate our GPU-based library Sentry and compare it against the naive Sigstore software signing on CPU. We run all GPU experiments on an NVIDIA RTX A6000 with CUDA version 12.8 and driver version 570.86.16. The GPU has 48 GB of GDDR6 memory, 84 stream multiprocessors (SM) and a memory bandwidth of 768 GB/sec [45]. Our data loader implementation incorporates GPUDirect Storage in compatibility mode [47] which uses POSIX APIs to handle cuFile reads and writes through CPU memory. While GPUDirect in compatibility mode does not achieve the full speedup offered by GPUDirect, it results in file load improvements as we show later. We run all CPU experiments on an AMD Ryzen Threadripper PRO 5995WX with 64-Cores at 4.5 GHz with 512 GB memory.

Datasets and Models We evaluate Sentry using one image dataset, namely CIFAR10 [32]) and one text dataset, namely Hellaswag [68]. We benchmark ResNet152 [26] and VGG19 [57] models for CIFAR10 and Bert [14], and three variants of GPT2 [51] for Hellaswag datasets. CIFAR10 is an image data set where each sample is a 32×32×3 image. Hellaswag is a query data set with varying sentence lengths, using state-of-the-art generators, discriminators and source text to curate the dataset [68]. The model specifications are summarized in Table 1.

System Parameters The default block size for hashing models is set to 8192 bytes. For dataset hashing, the block size was set to 32×32×3 bytes for CIFAR10 corresponding to the size of an image, and to the size of the samples in the batch for Hellaswag where samples are padded to the same length. The maximum number of threads per thread block is set to 512 to ensure a thread block has enough resources to use in a stream multiprocessor.

5.1 Model Authentication Performance

5.1.1 Runtime Evaluation. We start by evaluating Sentry runtime for model hashing and signature verification. Figure 8 summarizes the runtime for model digest computation of Merkle tree with SHA256, BLAKE2B, and SHA3 and comparison to Sigstore (based on hashlib [25] library) baseline. Additionally, we present model digest computation runtime based on Sentry lattice hash.

Among Merkle tree variations, the in-place architecture observes the lowest runtime in most settings, achieving between 11× (for GPT2 using SHA256) and 269× (for GPT2-XL using SHA3) speedup compared to the CPU baseline with Sigstore. The GPU runtime improvements under BLAKE2B and SHA3-based Merkle tree implementations are higher than SHA256-based implementation. This result is due to SHA256 CPU implementation benefiting from specialized hardware instruction sets. Specifically, modern Intel and AMD CPUs accelerate common cryptographic operations [29] including SHA256 which grants a boost to CPU performance of this hash function. Nevertheless, Sentry SHA256-based Merkle tree implementation achieves between 1.1× and 30× speedup compared to CPU implementation across different implementations and models. Compared to in-place implementation, coalesced runtime is slightly higher consistently across different settings. This is because the coalesced implementation of Merkle tree incorporates an additional step which coalesces fragmented model layers in a contiguous memory block and therefore incurs additional overhead due to memory copy operations compared to in-place implementation. The per-layer implementation of Merkle tree typically performs worse than other implementations, since digest values for each layer has to be computed separately which limits parallelization opportunities. However, model architecture can affect concrete runtime and observed speedup as explained next.

For the largest model we benchmark, namely GPT2-XL with 1.6B parameters, we observe the greatest speedup from GPU acceleration. ResNet152 is the smallest model we benchmark with 60M parameters, but it has the most layers at 932 as detailed in Table 1. Due to the large number of layers, per-layer implementation of Merkle tree for ResNet152 incurs significantly more overhead against coalesced and in-place compared to other models as demonstrated in Figure 8. Although VGG19 is 2.4× larger than ResNet152 in terms of number of parameters, it has fewer number of layers (38 layers). As a result, per-layer implementation of Merkle tree when for VGG19 causes a smaller increase in runtime compared to coalesced and in-place hashing. This is evidently shown in Figure 8, where the per-layer runtime for ResNet152 is greater than VGG19

	Model Specifications			Additional Memory Usage		
	Weights (M)	Layers (num)	Size (MB)	Coalesced (MB)	Per-layer (MB)	In-place (MB)
ResNet152	60	932	270	270	2	4
Bert	109	199	538	538	4	4
GPT2	124	149	1077	1077	8	6
VGG19	143	38	1077	1077	8	6
GPT2-XL	1610	581	8623	8623	54	54

Table 1: Description of models and memory usage of proposed implementations over model size.

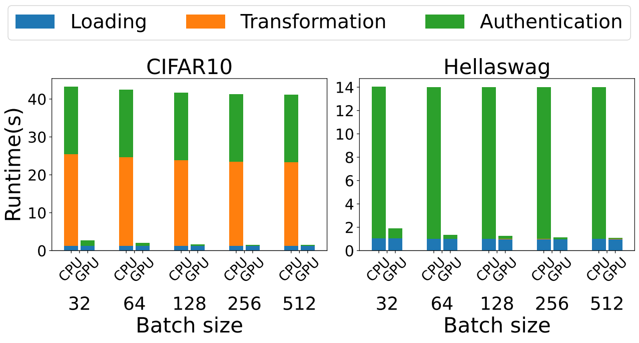
in all hashing algorithms even though VGG19 is a larger model by parameter count.

The last plot in Figure 8 shows the performance of Sentry lattice hash computation. Sigstore does not support lattice hashing, so we implement a CPU-based lattice hashing for model authentication based on Folly [16]. As the figure shows, the comparative performance of coalesced, per-layer, and in-place implementations follows the same trends as Merkle tree hash discussed above with in-place and per-layer having the best and worst runtime respectively. Specifically, compared to the CPU baseline, Sentry achieves between 5× (for per-layer on ResNet152) and 500× (for inplace on GPT2-XL) speedup. In this figure we also compare the performance of baseline per-layer lattice hash to our optimized per-layer lattice hash which benefits from the order invariant property as discussed in Section 4.2.2. The optimized per-layer hash achieves between 5% and 41% speedup over the baseline per-layer implementation.

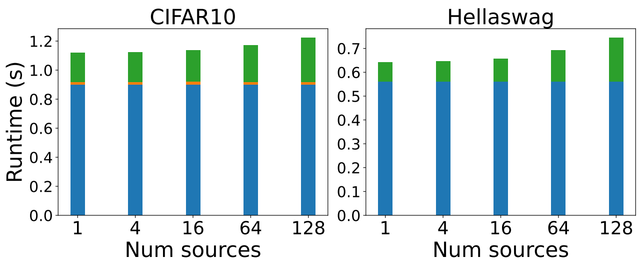
After model digests are computed, Sentry creates the attestation payloads and incorporates a GPU accelerated ECDSA implementation (based on RapidEC [18]) to sign the payloads. We note that the payload size depends on the number of digests inserted and is equal to 558 bytes for single digest and varies if per-layer digests are included, between 12 KB for VGG19 and 294 KB for ResNet152. Accordingly, the signing operation on GPU incurs a small overhead of 8.94 ms for single digest and varies between 10.7 ms and 38.4 ms for per-layer digests in VGG19 and ResNet152 respectively.

5.1.2 Memory Usage Evaluation. Table 1 presents the GPU memory usage for coalesced, per-layer, and in-place implementations. The memory usage is the same for Merkle tree and lattice hash variations. We report *additional* memory usage beyond the space needed to load the model. As the table shows, coalesced hashing requires the largest memory with additional allocation equal to the size of the model for coalescing model layers in a contiguous block. Meanwhile, the memory usage for per-layer and in-place implementations are minimal since they only use buffers to store digests produced by the initial hash block kernel. Each digest is of size 32 bytes or 64 bytes depending on hashing algorithm, which is significantly smaller than the default block size of 8192 bytes. For comparison, the baseline CPU implementation incurred 2 MB of memory usage in addition to memory allocation for the ML models.

Our results show that in-place hash implementation offers favorable performance in terms of both runtime and memory usage. However, if the underlying application requires incorporation of digest values for each layer, then per-layer implementation can be used.



(a) Sentry **data pipeline runtime compared to CPU baseline**



(b) Sentry **data pipeline runtime for varying number of data sources**

Figure 9: Runtime breakdown of Sentry **data pipeline, (a) comparing to CPU baseline, and (b) with varying number of data sources, for image (left) and text (right) datasets.**

5.2 Dataset Authentication Performance

In this section, we evaluate the performance of Sentry data pipeline. Figure 9(a) shows the breakdown of Sentry data pipeline compared to a CPU baseline (based on Sigstore) for CIFAR10 image and Hellaswag text datasets, assuming a batch size of 128. The data transformations implemented for CIFAR10 follow practices for processing images from NVIDIA's DALI tutorial page [49] and include resize, rotate, crop, mirror and normalize operations. For the text dataset, we perform padding as the sole data preprocessing step such that the inputs within a batch are of equal length. We report the runtime for processing the entire training set (50K for CIFAR and 40k for Hellaswag) for a range of batch sizes between 32 and 512 assuming 16 data sources. For Sentry, we incorporate GPUDirect Storage in compatibility mode for data loading. As shown in Figure 9(a), Sentry data pipeline on GPU achieves significant speedup over CPU baseline for both datasets, resulting in up to 27× and 10× runtime improvement for CIFAR10 and Hellaswag datasets respectively.

As a percentage of total data pipeline runtime for a batch size of 32, dataset authentication including digest computation and signature verification for CIFAR10 (Hellaswag) takes up 41% (76%) for the CPU baseline compared to 13% (5%) for Sentry. With a larger batch size of 512, the dataset authentication portion of total runtime falls to just 15% (7%). We note that data loading can take up to 84% of total data processing pipeline, with 29% runtime benefits from GPUDirect Storage running in compatibility mode. For systems

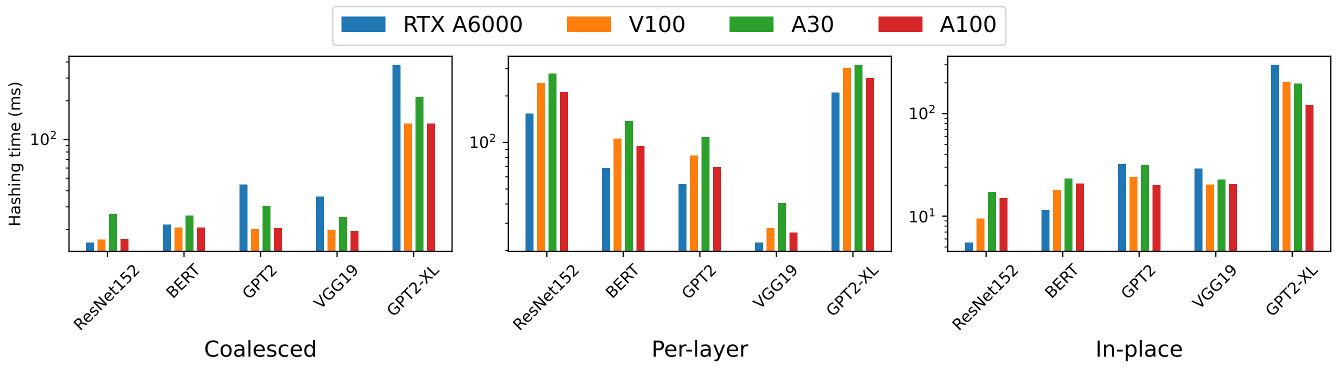


Figure 10: Runtime of Sentry **for hashing models and datasets on various GPUs.**

that support GPUDirect Storage fully, this overhead can be further reduced.

Additionally, we evaluate the performance of Sentry data pipeline with varying number of data sources as depicted in Figure 9(b) for CIFAR10 and Hellaswag datasets. We vary the data source number from 1 to 128 and use Derichlet data partitioning [27] with alpha value of 1.0 to assign data points to sources. The runtime for data processing is only impacted by the number of data sources. Therefore, while changing α changes the class distributions, it has no effect on total runtime. As Figure 9(b) shows, the increase in number of data sources slightly increases the data authentication runtime (by 15% from 1 to 128).

After dataset digest digests are computed (one per data source), Sentry verifies the signatures on each payload containing the final digest. The signature verification for all data sources is executed concurrently on the GPU, incurring a small overhead of 128 ms for 128 data sources.

Note on Storage Overhead Sentry adds attestation payloads to be stored with ML model and datasets. However, such payloads only adds a small overhead in terms of storage. To demonstrate this, assume that the payload for model contains per-layer digests and the payload for dataset contains per-sample digests with digest length of 64 Bytes. For this setting, the payload for ResNet152 (with the largest number of layers) is 294732 Bytes, adding 0.11% overhead to storage cost. For datasets, the payload file contains per-sample digests and source IDs and adds only 6.5% and 5% storage overhead to CIFAR10 and Hellaswag datasets respectively.

5.3 Cross-platform Performance and Compatibility

As previously discussed, Sentry proposes new algorithms for hashing tailored for GPUs. However, the proposed algorithms are based on common constructions such as Merkle tree or SHA256 and reproducible in CPU-based systems and across different GPU architectures. While special purpose accelerators (e.g., TPUs [30]) have to rely on CPU-based authentication, Sentry ensures compatibility and orders of magnitude speedup when GPUs are available.

In Figure 10, we evaluate the runtime of Sentry (using Merkle hashing with SHA256) across different GPUs (RTX A6000, V100, A30, and A100) for which we confirm identical digest computations. While there are variances in runtime across different constructions, we observe similar speedup trends with respect to the CPU runtime. For coalesced and in-place hashing, RTX A6000 performs best on smaller models, i.e., ResNet152 and BERT, achieving up to 13.4× and 29.9× speedup compared to the CPU baseline for coalesced and in-place respectively. For larger models such as GPT2-XL, A100 achieves the best performance for coalesced and in-place hashing scheme, reaching 47.7× and 52.4× speedup respectively compared to CPU. This varying performance across models can be attributed to the fact that workstation GPUs such as RTX A6000 have more CUDA cores but lower memory bandwidth when compared to data center GPUs such as A100, enabling a greater capacity for parallel computation but slower movement of data between GPU memory regions. Similarly, RTX A6000 performs best for per-layer hashing across all benchmarks, achieving 30× speedup over a CPU baseline, compared to A30 (least speedup) that achieves 20× runtime benefit over the CPU.

Limitations: Sentry currently only provides support for NVIDIA GPUs. However, the proposed parallel algorithms and kernels can be converted to other GPU architectures which is an interesting direction for future work. For instance, HIPIFY [3] converts CUDA to portable HIP C++ code for AMD GPUs, and CUDA frameworks can be ported to SYCL for compatibility with Intel GPUs [60].

6 Related Work

Software Security: Security issues in software supply chains have gained significant attention in recent years with several exiting work aiming to mitigate security risks. One such tool is Sigstore [56] which includes features such as OIDC identity authentication, PKI for signing, and a logging feature for signatures. To protect other parts of the software supply chain beyond just software integrity, SLSA [58] was proposed as a set of specifications on how to perform compliant operations in every step of the supply chain to protect against supply chain attacks. As a result, SLSA allows the end user to verify the inputs, code base, and metadata that create the software. SLSA makes use of in-toto attestations to maintain the claims of performing each step correctly in a statement for later verification [59]. In-toto is a framework for recording information about the steps of a software supply chain [64]. Every party involved in the supply chain logs their actions into a payload describing the steps performed and how it was performed. Another important concept in software security is the Software Bill of Materials (SBOM) to document the list of components that make up a software [55]. Nevertheless, none of the previously mentioned frameworks is specifically tailored to ML systems and scalability requirements that accompany the corresponding artifacts which we address in this work.

Machine Learning Governance: Prior work on formalizing machine learning governance include a systematization of knowledge by Chandrasekaran et al. [12] where they propose establishing identities in the ML supply chain through data and model ownership. The authors also highlighted the importance of model management tools to patch ML systems. In parallel with the goal of authenticating software, Google's Model Transparency [36] initiative proposes adopting existing software authentication tools for the machine learning life cycle, incorporating the above two solutions, Sigstore and SLSA. Besides that, model card initiatives such as those incorporated by Huggingface, incorporate information about build specifications, expected results, and the checksum of each file if given by the provider. However, there is no certain way to verify their truthfulness.

Atlas [61] presents a framework for ML pipeline attestation and integrates the Content Provenance and Authenticity (C2PA) standard for provenance. The attestation client in Atlas runs within trusted execution environments (e.g., Intel SGX) as a root of trust and protection against runtime tampering. We note that Sentry provides a related and orthogonal goal to the works mentioned above, and can be integrated within these frameworks to improve authentication performance. Specifically, existing work are limited to running artifact authentication on CPU resulting in significant overhead for state-of-the-art ML artifacts and incompatibility with efficient GPU data movement solutions such as GPUDirect Storage or RDMA.

GPU Acceleration of Cryptographic Protocols: Prior work has explored accelerating cryptographic primitives and protocols such as AES [33, 35], hash functions [69], digital signatures [18], and secure multiparty computation (MPC) [19]. GPU acceleration of MPC and homomorphic encryption (HE) is of particular interest due to applications in privacy-preserving machine learning [2, 20, 63, 66]. However, existing solutions do not apply to ML systems and lack specific design features to support ML system and artifact properties as discussed in this work.

Confidential Computing on GPU: Systems based on Trusted Execution Environments (TEE) and Confidential Computing (CC) on the CPU has gained popularity due to their ability to isolate memory from unauthorized sources and perform computations in isolation [54]. Recent developments of CC on GPU started with NVIDIA supporting such feature on the Hopper and Blackwell GPUs [4]. CC on the GPU aims to use data isolation and restricted access from the CPU to prevent unauthorized access and tampering of data in computation. As Sentry is designed to operate on the GPU with minimal CPU involvement, this feature complements the goal of Sentry. We believe these recent developments in secure computation in the GPU field to be a promising prospect for an end-to-end secure AI computation on a trusted execution device.

7 Conclusion

This paper introduced Sentry, the first GPU-based ML artifacts authentication framework that which is tailored specifically for ML systems. We presented GPU implementation of cryptographic hash constructions based on Merkle tree and lattice hashing which incorporated various GPU optimization strategies. Additionally, we detailed Sentry's custom modules for model authentication and data set authentication compatible with memory fragmentation and GPUDirect solutions. Our benchmarks showed that Sentry achieves up to 269× and 180× speedup for time critical phases of model and data set authentication respectively. Using GPUs, a compute platform that is both ubiquitous and widely used for ML training and inference, we showcased the potential of embedding security into existing ML frameworks that already run on those hardware architectures. With ever growing model sizes and data set sizes for complicated machine learning systems, and stronger demand for verifying the integrity and provenance of ML artifacts, our work opens up a new domain in GPU architectural design considerations for the future of the machine learning life cycle.

References

[1] [n. d.]. DSSE: Dead Simple Signing Envelope. https://github.com/secure-systems-lab/dsse

[2] Ghada Almashaqbeh and Zahra Ghodsi. 2025. ANOFEL: supporting anonymity for privacy-preserving federated learning. *Proceedings on Privacy Enhancing Technologies* (2025).

[3] AMD. [n. d.]. HIPIFY documentation. https://rocm.docs.amd.com/projects/HIPIFY/en/latest/

[4] Emily Apsey, Phil Rogers, Michael O'Connor, and Rob Nertney. 2023. Confidential Computing on NVIDIA H100 GPUs for Secure and Trustworthy AI. https://developer.nvidia.com/blog/confidential-computing-on-h100-gpus-for-secure-and-trustworthy-ai/

[5] Jean-Philippe Aumasson, Samuel Neves, Zooko Wilcox-O'Hearn, and Christian Winnerlein. 2013. BLAKE2: simpler, smaller, fast as MD5. Cryptology ePrint Archive, Paper 2013/322. https://eprint.iacr.org/2013/322

[6] BBC. 2024. TikTok owner sacks intern for sabotaging AI project. https://www.bbc.com/news/articles/c7v62gg49zro

[7] Mihir Bellare, Oded Goldreich, and Shafi Goldwasser. 1994. Incremental cryptography: The case of hashing and signing. In *Advances in Cryptology—CRYPTO'94: 14th Annual International Cryptology Conference Santa Barbara, California, USA August 21–25, 1994 Proceedings 14*. Springer, 216–233.

[8] Mihir Bellare and Daniele Micciancio. 1997. A new paradigm for collision-free hashing: Incrementality at reduced cost. In *International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 163–192.

[9] Guido Bertoni, Joan Daemen, Michael Peeters, and Gilles Van Assche. 2015. Keccak. Cryptology ePrint Archive, Paper 2015/389. doi:10.1007/978-3-642-38348-9_19

[10] Nicholas Carlini, Matthew Jagielski, Christopher A Choquette-Choo, Daniel Paleka, Will Pearce, Hyrum Anderson, Andreas Terzis, Kurt Thomas, and Florian Tramèr. 2024. Poisoning web-scale training datasets is practical. In *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE, 407–425.

[11] Nicholas Carlini, Matthew Jagielski, and et al. 2024. Poisoning Web-Scale Training Datasets is Practical. In *2024 IEEE Symposium on Security and Privacy (SP)*. 407–425.

[12] Varun Chandrasekaran, Hengrui Jia, Anvith Thudi, Adelin Travers, Mohammad Yaghini, and Nicolas Papernot. 2021. SoK: Machine Learning Governance. arXiv:2109.10870 [cs.CR] https://arxiv.org/abs/2109.10870

[13] Ivan Bjerre Damgård. 1989. A design principle for hash functions. In *Conference on the Theory and Application of Cryptology*. Springer, 416–427.

[14] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. arXiv:1810.04805 [cs.CL] https://arxiv.org/abs/1810.04805

[15] Hugging Face. [n. d.]. Hugging Face Models. https://huggingface.co/models

[16] Facebook. [n. d.]. An open-source C++ library developed and used at Facebook. https://github.com/facebook/folly

[17] Shanglun Feng and Florian Tramèr. 2024. Privacy backdoors: stealing data with corrupted pretrained models. In *ICML'24: Proceedings of the 41st International Conference on Machine Learning*. 13326–13364.

[18] Zonghao Feng, Qipeng Xie, Qiong Luo, Yujie Chen, Haoxuan Li, Huizhong Li, and Qiang Yan. 2022. Accelerating Elliptic Curve Digital Signature Algorithms on GPUs. In *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–13. doi:10.1109/SC41404.2022.00032

[19] Andrew Gan, Setsuna Yuki, Timothy Rogers, and Zahra Ghodsi. 2025. cuOT: Accelerating Oblivious Transfer on GPUs for Privacy-preserving Computation. IEEE International Symposium on Hardware Oriented Security and Trust (HOST).

[20] Karthik Garimella, Zahra Ghodsi, Nandan Kumar Jha, Siddharth Garg, and Brandon Reagen. 2023. Characterizing and optimizing end-to-end systems for private inference. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*. 89–104.

[21] Dan Geer, Bentz Tozer, and John Speed Meyers. 2020. For Good Measure Counting Broken Links: A Quant's View of Software Supply Chain Security. *;login:* 45, 4 (2020), 83–86.

[22] Kathrin Grosse, Lukas Bieringer, Tarek R Besold, Battista Biggio, and Alexandre Alahi. 2024. When your AI becomes a target: AI security incidents and best practices. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 38. 23041–23046.

[23] Tianyu Gu, Brendan Dolan-Gavitt, and Siddharth Garg. 2019. BadNets: Identifying Vulnerabilities in the Machine Learning Model Supply Chain. arXiv:1708.06733 [cs.CR] https://arxiv.org/abs/1708.06733

[24] Tianyu Gu, Kang Liu, Brendan Dolan-Gavitt, and Siddharth Garg. 2019. Badnets: Evaluating backdooring attacks on deep neural networks. *IEEE Access* 7 (2019), 47230–47244.

[25] Hashlib [n. d.]. hashlib — Secure hashes and message digests. https://docs.python.org/3/library/hashlib.html

[26] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2015. Deep Residual Learning for Image Recognition. arXiv:1512.03385 [cs.CV] https://arxiv.org/abs/1512.03385

[27] Tzu-Ming Harry Hsu, Hang Qi, and Matthew Brown. 2019. Measuring the Effects of Non-Identical Data Distribution for Federated Visual Classification. arXiv:1909.06335 [cs.LG] https://arxiv.org/abs/1909.06335

[28] in toto. [n. d.]. in-toto Specification. https://github.com/in-toto/specification/blob/v1.0/in-toto-spec.md

[29] Intel. [n. d.]. New Instructions Supporting the Secure Hash Algorithm on Intel® Architecture Processors. https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sha-extensions.html

[30] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. 2017. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th annual international symposium on computer architecture*. 1–12.

[31] Kaggle. [n. d.]. Kaggle Models. https://www.kaggle.com/models

[32] Alex Krizhevsky. [n. d.]. Learning Multiple Layers of Features from Tiny Images. https://www.cs.toronto.edu/~kriz/learning-features-2009-TR.pdf

[33] Wai-Kong Lee, Hwa Jeong Seo, Seog Chung Seo, and Seong Oun Hwang. 2022. Efficient Implementation of AES-CTR and AES-ECB on GPUs With Applications for High-Speed FrodoKEM and Exhaustive Key Search. *IEEE Transactions on Circuits and Systems II: Express Briefs* (June 2022), 2962–2966.

[34] Kevin Lewi, Wonho Kim, Ilya Maykov, and Stephen Weis. 2019. Securing update propagation with homomorphic hashing. *Cryptology ePrint Archive* (2019).

[35] Svetlin A Manavski. 2007. CUDA compatible GPU as an efficient hardware accelerator for AES cryptography. In *2007 IEEE International Conference on Signal Processing and Communications*. IEEE, 65–68.

[36] Mihai Maruseac, Sarah Meiklejohn, Mark Lodato, and Google Open Source Security Team (GOSST). [n. d.]. Increasing transparency in AI security. https://security.googleblog.com/2023/10/increasing-transparency-in-ai-security.html Accessed: 2024-09-27.

[37] Ralph Charles Merkle. 1979. *Secrecy, authentication, and public key systems*. Stanford university.

[38] Ralph C Merkle. 1989. One way hash functions and DES. In *Conference on the Theory and Application of Cryptology*. Springer, 428–446.

[39] ModelScope. [n. d.]. ModelScope Models. https://modelscope.cn/models

[40] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, et al. 2021. Efficient large-scale language model training on gpu clusters using megatron-lm. In *Proceedings of the international conference for high performance computing, networking, storage and analysis*. 1–15.

[41] Zachary Newman, John Speed Meyers, and Santiago Torres-Arias. 2022. Sigstore: Software signing for everybody. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*. 2353–2367.

[42] NIST. [n. d.]. Secure Hash Standard (SHS). https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.180-4.pdf

[43] NVIDIA. [n. d.]. CUDA Python 12.8.0 documentation. https://nvidia.github.io/cuda-python/latest/

[44] Nvidia. [n. d.]. NVIDIA Magnum IO GPUDirect Storage Overview Guide. https://docs.nvidia.com/gpudirect-storage/overview-guide/index.html

[45] NVIDIA ADA [n. d.]. NVIDIA ADA LOVELACE PROFESSIONAL GPU ARCHITECTURE. https://images.nvidia.com/aem-dam/en-zz/Solutions/technologies/NVIDIA-ADA-GPU-PROVIZ-Architecture-Whitepaper_1.1.pdf

[46] NVIDIA CUDA [n. d.]. CUDA C++ Programming Guide. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

[47] NVIDIA cuFile API Reference [n. d.]. 3.3. cuFile Compatibility Mode. https://docs.nvidia.com/gpudirect-storage/api-reference-guide/index.html#cufile-compatibility-mode

[48] NVIDIA DALI [n. d.]. NVIDIA Data Loading Library. https://docs.nvidia.com/deeplearning/dali/user\protect\discretionary{\char\hyphenchar\font}{}{}guide/docs/index.html

[49] NVIDIA DALI GitHub [n. d.]. NVIDIA DALI GitHub. https://github.com/NVIDIA/DALI

[50] ONNX. [n. d.]. ONNX Model Zoo. https://onnx.ai/models/

[51] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. 2019. Language Models are Unsupervised Multitask Learners. *OpenAI* (2019). https://cdn.openai.com/better-language-models/language_models_are_unsupervised_multitask_learners.pdf Accessed: 2024-11-15.

[52] Beatriz M Reichert and Rafael R Obelheiro. 2024. Software supply chain security: a systematic literature review. *International Journal of Computers and Applications* 46, 10 (2024), 853–867.

[53] Phillip Rogaway and Thomas Shrimpton. 2004. Cryptographic Hash-Function Basics: Definitions, Implications, and Separations for Preimage Resistance, Second-Preimage Resistance, and Collision Resistance. https://www.iacr.org/archive/fse2004/30170373/30170373.pdf

[54] Mohamed Sabt, Mohammed Achemlal, and Abdelmadjid Bouabdallah. 2015. Trusted Execution Environment: What It is, and What It is Not. 57–64. doi:10.1109/Trustcom.2015.357

[55] SBOM FAQ [n. d.]. https://www.cisa.gov/sites/default/files/2024-07/SBOM%20FAQ%202024.pdf

[56] Sigstore. [n. d.]. Model Transparency. https://github.com/sigstore/model-transparency

[57] Karen Simonyan and Andrew Zisserman. 2015. Very Deep Convolutional Networks for Large-Scale Image Recognition. arXiv:1409.1556 [cs.CV] https://arxiv.org/abs/1409.1556

[58] SLSA [n. d.]. https://slsa.dev/

[59] SLSA In-toto [n. d.]. https://slsa.dev/blog/2023/05/in-toto-and-slsa

[60] Leonardo Solis-Vasquez, Edward Mascarenhas, and Andreas Koch. 2023. Experiences Migrating CUDA to SYCL: A Molecular Docking Case Study. In *Proceedings of the 2023 International Workshop on OpenCL*.

[61] Marcin Spoczynski, Marcela S. Melara, and Sebastian Szyller. 2025. Atlas: A Framework for ML Lifecycle Provenance & Transparency. arXiv:2502.19567 [cs.CR] https://arxiv.org/abs/2502.19567

[62] Springer 2001. *The Elliptic Curve Digital Signature Algorithm (ECDSA)*. Springer. doi:10.1007/s102070100002

[63] Sijun Tan, Brian Knott, Yuan Tian, and David J Wu. 2021. CryptGPU: Fast privacy-preserving machine learning on the GPU. In *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1021–1038.

[64] Santiago Torres-Arias, Hammad Afzali, Trishank Karthik Kuppusamy, Reza Curtmola, and Justin Cappos. 2019. in-toto: Providing farm-to-table guarantees for bits and bytes. In *28th USENIX Security Symposium (USENIX Security 19)*. 1393–1410.

[65] Stavros Volos, Kapil Vaswani, and Rodrigo Bruno. 2018. Graviton: trusted execution environments on GPUs. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*. USENIX Association.

[66] 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.

[67] Ardhi Wiratama Baskara Yudha, Jake Meyer, Shougang Yuan, Huiyang Zhou, and Yan Solihin. 2022. LITE: a low-cost practical inter-operable GPU TEE. In *Proceedings of the 36th ACM International Conference on Supercomputing*. Association for Computing Machinery.

[68] Rowan Zellers, Ari Holtzman, Yonatan Bisk, Ali Farhadi, and Yejin Choi. 2019. HellaSwag: Can a Machine Really Finish Your Sentence?. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*.

[69] Matt Zweil and The Mochimo Core Contributor Team. [n. d.]. CUDA Hashing Algorithms Collection. https://github.com/mochimodev/cuda-hashing-algos