Honeycomb: Secure and Efficient GPU Executions via Static Validation

Haohui Mai, *PrivacyCore Inc.;* Jiacheng Zhao, *SKLP, Institute of Computing Technology, CAS; Zhongguancun Laboratory; and UCAS;* Hongren Zheng, *IIIS, Tsinghua University;* Yiyang Zhao, *SKLP, Institute of Computing Technology, CAS; and UCAS;* Zibin Liu, *BUPT;* Mingyu Gao, *IIIS, Tsinghua University;* Cong Wang, *IDEA Shenzhen;* Huimin Cui, *SKLP, Institute of Computing Technology, CAS; and UCAS;* Xiaobing Feng, *SKLP, Institute of Computing Technology, CAS; Zhongguancun Laboratory; and UCAS;* Christos Kozyrakis, *PrivacyCore Inc. and Stanford*

https://www.usenix.org/conference/osdi23/presentation/mai

This paper is included in the Proceedings of the 17th USENIX Symposium on Operating Systems Design and Implementation.

July 10–12, 2023 • Boston, MA, USA

978-1-939133-34-2

Open access to the Proceedings of the 17th USENIX Symposium on Operating Systems Design and Implementation is sponsored by

Honeycomb: Secure and Efficient GPU Executions via Static Validation

Haohui Mai[♪*] Jiacheng Zhao[1,4,7†] Hongren Zheng[2] Yiyang Zhao[1,7] Zibin Liu[6]
Mingyu Gao[2] Cong Wang[5] Huimin Cui[1,7] Xiaobing Feng[1,4,7] Christos Kozyrakis[♪,3]

SKLP, Institute of Computing Technology, CAS[1] *PrivacyCore Inc.*[♪] *IIIS, Tsinghua University*[2]
Stanford[3] *Zhongguancun Laboratory*[4] *IDEA Shenzhen*[5] *BUPT*[6] *UCAS*[7]

Abstract

Graphics Processing Units (GPUs) unlock emerging use cases like large language models and autonomous driving. They process a large amount of sensitive data, where security is of critical importance. GPU Trusted Execution Environments (TEEs) generally provide security to GPU applications with modest overheads. Recent proposals for GPU TEEs are promising, but many of them require hardware changes that have a long lead time to deploy in production environments.

This paper presents Honeycomb, a software-based, secure and efficient TEE for GPU applications. The key idea of Honeycomb is to leverage static analysis to validate the security of GPU applications at load time. Co-designing with the CPU TEE, as well as adding OS and driver support, Honeycomb is able to remove both the OS and the driver from the trusted computing base (TCB). Validation also ensures that all applications inside the system are secure, enabling a concise and secure approach to exchange data in plaintext via shared device memory on the GPU.

We have prototyped Honeycomb targeting the AMD RX6900XT GPU. Honeycomb is evaluated on five representative benchmarks and 23 applications in total, covering workloads of high performance computing, deep learning, and image processing. The results show that Honeycomb is both *practical* and *efficient* to secure real-world GPU applications. Validating applications to run on Honeycomb requires modest developer efforts. The TCB is 18× smaller than the Linux-based systems. Secure inter-process communication is up to 529× faster. Moreover, running large language model workloads like BERT and NanoGPT has ∼2% overheads.

1 Introduction

Innovations in hardware accelerators and deep neural networks continue to enable personalized experiences for our physical and digital presences, reshaping areas ranging from smart homes [34], virtual reality [85], to personalized cancer medicines [22]. Offering such intimate experiences heavily relies on large amounts of valuable and sensitive user data, which requires high levels of security and privacy support on hardware accelerators such as GPUs.

Trusted Execution Environments (TEEs) [4] encapsulate applications into enclaves to enhance security. TEEs enforce strong isolation among enclaves and the untrusted host environments, so that applications inside the enclaves can process plaintext data securely at native speed. For each enclave, all traffic that crosses its boundaries is encrypted to maintain confidentiality and integrity. Recent prototypes [28,44,45,83] realize GPU TEEs with modest overheads, via serializing secure access to the GPU [28], augmenting the GPU hardware [83], customizing the I/O bus [44], or leveraging the sharing capabilities in device drivers [45].

This paper explores an alternative approach – using static analysis to *validate* that mutually distrusted GPU applications are confined to their enclaves. Intuitively, a validator inspects the binary code of GPU kernel functions (GPU kernels for short) to show that all possible execution traces maintain the confidentiality and integrity of the system, therefore these applications can safely share the GPU. This approach offers three benefits. First, it can complement the hardware limitations of existing GPUs. For example, low-cost GPUs such as the VC4 used by Raspberry Pi allow arbitrary writes to memory due to the lack of corresponding MMUs [16]. A validator can detect insecure behaviors and thwart the attacks by running standard static analysis such as def-use analysis and range checks on the GPU kernels. Moreover, advanced static analysis [59] might mitigate new attacks [19,21] much faster compared to deploying new hardware supports in production.

The second benefit is that it allows more efficient implementations of current GPU TEEs. Shifting the runtime checks to load time removes them from the critical paths. Moreover, validating that applications that always have disjoint contexts might save the TEE implementation from flushing architectural contexts, including TLBs and buffer queues during every context switch [28], thus improving performances.

*Haohui Mai is also affiliated with Hengmuxing Technologies.
†Jiacheng Zhao is the corresponding author.

Finally, validating every application provides a system-wide security invariant asserting that all applications are "good citizens". The security invariant enables secure and efficient communication among enclaves. Real-world applications such as autonomous driving [89] and video analytics [66, 70] process data in multiple-stage pipelines. Separating each stage of the pipeline into different enclaves and connecting them using Inter-Process Communication (IPC) not only increases modularity and robustness, but also enables assembling the pipelines using mutually distrusted components from multiple vendors [63, 74]. Current GPU TEEs focus on strengthening isolation, for example, enforcing exclusive ownerships of GPU device memory [83]. Therefore two mutually distrusted enclaves need to tunnel the data through an encrypted shared buffer on the host memory for IPC. The overheads are prohibitive for production applications. For example, the GPUs on an autonomous vehicle process up to 50 GB/s of uncompressed video streams to make timely driving decisions [69]. Copying 50 GB/s of data to the host already takes up 30∼40% of the total memory bandwidth of a commodity, high-end AMD Zen3 server, let alone the overheads of encrypting/decrypting the data. The capability of exchanging plaintext data directly in GPU reduces the overheads drastically, thus enabling real-world applications to migrate towards a more modular and robust architecture.

This paper presents the design and implementation of Honeycomb, a software-based, secure and efficient TEE for GPU applications. Honeycomb runs multiple mutually distrusted applications on the same GPU, and facilitates efficient and secure data exchange between applications. It supports common GPU workloads from simulations of molecular dynamics to training and inference of neural networks. All these capabilities of Honeycomb are built upon the idea of using static analysis to confine the behaviors of GPU applications.

Honeycomb faces three challenges to realize the three benefits and to provide a complete, real-world solution for GPU TEEs. First, it must balance the trade-offs between the capabilities and the complexities of the validator. A validator equipped with theorem provers gains their power, but then Honeycomb must include the theorem provers in the TCB, which is complex (e.g., Z3 4.12.2 has ∼525 K lines of code) and occasionally error-prone [77]. On the other hand, a naïve validator might be insufficient to validate common security checks at load time, requiring inserting extra runtime checks that sit squarely on the performance critical paths.

Second, Honeycomb must minimize the end-to-end TCB to provide high confidence in security. The software/hardware stack of GPU applications is quite complex. For example, the compiler toolchain and the driver for the AMD RX6900XT GPU each consist of two million lines of code. Defects and vulnerabilities in these components are inevitable [23, 25, 26], but they should not compromise the security of Honeycomb.

Finally, Honeycomb must provide system-level support for secure and efficient IPC. The aforementioned plaintext IPC among GPU enclaves can only be securely implemented if the data copies are cautiously initialized by the Honeycomb system and from/to strictly protected memory regions.

Honeycomb addresses the above challenges with three key techniques. First, the validator of Honeycomb performs static analysis of GPU kernels directly on binaries. It decodes the instructions of the GPU kernels to reconstruct the control and data flows. It models the memory access patterns using scalar evolution [6] and polyhedral models [14]. Our evaluation shows that the approach is effective to validate that the majority of memory accesses in GPU kernels are safe, because real-world GPU kernels tend to be well-optimized, having highly regular control flow structures and memory access patterns. The few remaining cases can be handled by inserting runtime checks, whose latencies are also well tolerated by the GPU memory hierarchy (§5).

Second, Honeycomb leverages hardware isolation mechanisms, and uses security monitors [54, 79, 90] to validate interactions in the system, so that it can minimize the trust on the software/hardware stack. Honeycomb launches applications inside CPU TEEs powered by AMD SEV-SNP [4]. The validator directly parses the GPU binaries to remove the compiler toolchain from the TCB. To remove both the user-space and kernel-space GPU drivers from the TCB, Honeycomb uses two security monitors to intercept and regulate all traffic between the applications and the GPU: (1) a Secure VM Service Module (SVSM) [4] running inside the application enclave, which enforces security policies at the application level (e.g., the application only launches validated kernels), and (2) a security monitor running inside a sandboxing hypervisor of the GPU, which regulates the behaviors of the GPU driver (e.g., the driver should never map a private memory page into two applications). Additionally, Honeycomb secures the data transfer between the CPU and the GPU to protect the confidentiality and integrity of the data (§6).

For the final challenge, Honeycomb reserves dedicated regions of the virtual address space for secure IPC to exchange plaintext data. Particularly, Honeycomb divides the virtual address space of each application into four regions: protected, read-only, read-write, and private. The validator ensures that application GPU kernels can only modify the private region. Putting the metadata and the receiving buffers into the protected and read-only regions prevents user applications from tampering with the IPC, reducing IPC in Honeycomb to copying plaintext data within the device memory (§7).

We have ported five representative benchmark suites, including the SpecACCEL 1.2 benchmark suites [76], inference applications of the ResNet18 neural network model [37] and the BERT language model [29], an application that trains GPT language models [48], and an image processing application that performs Canny edge detection [20], i.e., 23 applications in total. We have evaluated them on a server equipped with two AMD EPYC 7443 24-core processors and an AMD RX6900XT GPU. The results are promising. The TCB is

18× smaller than the Linux-based systems. A concise validator is sufficient to statically verify the security of large parts of GPU applications. Validating inference workloads on neural networks like ResNet18 and BERT requires adding zero runtime checks into the GPU kernels. Large language model workloads like BERT and NanoGPT have $\sim 2\%$ runtime overheads. IPC in Honeycomb is up to 529× faster than exchanging data using an encrypted, shared buffer on the host.

This paper makes the following contributions:

- The use of static analysis on GPU kernels to confine the behaviors of GPU applications to improve security. Our evaluations on five representative benchmark suites show that the analysis is both practical and effective to determine whether real-world GPU kernels are safe at load time with minimal additional runtime checks.
- The design and the implementation of a lightweight, end-to-end secure execution environment for GPU applications based on static validation.
- An IPC primitive that enables secure and efficient communications between GPU applications. The co-design of static analysis and OS support leads to a highly concise implementation.

2 Background

To understand the design of Honeycomb, it is important to first review the architectures and the programming interfaces of GPUs, as well as the basic concepts of polyhedral analysis [14, 35] used in this paper.

Architectures and programming interfaces of commodity GPUs. Modern GPUs offer the single instruction, multiple thread (SIMT) programming model to the applications. To run a workload, an application submits a launch request to the command queue of the GPU. The request specifies the binary function (i.e., GPU kernel), its arguments, the number of threads, and optionally, the size of a user-controllable, on-die high-speed scratchpad (i.e., shared memory) to perform the workload. Threads are organized into grids and blocks uniformly. Each grid consists of the same number of blocks, and each block consists of the same number of threads. Each thread within the same block has its own vector registers but shares access to the shared memory. The programming model provides a conceptual view where each thread executes the same instruction based on the values of its own registers. To achieve parallelization, each thread loads the inputs into its own registers and computes the outputs in parallel. Figure 1 presents an example of filling a region of memory to a specific value under the SIMT model.

The hardware architecture of GPUs closely matches the SIMT model above. A typical GPU consists of thousands of processing elements (PE) that are grouped into a three-level hierarchy. The lowest level is called a warp, consisting of 32 or 64 logical PEs executed in lock-step. The micro-architecture (e.g., AMD GCN) might introduce parallel scalar units to perform uniform computation within a warp, or pipeline the computations on physical PEs to hide execution latency. Warps are further grouped into Compute Units (CU). A CU consists of a pool of vector registers and shared memory. Finally, a single GPU packages multiple CUs on the same die.

The hardware scheduler multiplexes the hardware resources across applications. The minimal scheduling unit is a warp. It always schedules all warps of a block within the same CU, therefore all threads within a block divide the vector register pool and share the same allocated shared memory inside the CU. The scheduler continuously schedules all the blocks and grids until the execution is completed.

The GPU driver creates a virtual address space for each GPU application. It allocates buffers for arguments and command queues out of the Graphics Translation Table (GTT) memory from the host. The buffers are mapped into the virtual address space on the GPU, from which the GPU kernels read the arguments and the layouts of grids and blocks directly.

AMD SEV-SNP. AMD SEV-SNP [4] (Secure Encrypted Virtualization-Secure Nested Paging) offers enhanced security features at the hardware level for Virtual Machines (VMs) running on an untrusted cloud system hypervisor. Similar to other TEEs, SEV-SNP supports remote attestation as well as both data confidentiality and integrity guarantees for the application VMs against untrusted host hypervisors. A dedicated hardware engine in the memory controller encrypts data before sending them to the off-chip main memory. SEV-SNP also tracks the ownership of each physical page with a Reverse Map Table (RMP) so that only the owner can write to a memory region. It further validates the page mapping to prevent malicious remapping of a single page to multiple owners. In such ways, it is able to alleviate typical data corruption, replay, memory aliasing, and memory remapping attacks.

In addition, SEV-SNP enables tagging each physical page with Virtual Memory Privilege Levels (VMPLs). It is similar to Ring 0-3 in the x86 architecture but for TEE VMs. One use case of VMPL is to implement Secure VM Service Module (SVSM). SVSM runs at VMPL0 and the guest operating system runs at VMPL1. SVSM can intercept syscalls and memory operations and serve as a security monitor.

Polyhedral model. The polyhedral model has been widely used in automatic parallelization and optimization of GPU programs [8, 14, 92]. Conceptually it represents each memory access as an affine expression (i.e. a linear combination) over an ordered set of loop variables. Analyzing the effects of memory access, such as aliasing and ranges, reduces to solving inequalities of integer variables. The polyhedral model works well with GPU kernels because they implicitly loop over the grids and the blocks, and performant GPU kernels have regular memory access patterns.

More concretely, an iteration vector $I = (i_0, i_1, \ldots, i_n) \in \mathcal{D}^{\mathtt{s}}$ records the values of loop variables $i_0, \ldots, i_n$ for an instruction `s`. The domain $\mathcal{D}^{\mathtt{s}}$ is called the iteration domain. Note that the iterator vector usually includes the grid index (`gid`) and the local thread index (`lid`) for instructions in GPU kernels. An access function $\mathcal{A}^{\mathtt{s}}$ (w.r.t. instruction `s`) takes an iterator vector as input and outputs the actual memory address.

Note that when $\mathcal{A}^{\mathtt{s}}$ is an affine function and $\mathcal{D}^{\mathtt{s}}$ is an affine space, all loops in I have fixed steps. For simplicity, we denote an access function as a vector with each element representing the coefficients of the corresponding dimension of the iteration vector. The dot product of the access function and the iteration vector is the actual memory address. We also introduce an extra dimension which always has the value 1 at the end of the iteration vector so that the access function can represent constant offsets in a uniform way.

Figure 2 shows the access functions of the kernel in Figure 1, a kernel filling a range of memory with a value. Affine operations on the values directly translate to affine operations on the vector forms of the corresponding access functions (e.g., $\mathcal{A}^5 = \mathtt{dim} \cdot \mathcal{A}^3 + \mathcal{A}^4$), provided that `dim` is a constant throughout the analysis. The GPU kernel actually loads `dim` from the memory, however. In this case, security invariants in Honeycomb ensure that the value `dim` remains constant throughout the executions so that the analysis remains valid.

3 Threat model

In this paper, we adopt a similar threat model to previous studies on secure execution environments for GPUs [44, 45, 83]. The adversary controls the entire software stack, including the compiler toolchains, the operating system, the hypervisor, and the device drivers. It also has physical access to the server hardware and may sniff the PCIe traffic. We assume that the host machine CPU features TEE capabilities such as AMD SEV-SNP or Intel TDX [43], and the GPU features a hardware random number generator or performance counters to collect entropy for cryptographic uses. We also assume that users have the specifications of the server hardware and how it is connected, such as which PCIe slot that the GPU is plugged in. Finally we assume that Honeycomb is able to establish a trusted MMIO path with the GPU. Our prototype uses AMD SEV-TIO [1] to establish it, but such a path can also be realized using other secure I/O buses [44, 65], or alternatively, equipping the server with tamper detection mechanisms [75] and establishing a trusted I/O path to the GPU using a hypervisor [94]. We defer the details to §8.

The adversary can launch applications in Honeycomb, alter the results of the compiler toolchains, and tamper with the physical memory of the server. Additionally, the adversary can tamper with the DMA buffers. However, we trust the device memory of the GPU, since modern GPUs usually integrate the device memory using 2.5D/3D silicon interposers inside the same package. We assume that the adversary cannot observe or corrupt the data stored in it [83]. Supporting integrated GPUs is out of the scope of this paper.

Similar to previous GPU TEEs [44, 83], side-channel attacks [17, 40, 82, 86] on trusted hardware are out of the scope of this paper. Honeycomb relies on the rich set of orthogonal work to alleviate these problems [9, 80]. Availability and denial-of-service attacks are also out of scope.

Under this threat model, Honeycomb should ensure confidentiality and integrity for multiple mutually distrusted applications running on the same GPU. The adversary cannot tamper with the code, the data and the control flows of both the CPU and GPU parts of the applications.

4 Overview

Figure 1 describes the overall architecture of Honeycomb. Honeycomb offers unified TEEs that cover both the CPU and GPU parts of the application. Honeycomb starts an application inside an AMD SEV-SNP TEE VM. It first starts the Secure VM Service Module (SVSM) at VMPL0. The SVSM bootstraps the BIOS, the guest Linux kernel, and finally the user-space application at VMPL1. SVSM regulates all interactions between the applications and the GPU. Recall that in CPU TEEs data are stored as plaintext within the CPU package. They are only encrypted when leaving for the off-chip main memory. In Honeycomb data on the device memory are stored decrypted, and the SVSM encrypts them when they are sent to the host. The path of reading data is similar.

The application requests GTT memory from Honeycomb to interact with the GPU. A piece of GTT memory can serve as a staging buffer for memory copies, which is mapped into the user-level address space, or serve as backing buffers for command queues, which are only accessible by the SVSM. In both cases the SVSM inspects the access to regulate secure memory transfers between the GPU and the applications [83], and launches validated GPU kernels with proper parameters. Note that although the current implementation of Honeycomb is based on AMD SEV-SNP, our design is applicable to other VM TEEs such as Intel TDX.

Honeycomb isolates the GPU inside a sandbox VM. The security monitor (SM) inside the sandbox is a hypervisor running below the Linux kernel. The SM regulates all interactions between the driver and the GPU. It ensures that the GPU follows the expected initialization sequences, and keeps track of the ownerships of the device memory pages to prevent accidental sharing of device memory among applications.

To execute GPU kernels, an application first loads the GPU binary that contains the GPU kernels into the device memory. The validator in Honeycomb takes both the binary code of a GPU kernel and the accompanying preconditions as inputs. It validates that each memory instruction in the GPU kernel can only access certain regions of the virtual address space. Note that the actual target addresses sometimes cannot be determined until the application executes the kernel with

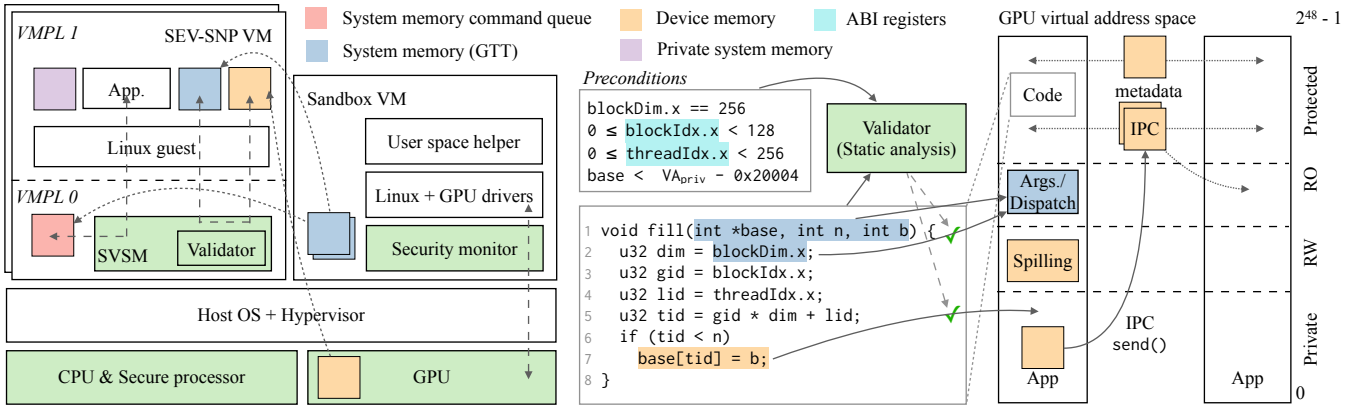

Figure 1: The overall architecture of Honeycomb. Left: Application VM and sandbox VM in their respective TEEs. Middle: Pre-conditions and source code of the GPU application kernel. Right: Layout of the virtual address spaces for two GPU applications. Long dashed arrows represent intercepted and validated requests by the validator (at load time) and the security monitors (at runtime). Dotted arrows represent physical memory page mappings. VA_{priv} is the topmost virtual address of the private region. Green boxes are components of the TCB.

$$
\begin{aligned}
\mathcal{D} &= \{(\text{gid},\text{lid}) | 0 \le \text{gid} < \texttt{gridDim}; 0 \le \text{lid} < \texttt{blockDim}\} \\
\mathcal{A}^3 &= (1,0,0) \\
\mathcal{A}^4 &= (0,1,0) \\
\mathcal{A}^5 &= \texttt{dim} \cdot \mathcal{A}^3 + \mathcal{A}^4 = (\texttt{dim},1,0) \\
\mathcal{A}^7 &= \mathcal{A}^5 + (0,0,\texttt{base}) = (\texttt{dim},1,\texttt{base})
\end{aligned}
$$

Figure 2: The iteration domain and the access functions (in the vector form) of the GPU kernel in Figure 1. The superscript denotes the corresponding statement. The parameter spaces of all access functions are $(\text{gid},\text{lid},1)$. `gridDim` and `blockDim` describe the total number of grids and the number of threads in a block.

the concrete values of the arguments (e.g., `base` in Figure 1). Therefore we introduce preconditions, which specify the constraints on the arguments so that the validator can analyze the bounds statically. Honeycomb checks the preconditions at runtime to ensure the attacker cannot subvert the analysis.

The validator decodes the instructions of the GPU kernel to reconstruct its control and data flows. It represents the target address of each memory instruction as a symbolic expression using scalar evolution and polyhedral models. It plugs in the preconditions to reason about the bounds of the target address, and ensures that the address stays within specified regions. The analysis is sound, meaning that once an access is proven, it is safe for all possible executions. For undecided cases like an indirect memory access `a[b[i]]`, Honeycomb requires the developer to annotate and add runtime checks to pass the validation. Our evaluation on real-world benchmark suites shows that the overheads of both development and runtime performance are modest – common production GPU kernels like matrix multiplications tend to have regular memory access patterns. The analysis is sufficient to capture the patterns, thus requiring few to none annotations.

The validator enforces access control that effectively divides the virtual address space of a GPU application into four regions: protected, read-only (RO), read-write (RW), and private, each of which has different access policies. For example, the application is prohibited to modify the RO region, but has full access to the private region. Honeycomb places the binary code and the arguments in the RO region so that a malicious kernel cannot modify the code on the fly after passing the validation. Furthermore, Honeycomb implements secure IPC through mapping the buffers into different regions. Honeycomb maps the IPC buffers into the sender's protected and receiver's RO region. The sender calls the trusted `send()` endpoint to copy the plaintext data to the IPC buffer, where both confidentiality and integrity are preserved.

5 Validator

The validator in Honeycomb checks the binary code for each GPU kernel of the application conforms with the following security invariants:

- *No dangling accesses*. A GPU kernel must never read uninitialized values from hardware registers.
- *All memory accesses reside in their regions*. All memory accesses to the memory regions conform with their access policies respectively.
- *Control flow integrity*. The execution must start at the designated entry point of the GPU kernel. The kernel can only transfer its control to the entry points of its basic blocks.

Checking uninitialized uses of values. The validator starts out parsing the binary code of the GPU kernel and building

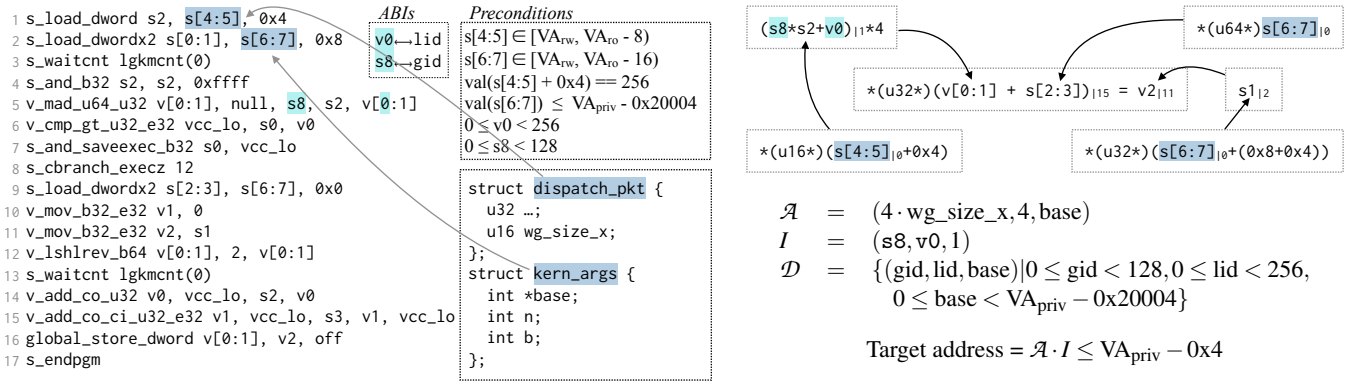

Figure 3: Workflow of validating the binary code generated from Figure 1. Left: the assembly, preconditions and the ABIs. Top right: the value chain of the symbolic expression that represents the target address of Line 16. Subscripts represent the line number on the left at which the value should be considered. Bottom right: the access function $\mathcal{A}$, the iteration vector I and the iteration domain $\mathcal{D}$.

the Static Single-Assignment (SSA) representation and the Control Flow Graph (CFG) for each kernel function. The validator checks dangling accesses by inspecting whether the SSA representation of the kernel function is valid.

Checking memory accesses. Figure 3 presents the overall workflow when validating the program described in Figure 1. For each memory instruction, the validator constructs a symbolic expression and derives the access function $\mathcal{A}$ to represent the target address of each memory instruction. The algorithm combines scalar evolution analysis and polyhedral models, and is flow-sensitive and path-insensitive.

The validator further derives the iteration vector I and the iteration domain $\mathcal{D}$ from the application binary interface (ABI) and the preconditions of the GPU kernel. Recall that the dot product $\mathcal{A} \cdot I$ computes the value of the target address. It is sufficient to plug in $\mathcal{D}$ to compute the range of the target address and to verify whether the memory access is inbound.

A closer look at Figure 3 shows that the validator must address practical complexities when analyzing the binary code. For example, the compiler promotes the load of `dispatch_pkt.wg_size_x` into a 32-bit load instruction (Lines 1 and 4). It also lowers a 64-bit addition into two instructions (Lines 14-15). The validator heuristically rediscovers their semantics when constructing the symbolic expressions. Additionally, the validator matches sequences of instructions to rediscover semantics of divisions, modulus, and min/max operators.

Another example is that conventional polyhedral models require all multipliers to be constants. The value of `s2` comes from a load instruction (Lines 1 and 4), breaking the subsequent analysis when constructing a polyhedral representation of the global ID (Line 5). The validator recognizes that the instruction is loading from the RO region and it is safe to treat it as a constant in the analysis. Such relaxation is essential to derive $\mathcal{A}$ and eventually to validate that Line 16 is safe.

Aggressive compiler optimizations can create additional burdens for analysis. For example, the definition and the usage of a value could be scattered in two basic blocks separated by other basic blocks in the CFG. They are guarded by the same condition so the program is valid at runtime but a path-insensitive algorithm fails to connect them. More powerful analysis or language-level support [30,41,60] will address the issue but we intentionally limit the capabilities of the validator to bound the size of TCB. Honeycomb requires the developer to alter the GPU kernel to pass the validation. Additionally, the validator requires the developer to add runtime checks for indirect memory accesses like `a[b[i]]` since it does not fully track the memory access of the heap.

We found that the simple algorithm is effective against commonly used production kernels such as matrix multiplications or element-wise transformations as the analysis perfectly captures the regular and predictable memory access patterns commonly seen in most GPU kernels.

Enforcing control flow integrity. It is relatively straightforward to decode GPU kernels since modern GPUs have RISC-style instruction sets. The validator simply validates that all branches jump to valid instructions. The validator does not support indirect branches. Although based on our experience they are rarely used in real-world GPU kernels, the developer can turn indirect branches to a series of branches that have explicit targets. The validator does not support self-modifying code to ensure the integrity of the analysis.

6 Security monitors

There are two types of security monitors in Honeycomb to regulate the interactions with the GPU. In Honeycomb every application runs inside its own TEE VM. The SVSM regulates the interactions between the application and the GPU. The security monitor (SM) in the sandbox VM regulates the interactions between the driver and the GPU. The SM also keeps track of the ownership of memory pages to prevent ac-

cidental sharing between applications. Together they are able to remove the OS kernel and the GPU driver from the TCB. Similar to existing GPU TEEs [83], Honeycomb implements the following functionalities.

Initialization. Honeycomb enforces the untrusted GPU driver to follow a correct sequence to initialize the GPU. However, to our best knowledge, no public specifications are available for our target device, the AMD RX6900XT GPU. We therefore collect the trace of an initialization sequence on the baseline platform and use it as the ground truth. We further inspect the source code of the driver to build state machines to model the initialization sequence. The SM intercepts all MMIO traffic to ensure that the GPU driver follows the transitions of the state machines. The SM directly passes the firmware to the GPU since the hardware will validate its integrity with cryptographic signatures.

Despite the fact that there is no specification, we were able to find five bugs in the AMDGPU driver that violate security. More specifically, there are two instances where the parameters of the hardware queues are initialized with incorrect values, two instances where the queues are prematurely enabled before all parameters are set, and one instance of out-of-bound access on the hardware buffer. All five bugs are confirmed by upstream developers, and their corresponding fixes have been deployed since Linux 5.19.

Launching GPU kernels. Applications call the same user-space APIs (e.g., the HIP APIs [2]) to launch GPU kernels on Honeycomb. First, applications call `hipModuleLoadData()` to load GPU binaries. The implementation of the API traps into the SVSM, where the SVSM validates the kernels, then copies the kernels into the protected region and records their preconditions given that they have passed the validations.

Applications call `hipLaunchKernel()` to launch a GPU kernel. Similarly, its implementation traps into the SVSM, where the SVSM confirms the preconditions are valid with respect to the actual arguments. It then updates the command queue to enqueue the launch if preconditions are satisfied.

Isolating address spaces. On the CPU side, Honeycomb leverages existing mechanisms in SEV-SNP TEE to enforce isolation between different applications. SEV-SNP ensures the integrity of VM data and protects against various vulnerabilities, including replay and remapping attacks (§2).

On the GPU side, the SM intercepts all traffic between the driver and the GPU to maintain a RMP table similar to Graviton [83] to track the ownership of the pages. The Linux driver allocates page tables inside the device memory and updates them through MMIO requests. The SM intercepts these requests and updates the RMP table. Additionally, the SM prevents applications from mapping the page tables into their address spaces to subvert the isolation.

Securing data transfers. Honeycomb implements secure data communication channels between the GPU and the host CPU, and coordinates all data transfers into and out of the GPU device memory. All transfers between the host and the device memory must be done via a special trusted kernel in Honeycomb, with all transferred data encrypted and authenticated under an ephemeral encryption key. Honeycomb disallows the applications from mapping the host memory into their address spaces or directly creating DMA queues.

One practical issue is how to bootstrap and maintain the secure channel. Honeycomb uses the `s_memrealtime` instruction to get the value of the real-time counter on the AMD RX6900XT GPU. Honeycomb launches a kernel to perform reads, invalidating caches to generate entropy and extract them. The entropy is used to establish a shared security key using Diffie-Hellman key exchange [31]. Honeycomb stores the entropy in the protected region to prevent user applications from accessing it.

7 Secure and efficient IPC

Honeycomb enables two enclaves to securely exchange plaintext data within the device memory. To make an IPC, Honeycomb maps the IPC buffer to the sender's protected region and the receiver's RO region. The sender calls `send()` to initiate the IPC. `send()` is a trusted endpoint that simply copies the data into the protected region and updates the indices of the IPC buffers. The GPU kernels on the receiver side can read the RO region directly but need to update the indices via `recv()` provided by Honeycomb. The scheme is secure because no GPU kernels from the user applications can access the protected region nor write to the RO region.

For simplicity, the current prototype of Honeycomb maps all IPC buffers to the protected regions consistently across all applications so that it is possible to identify the endpoints using only the virtual addresses. Adding finer-grained access control through capabilities [33] is relatively straightforward.

To summarize, the ultimate simplicity comes from the guarantee that none of the user-provided GPU kernels inside Honeycomb is able to tamper with the data in the protected, RO and RW regions, making exchanging data between two enclave applications in Honeycomb as simple as copying a piece of memory.

8 Discussion

Establishing a trusted I/O path to the GPU. When the server does not have AMD SEV-TIO or other secure I/O buses, Honeycomb can leverage prior work [94] to establish a trusted I/O path to the GPU. On the high level, Honeycomb acquires exclusive control of the I/O paths at the beginning of the boot-up process, before any untrusted components can access the GPU. Particularly, the server first boots into the SM where the whole boot-up process is validated and attested via SecureBoot [55]. Second, the SM enumerates the PCIe buses to discover the MMIO regions of the GPU and all of its

upstream PCIe switches. Then it initializes the IOMMU to protect the MMIO regions from any unauthorized accesses by the hypervisor and other devices' DMA buffers. It further programs the PCIe Access Control Services (ACS) registers [5] to stop unauthorized peer-to-peer PCIe transactions. After these steps Honeycomb can continue the normal boot-up process. By correctly configuring the IOMMU and by disabling peer-to-peer PCIe accesses, the above initialization process is sufficient to isolate the MMIO regions of the GPU from the untrusted software components and other I/O devices inside the server [94]. Honeycomb relies on additional tamper detection mechanisms to detect and mitigate physical attacks.

In the alternative setup above, the TCB must include all components used in the boot-up process, such as the UEFI BIOS. The SM and the untrusted hypervisor must be adopted to support nested virtualization [11]. Legacy systems without tamper detection mechanisms have a weaker threat model as they are unable to defend against physical attacks.

Remote attestations. The user application needs to attest both its own SEV-SNP VM and the sandbox VM to validate the security of the execution environment. It follows the standard procedures to attest its own VM. The attestation also guarantees a valid execution context of the GPU since the SVSM is part of the attestation, and it regulates the GPU execution context. Note that the SM is part of the TCB. The SM maintains a public-private key pair for the attestation of the sandbox VM. The user application authenticates the SM using the key pair to validate the security of the sandbox VM.

9 Security analysis

Attacking the software stack. An attacker might try to launch a malicious GPU kernel by subverting the validation in Honeycomb. Some possible attacks include invalidating the preconditions with invalid arguments, subverting the data flows via manipulating the values of spilled registers, and modifying the code or the target addresses of a branch to hijack the control flows [18]. Recall that SEV-SNP ensures that the user application cannot tamper with the SVSM. The SVSM reevaluates the preconditions with the arguments before executing the GPU kernels to defend against the first attack. Honeycomb protects the code and the spilling regions in the RO and RW regions to defend against the second and third attacks. Particularly, the validator analyzes each global memory access in the GPU kernel to ensure that it cannot tamper with the reserved regions, maintaining the integrity of the validation.

An attacker might tamper with the system software stack, including the GPU driver, the host operating system, and the hypervisor, to subvert the security of Honeycomb. To gain access to the plaintext information residing in the device memory, they might execute code to craft malicious MMIO requests, to initiate unauthorized DMA requests, or to map the device memory from other applications to different address spaces. This is ineffective because the SM in Honeycomb regulates all MMIO and DMA requests from the system software stack. By intercepting the MMIO requests, it enforces isolation on address spaces and prevents accidental sharing. It also enforces that data communication with the external world is all encrypted and authenticated.

The attacker might alter the GPU firmware or divert from the designated bootup sequences in order to control the GPU. This is ineffective because the GPU hardware verifies the integrity of the firmware [55], and the SM in Honeycomb validates the bootup sequences during GPU initialization.

Attacking the hardware stack. An attacker might interpose the host memory to try to alter the trusted components like the SVSM in Honeycomb. This is ineffective because SEV-SNP includes attestation procedures to verify the integrity of the trusted components. SEV-SNP also incorporates memory encryption and integrity to defend against the attack.

An attacker might interpose on the PCIe fabrics to insert MMIO or DMA requests, or tamper with existing requests to access the plaintext information residing in the device memory. Alternatively, they might map the MMIO regions of the GPU to another I/O device or initiate peer-to-peer PCIe transactions to interact with the GPU. Both types of attacks are ineffective when the GPU is attached to a secure I/O bus [1, 44, 65]. When using the alternative initialization process described in §8 to establish a trusted I/O path, Honeycomb detects and stops the first type of attacks using tamper detection mechanisms [75]. To defend against the second type of attacks, Honeycomb programs the IOMMU and PCIe ACS registers to acquire exclusive control on the MMIO regions of the GPU before starting any untrusted components. Additionally an attacker might write to the I/O ports that map to the registers in the PCIe configuration space, in the hope of relocating the MMIO regions of the GPU. Honeycomb is able to identify and stop potential attacks as the hardware topology uniquely determines the mappings [94]. An attacker might also initiate peer-to-peer PCIe transactions between an I/O device and the GPU bypassing the IOMMU. Honeycomb stops the attacks because it programs the PCIe ACS registers to prevent unauthorized peer-to-peer PCIe transactions.

Our threat model assumes that an attacker cannot snoop or tamper with the device memory of the discrete GPU. The attacker can also try to perform the row hammer attack [51], which can be mitigated by orthogonal research [7, 67, 87].

Side-channel attacks. An attacker might try to exploit various timing and power side channels. Defending them is out of the scope of this paper and can leverage orthogonal work [9, 80].

10 Implementation

We have implemented Honeycomb on top of Rust 1.64.0 nightly with about 32,000 lines of code. The current prototype supports the x64 architecture and the AMD RX6900XT GPU.

We use the AES256-GCM [32] to encrypt and decrypt traffic between the host and the GPU.

The validator understands the AMDGPU ELF binary format and disassembles the machine code of the GPU kernels of the AMD RDNA2 ISA. The structures of scalar evolution analysis and polyhedral representations closely resemble the corresponding parts in LLVM [53].

We have implemented both the SVSM and the SM in Rust. The SM is implemented as a Type I hypervisor. We have implemented the user-space runtime, including the corresponding bindings of HIP and OpenCL in C++, in around 8,500 lines of code. The user-space runtime is outside the TCB.

11 Evaluation

The evaluation of Honeycomb tries to answer the following questions both qualitatively and quantitatively:

- Does static validation in Honeycomb improve security?
- Is Honeycomb practical for real-world applications?
- Where do the overheads in Honeycomb come from?
- How efficient is the IPC in Honeycomb?
- How much effort is required to adopt Honeycomb for new applications?

11.1 Experiment setup

We evaluate Honeycomb on a server equipped with two 24-core 2.85 GHz AMD EPYC 7443 CPUs, 128 GB DDR4 memory, and a 480 GB SAMSUNG PM893 SSD. The server has an AMD RX6900XT GPU that has 16 GB of device memory. It connects to a gigabit Ethernet with the Broadcom BCM5720 Ethernet adapter. The machine runs a patched Linux 5.15.0 kernel to support SEV-SNP VMs. Both the sandbox and the application VM runs Linux 5.17.0 on top of QEMU 7.1.0. We have not yet enabled SEV-SNP for the sandbox VM due to complications of passing the AMD RX6900XT GPU directly into the VM. We use the ROCm 5.4.0 [3] GPU driver when running the baseline experiments. We pin all applications to the first CPU socket where the GPU is attached.

11.2 TCB

Honeycomb provides a secure and efficient execution environment for GPU applications. To quantitatively evaluate our efforts, we count the lines of code (LOC) in the TCB of both Honeycomb and the Linux platform using SCC [15]. The current prototype of Honeycomb only supports a limited set of hardware, thus we only count the lines of code for the x64 platform and the essential parts of the driver for AMD RX6900XT. Figure 4 presents the counts of LOC for the TCB of both Honeycomb and the Linux platform.

Honeycomb provides security guarantees with respect to the threat model in Section 3 with an order of magnitude smaller TCB compared to the normal Linux platform. The security of a GPU application running on Linux relies on the correctness of both the kernel space and the user space (ROCm) of the GPU driver. The result of the smaller TCB is consistent with other systems that adopt the design of security monitors [79, 90]. The SM and the validator in Honeycomb separate the concerns of enforcing security from implementing the required functionalities, removing the heavy-lifting portions (e.g., Linux) of the system out of the TCB.

System	LOC
Honeycomb	82,738
SVSM	9,839
SM+Sandbox VM	9,376
Validator	12,299
Rust runtime	50,864
Linux 5.17	$\sim$ 1,503,519
Core functionalities for x86	844,993
AMDGPU driver for AMD 6900XT	607,689
Kernel libraries (DRM & TTM)	50,837
ROCm 5.4.0	397,151
HIP Library	188,995
ROCR Runtime	73,241
ROCm Common Runtime	62,173
ROCR Thunk interface	72,742

Figure 4: Estimated LOC for TCBs of Honeycomb and Linux. It also shows the LOC of some major components in the TCB.

11.3 End-to-end performance

We choose five representative benchmark suites to study how Honeycomb performs on real-world workloads:

SpecACCEL. SpecACCEL is a performance test suite that represents high-performance computing applications like simulations of computational fluid dynamics and molecular dynamics. We evaluate all 19 OpenCL applications in the SpecACCEL 1.2 benchmark suites. All benchmarks are evaluated against the default parameters and the reference input size.

ResNet18. ResNet18 is an 18-layer convolutional neural network model. It is a popular image classification model that is used on low-power edge devices. We implement a benchmark that classifies 10 images using the ResNet18 model. The model uses the single-precision, pre-trained weights (`IMAGENET1K_V1`) from PyTorch 1.12.1 [64].

BERT. BERT is a large transformer model that powers various natural language processing tasks. We derive a benchmark from the NVIDIA FasterTransformer backend [62]. We use the BERT_BASE configuration [29]. The model has 12 layers

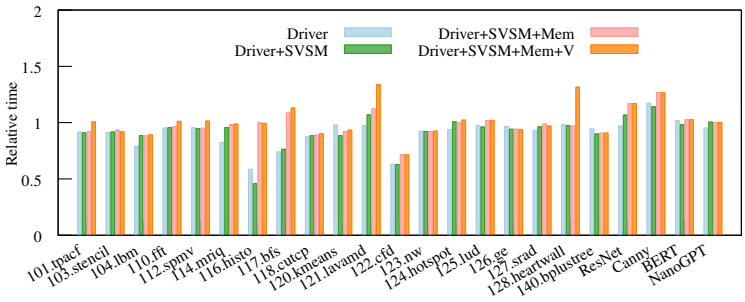

Figure 5: Relative execution time for the five benchmark suites evaluated on Honeycomb.

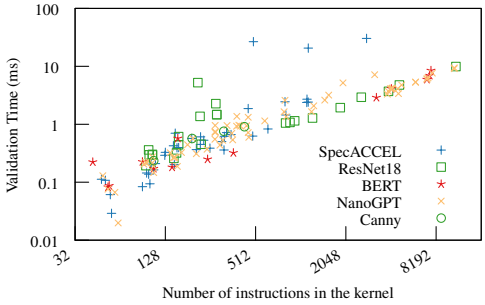

Figure 6: The time spent on validating GPU kernels in wall clock vs. the size of the kernels.

and 110M parameters, preloaded with the single-precision, pre-trained weights called `bert-base-uncased` [38]. The benchmark reports the time of performing a single shot of inference on BERT.

NanoGPT. NanoGPT is a minimal implementation of training medium-size Generative Pre-trained Transformer (GPT) models. GPT models are often used to power chat bots or to generate human-like content. We implement a benchmark that fine-tunes the GPT2 model [68] using the tiny Shakespeare dataset in the NanoGPT repository. We preload the weights of its 124M parameters from [39]. The benchmark trains with a batch size of 4 and uses ∼15 GB out of the 16 GB of total device memory available.

Canny. Canny implements the Canny edge detection algorithm to detect edges in images. We implement a benchmark that detects edges on an image in the UHDSR4K dataset [91]. The resolution of the image is 3840× 2160.

Figure 5 presents the relative execution time of all five benchmark applications. The relative execution time ranges from 0.89 (104.lbm) to 1.27 (Canny). Large language models in Honeycomb are particularly efficient: the relative slowdowns of BERT and NanoGPT are 2% and 0%. This is because their execution time is dominated by matrix multiplications, whose memory accesses can be efficiently reasoned about with scalar evolution analysis and polyhedral models. The validator requires no runtime checks to be inserted into the performance-critical, general matrix multiplication (GEMM) GPU kernels to pass validation. Honeycomb essentially launches the exact same GPU kernels compared to the baseline.

Figure 5 further breaks down the overheads into four categories: (1) Driver (slowdowns from an alternative driver), (2) SVSM (validating the requests in the command queues), (3) Mem (securing memory transfers) and (4) V (runtime checks). The characteristics of runtime overheads vary among applications. First, the alternative driver is simpler and faster in general but lacks the optimizations on large memory copies. Running Canny on the alternative driver is 18% slower (7.16s vs. 6.11s) because it loads an 8MB image into the GPU before processing it. Second, to enforce security the SVSM must inspect each request of kernel launch. The overhead is more evident for applications that mostly consist of small, fast GPU kernels like ResNet.

The third source of overheads is secure memory transfer. For example, 117.bfs copies the frontier and the tail of the BFS queue back and forth between the host and the device in each iteration, transferring 400 bytes of data for 108,000 times. Enabling secure memory transfer results in a 42% slowdown (13.97s vs. 9.82s). 116.histo also has significant overheads because it uses `memcpy()` to zero out a piece of device memory at the beginning of each iteration. Changing it to `memset()` eliminates the overheads.

The final source of overheads comes from the runtime checks in GPU kernels that are inserted to facilitate validations. Runtime checks slow down 121.lavamd by 19% (5.80s vs. 4.87s). However, most of the overhead can be attributed to one single runtime check. The GPU kernel writes to `a[b[i] + threadIdx.x * j]` in two-level nested loops. `i` and `j` are loop variables thus `b[i]` remains constant in the outer loop. Developer must insert a runtime check to aid the validation since `b[i]` is a value from the memory where the validator does not model. Note that the runtime check can be hoisted to the outer loop since checking the indices at the first and the last iterations of `j` is sufficient to guarantee safety. Hoisting the check effectively eliminates the overheads (both 4.87s for disabling runtime checks and hoisting the check to the outer loop). The case of 128.heartwall is similar. Extending the validation to understand hoisting is left to future work.

11.4 Overheads

The previous subsection has discussed the overheads on the runtime checks inside the GPU kernels. This section further studies other overheads introduced by the system design of Honeycomb, namely:

- Validating the GPU kernels at load time.

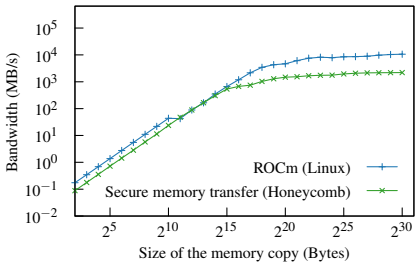

Figure 7: Round-trip bandwidth of data copies of various lengths between the host and the GPU, with and w/o secure memory transfers.

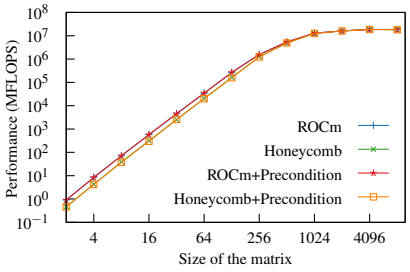

Figure 8: The achieved FLOPS in matrix multiplication on various sizes of the matrices. Both the x and y axes are on log scales.

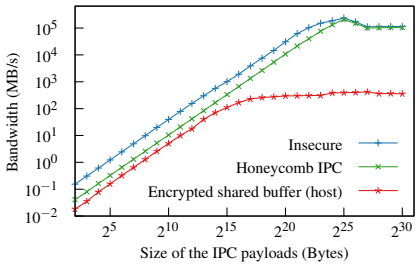

Figure 9: Round-trip IPC bandwidth of the ping-pong application with different sizes of payloads.

- Securing memory transfers between the host and the GPU.
- Checking the preconditions against the arguments and launching the GPU kernels.

Overheads of validating GPU kernels. Figure 6 describes the time spent on validating the GPU kernels in all five benchmark suites we evaluate vs. the number of instructions they have. The time used by validation is roughly linear with respect to the size of the GPU kernel. Out of 149 GPU kernels we have evaluated, the largest one is a GEMM GPU kernel that has 11297 instructions coming from rocBLAS [49]. The longest time spent on validating an individual GPU kernel is around 30 ms (128.heartwall). At the application level, validating NanoGPT training takes the longest time in our evaluation. It consists of 73 GPU kernels, taking 162 ms in total to validate all of them. Note that the validation is a one-time overhead when loading the applications. Real-world GPU applications like training execute the kernels continuously for days. The evaluation shows that validating GPU kernels is efficient and has negligible overheads on overall application performance.

Overheads of secure memory transfers. We study the overheads of secure memory transfers using a benchmark that transfers data back and forth between the host and the GPU. A round trip of a secure memory transfer includes (1) encrypting the data on the host CPU, (2) copying the encrypted data to and from the GPU, and (3) decrypting the data. We warm up the benchmark for 5 seconds and report the average transfer bandwidth over a 30-second period for various sizes of transfers. Figure 7 presents the round-trip data bandwidth with and without secure memory transfers. The bandwidths of both ROCm and Honeycomb first increase linearly with the sizes of the payloads and then peak at 10.63 and 2.20 GB/s. The bandwidth is bounded by AES encryption/decryption throughput of a single CPU core.

Overheads of checking preconditions and launching GPU kernels. We study the performance impacts of checking preconditions in Honeycomb by measuring the performance of multiplying two single-precision square matrices of various sizes in Honeycomb. We implement the benchmark using the GEMM GPU kernels from rocBLAS. The validator has verified that all global memory accesses in these GPU kernels are safe, thus there are no extra runtime checks inside the GPU kernels. We have further ported all checks on preconditions directly into the benchmark, making precondition checking the sole overhead in this benchmark.

Figure 8 presents the achieved FLOPS on both Linux and Honeycomb against various sizes of the square matrices (from 2×2 to 8192×8192), with or without checking the preconditions. Honeycomb performs 41 range checks on the kernel arguments to validate the preconditions on each launch, taking roughly 0.04μs to complete. All GEMM GPU kernels in the benchmark have the identical function signature. Both the number of checks on preconditions and the performance are consistent.

Honeycomb is slightly slower than Linux when the size of the matrices is less than 1024, because SVSM must check each request to ensure that applications can only launch validated kernels. We observe that the overhead is $\sim$ 8μs per launch of GPU kernels. To cross-validate the overheads, we compare the latency of launching a no-op kernel on Linux and on Honeycomb. The average latencies over a million launches on Linux and on Honeycomb are 13.15μs and 25.06μs. The overhead of checking preconditions is two orders of magnitude smaller than launching a no-op kernel.

11.5 IPC performance

We study the case of exchanging data between two TEE applications on the same host. We compare the bandwidth between exchanging the data via (1) an encrypted shared buffer on the host, and (2) the IPC mechanism in Honeycomb, where no encryption is needed. We have built two applications and evaluated their performance: (1) a ping-pong application that sends data back and forth, and (2) a two-stage image processing application that mimics the perception pipelines in autonomous vehicles. It ingests a video stream in an enclave

and performs edge detection in another one. The isolation not only increases modularity and robustness, but also simplifies the integrations with third-party vendor SDKs.

Figure 9 presents the effective bandwidth of the ping-pong application with different sizes of IPC payloads, along with a reference, insecure implementation that copies the payloads within the device memory using `hipMemcpyDtoD()`. IPC in Honeycomb is 2-529× faster compared to exchanging data via a shared encrypted memory buffer on the host. The effective bandwidth of round-trip IPCs peaks at ∼233 GB/s (89% of the reference insecure implementation) when sending payloads of 32 MB, where three buffers of such size utilize the shared L3 cache of the GPU (128 MB). In contrast, the bandwidth of using an encrypted shared buffer for IPC peaks at ∼411 MB/s. We attribute the inefficiency to the fact that GPU TEEs make secure memory transfers transparent to applications. Other GPU TEEs cannot give out the encryption keys to the applications without compromising the security, so the only way of sharing data securely is to re-encrypt the data before sharing them, where the performance is bounded by the CPU performance of encryption and decryption as shown in Figure 7.

We have assembled the Canny application into a two-stage image processing pipeline. The end-to-end latencies of processing a single frame of 4K image are 679 μs and 18579 μs when using direct IPC and an encrypted shared buffer on the host, where 617 μs is spent on actual computation.

11.6 Developer experience

Figure 10 presents the metrics on the kernels and development efforts of the five benchmark suites we have evaluated. It presents the number of GPU kernels, the number of memory instructions, the number of runtime checks inserted, and the number of preconditions written for each application. Neural network applications ResNet18, BERT, and NanoGPT are considerably bigger, where the GEMM kernels contribute to more than 70% of the total number of instructions. RocBLAS launches different GEMM kernels based on the sizes of matrices for optimal performances.

Experience with neural network models and the Canny edge detector. We have ported 109 GPU kernels in total for ResNet18, BERT, Nano and Canny. We are able to classify the GPU kernels used in neural network models into three categories: (1) element-wise operations, (2) matrix operations, including multiplication, transposition and convolution, and (3) special-purpose GPU kernels such as Im2d2col or radix sort. Note that developers do not directly write the GPU kernels. The GPU kernels either come from well-optimized libraries such as MIOpen [49] or are generated by PyTorch.

We found that GPU kernels in the first two categories have well-optimized, regular memory access patterns. Scalar evolution analysis and polyhedral models are sufficient to verify the safety of the memory accesses, meaning that no extra runtime checks are required. However, it is important to extend the polyhedral models to treat the values of some kernel arguments as constants (§5) to complete the analysis. Many of these GPU kernels are generic library functions. They take the shape and the length of the data as arguments, which are often used in calculating addresses. GPU kernels used in the Canny edge detector also fall within the first two categories.

GPU kernels in the third category require case-by-base discussion. The class of Im2d2col GPU kernels used by ResNet18 essentially unrolls a matrix into a long vector under different configurations. The challenge of analyzing their memory access is that the GPU kernels use division and modulo operations to transform the basis of indices. For example, the statements `out_x = inner_lid % out_cols_wg; out_y = inner_lid / out_cols_wg;` repartition the index `inner_lid` based on the value of `out_cols_wg`. It is easy to see such accesses are inbound but neither the standard scalar evolutions nor polyhedral representations can model them. Such repartitions are often parts of the tight loops thus extra runtime checks can incur significant performance overheads. Fortunately, we found out that the compiler generates pretty stable code sequences for these statements. We have implemented a pattern matching algorithm to iterate over the instructions to uncover the semantics of repartitions, so that the validator can verify these Im2d2col GPU kernels without the need of runtime checks.

Radix sorts are introduced to speed up the training of neural networks on GPUs. Particularly, during the backward propagation pass the training application sorts the sparse gradients before propagating the values in order to improve locality and to save the precious memory bandwidth. While radix sorts are efficient on GPUs, they pose challenges for validation due to the presence of indirect memory references. It is a non-goal for the validator in Honeycomb to verify the safety of indirect memory references, so we have added runtime checks to the sorting kernels in the NanoGPT training application. The overall overheads are insignificant as radix sorts are accountable for less than 0.02% of the total running time. Replacing radix sort with an algorithm like merge sort that is more friendly to validation may be a good alternative.

Many preconditions are mechanical and usually straightforward (e.g., ensuring that the whole matrix is in the private region). Since GPU kernels take data shapes as inputs, all of which must be specified in the preconditions. For instance, each GEMM kernel requires 30 preconditions. Writing these preconditions is tedious, and we have developed a script to generate the preconditions automatically.

In short, it requires inserting zero runtime checks into ResNet18, BERT and Canny to pass validation in Honeycomb. We introduce runtime checks in the NanoGPT training application with negligible performance overheads. Developing preconditions for the GPU kernels requires modest effort. Patching frameworks like PyTorch to use the validated versions of the GPU kernels, however, turns out to be a big-

Benchmark	101.tpacf	103.stencil	104.lbm	110.fft	112.spmv	114.mriq	116.histo	117.bfs	118.cutcp	120.kmeans	121.lavamd	122.cfd	123.nw	124.hotspot	125.lud	126.ge	127.srad	128.heartwall	140.bplustree	ResNet18	BERT	NanoGPT	Canny
Kernels	1	1	1	1	1	2	5	2	1	2	1	5	2	1	3	2	6	1	2	24	14	67	4
Mem. instrs.	15	15	30	9	9	21	77	14	10	14	7	119	76	8	105	20	60	121	46	1,531	1,768	7,202	55
Checks	0	0	0	0	5	0	1	8	3	0	5	5	0	0	0	0	6	14	16	0	0	44	0
Preconds.	5	8	5	6	9	13	29	16	8	12	4	29	20	8	9	13	45	25	19	443	181	1,529	18

Figure 10: Metrics on the kernels and development efforts to validate GPU kernels in the five evaluated benchmark suites.

ger practical challenge. We eventually end up patching the userspace runtime to load the validated GPU kernels.

Experience with the SpecACCEL benchmark suites. We have ported all 19 benchmarks (40 kernels in total) in the SpecACCEL 1.2 benchmark suites to Honeycomb. We classify the required changes into three categories: (1) adding optimization, (2) undoing optimization, and (3) indirect heap references.

Adding optimization. The validator can benefit from optimizing the GPU kernel. For example, 110.fft has a division instruction in the kernel. The divisor is a power-of-2 constant. Propagating it into the GPU kernel reduces the division into bit shifts, simplifying the validation.

Undoing optimization. Aggressive optimization in compilers issue instruction sequences that are difficult to model in scalar expression. For example, the compiler compiles the expression `-1-bx` in 123.nw to a single instruction `s_not_b32 bx`. It is difficult for the validator to model such an instruction as a scalar expression. We have to rewrite the expression to undo the optimization so that the validator can recognize the expression.

Indirect heap references. There are 9 benchmarks that have indirect heap references in the code. Each instance of irregular heap access requires adding a runtime check which incurs runtime overheads. For example, 118.cutfp casts a float to the index of an array; other benchmarks like 112.spmv expose patterns like `a[b[i]]`. All these instances require adding runtime checks to pass the validations.

12 Related work

TEE designs on GPUs. GPU TEEs enforce isolation among mutually distrusted enclaves. Graviton [83] augmented the GPU hardware with RMP tables to isolate physical memory pages among enclaves. Telekine [42] was built upon Graviton to remove a side channel regarding the execution time of GPU kernels, enhancing the overall isolation confidence. HIX [44] and CRONUS [45] relied on the GPU driver's isolation mechanisms to properly protect and isolate applications. However, modern GPU drivers are inherently complex, and even security features like isolation are prone to vulnerabilities [24, 27]. StrongBox [28] leveraged the secure IOMMU on the SoC to isolate enclaves on integrated GPUs. It required updating the IOMMU and flushing the IOMMU TLB to switch between different execution environments.

HETEE [95] deployed a cluster of tamper-resistant servers with commodity GPUs. These servers accessed secure accelerator boxes through a centralized FPGA-based controller, achieving isolation through physical separation. Visor [66] focused on privacy-preserving video analytics in the cloud. It combined oblivious algorithms at the application level and a hybrid TEE at the system level to provide isolation.

Honeycomb enforces isolation via confining the behaviors of GPU applications with static analysis. Honeycomb's approach complements the hardware limitations of existing GPUs, reduces overheads, and creates opportunities for optimizations like directly sharing data via IPC. Performing IPC in current GPU TEEs requires copying the data back and forth through an encrypted shared memory buffer on the host. Honeycomb combines confinements from static analysis and system-level designs to reduce IPC into copying plaintext within the device memory. IPC in Honeycomb is up to two orders of magnitude faster than conventional methods, enabling real-world applications to adopt a more modular architecture with modest overheads.

GPU TEEs also enforce isolation between enclaves and the untrusted host environment. They need to establish secure communication channels between the application running inside the CPU TEE and the GPU. Prior work implemented end-to-end secure communication channels in the GPU hardware [83], in the PCIe fabrics [44], or leveraging the secure IOMMU inside the SoC [28, 45]. Honeycomb leverages existing work on secure I/O bus [1, 44, 65] or software-based solutions [94] to establish secure communication channels.

Crypto-based secure computing on GPUs. Recent advances in modern cryptography offer theoretically provable solutions for privacy-preserving computing, such as Multi-Party Computation (MPC), Garbled Circuit (GC) and Homomorphic Encryption (HE). These algorithms have been used to realize secure GPU computations for machine learning and data analytics. On top of GAZELLE [47], Delphi [56] used GPU to accelerate the HE-based linear operations, and also selectively replace the expensive GC-based nonlinear ReLU opera-

tors with polynomial approximations. CryptGPU [78] further implemented both linear and nonlinear operations in MPC-based protocols on GPUs. It embedded the secret-shared value computations into floating-point operations, effectively utilizing GPU hardware units. GForce [61] instead focused on inference and addressed the high latency of non-linear operators by applying new quantization approaches and employing GPU-friendly protocols. Finally, Piranha [84] was a general and modular framework for accelerating secret-sharing-based MPC protocols on GPUs, leveraging optimized integer-based GPU kernels and memory-efficient in-place computations.

The cryptographic solutions do not keep the plaintext values in the untrusted platforms, so they are more resistant to side-channel vulnerabilities. However, their substantially higher computational cost causes huge slowdown compared to native processing. Specialized hardware [50, 71, 72] and trusted hardware units [93] have been proposed to accelerate HE and MPC, but all require non-trivial hardware changes.

Slalom [81] took a different approach. It used a CPU TEE to compute the non-linear parts, and offloaded encrypted data to the untrusted GPU to process linear operations. Both confidentiality and integrity are guaranteed. DarKnight [36] further optimized the flow with a better encryption method that greatly reduced the communication cost between CPU and GPU as well as the computations involved in the CPU TEE.

Secure operating systems. There is fruitful research on improving the security of operating systems, including explicitizing the security policies [73, 90], applying safe languages in the OS kernel [13, 41], and proving properties via formal verifications [52, 54]. Honeycomb utilizes techniques including security monitors and virtualization [10, 79] to remove the Linux kernel and the device driver out of the TCB.

Software fault isolation (SFI). Lightweight fault isolations [46, 57, 88] have been proven effective on the x86 architecture. Essentially, validation in Honeycomb is a form of SFI for GPU kernels. Honeycomb, however, combines the SFI with an alternative memory layout and other system-level supports to extend the fault isolation to a secure execution environment.

Polyhedral analysis. There is rich literature on utilizing polyhedral representations for loop analysis and transformations [8, 14, 35, 58]. Researchers have extended the approaches to more general cases [12]. Honeycomb uses the polyhedral analysis to model the effects of GPU memory access and to ensure that the memory access conforms with the security policy.

13 Conclusion

Honeycomb demonstrates that static analysis (validation) is a practical and flexible technique to enforce security for GPU applications. Combining with hardware and OS support, Honeycomb's validation guarantees powerful system-wide invariants like every memory access in the applications conforms with the security policies. As a result, Honeycomb has reduced the size of TCB by 18×, and provided a secure IPC primitive that is 529× faster than conventional approaches.

The evaluation of Honeycomb on five representative benchmark suites, 23 applications in total, shows that Honeycomb is practical and efficient to provide secure GPU TEEs for real-world applications. It requires inserting few or none runtime checks into the GPU kernels to validate them, thus the runtime overhead is minimal. Large language model workloads like BERT and NanoGPT have ∼2% runtime overheads on Honeycomb.

The boom of GPU applications today requires continuous innovations in GPU software/hardware stack. Our experience on Honeycomb shows that static analysis has a lot of potential to help explore novel designs in the full software/hardware stack and to speed up innovations.

Acknowledgments

We would like to thank our shepherd, Christopher Rossbach, and the anonymous reviewers for their comments and feedback on our work. We thank Quanxi Li, Shuoming Zhang for their contributions on an early implementation of this work. We also thank the Stanford Platform Lab and its affiliates. This work was partially supported by the National Key R&D Program of China (2021ZD0110101) and the National Natural Science Foundation of China (62072262, 62090024, 62232015).

References

[1] Advanced Micro Devices, Inc. AMD SEV-TIO: Trusted I/O for secure encrypted virtualization. https://www.amd.com/system/files/documents/sev-tio-whitepaper.pdf.

[2] Advanced Micro Devices, Inc. HIP: C++ heterogeneous-compute interface for portability. https://github.com/ROCm-Developer-Tools/HIP.

[3] Advanced Micro Devices, Inc. ROCm. https://github.com/RadeonOpenCompute/ROCm.

[4] Advanced Micro Devices, Inc. AMD SEV-SNP: Strengthening VM isolation with integrity protection and more. White paper, 2020.

[5] Advanced Micro Devices Inc. and Hewlett-Packard Inc. PCI express access control services (ACS), 2006.

[6] Alfred V Aho, Ravi Sethi, and Jeffrey D Ullman. *Compilers: Principles, Techniques, and Tools*. Pearson Education, Inc., 2007.

[7] Zelalem Birhanu Aweke, Salessawi Ferede Yitbarek, Rui Qiao, Reetuparna Das, Matthew Hicks, Yossi Oren, and Todd Austin. ANVIL: Software-based protection against next-generation rowhammer attacks. In *ASPLOS*, 2016.

[8] Riyadh Baghdadi, Jessica Ray, Malek Ben Romdhane, Emanuele Del Sozzo, Abdurrahman Akkas, Yunming Zhang, Patricia Suriana, Shoaib Kamil, and Saman Amarasinghe. Tiramisu: A polyhedral compiler for expressing fast and portable code. In *CGO*, 2019.

[9] Jonathan Behrens, Anton Cao, Cel Skeggs, Adam Belay, M. Frans Kaashoek, and Nickolai Zeldovich. Efficiently mitigating transient execution attacks using the unmapped speculation contract. In *OSDI*, 2020.

[10] Adam Belay, Andrea Bittau, Ali Mashtizadeh, David Terei, David Mazières, and Christos Kozyrakis. Dune: Safe user-level access to privileged CPU features. In *OSDI*, 2012.

[11] Muli Ben-Yehuda, Michael D. Day, Zvi Dubitzky, Michael Factor, Nadav Har'El, Abel Gordon, Anthony Liguori, Orit Wasserman, and Ben-Ami Yassour. The Turtles project: Design and implementation of nested virtualization. In *OSDI*, 2010.

[12] Mohamed-Walid Benabderrahmane, Louis-Noël Pouchet, Albert Cohen, and Cédric Bastoul. The polyhedral model is more widely applicable than you think. In *CC*, 2010.

[13] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility safety and performance in the SPIN Operating System. In *SOSP*, 1995.

[14] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral program optimization system. In *PLDI*, 2008.

[15] Ben Boyter. Sloc Cloc and Code. https://github.com/boyter/scc.

[16] Broadcom Inc. Videocore IV 3D architecture reference guide. https://docs.broadcom.com/doc/12358545.

[17] David Brumley and Dan Boneh. Remote timing attacks are practical. *Computer Networks*, 2005.

[18] Erik Buchanan, Ryan Roemer, Hovav Shacham, and Stefan Savage. When good instructions go bad: Generalizing return-oriented programming to RISC. In *CCS*, 2008.

[19] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the keys to the Intel SGX Kingdom with transient Out-of-Order execution. In *USENIX Security*, 2018.

[20] John Canny. A computational approach to edge detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 1986.

[21] Guoxing Chen, Mengyuan Li, Fengwei Zhang, and Yinqian Zhang. Defeating speculative-execution attacks on SGX with hyperrace. In *DSC*, 2019.

[22] Francis S Collins and Harold Varmus. A new initiative on precision medicine. *New England journal of medicine*, 2015.

[23] CVE. CVE-2020-5991. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-5991.

[24] CVE. CVE-2021-1098. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-1098.

[25] CVE. CVE-2022-20186. https://cve.mitre.org/cgi-bin/cvename.cgi?name=2022-20186.

[26] CVE. CVE-2022-21821. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2022-21821.

[27] CVE. CVE-2022-31609. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2022-31609.

[28] Yunjie Deng, Chenxu Wang, Shunchang Yu, Shiqing Liu, Zhenyu Ning, Kevin Leach, Jin Li, Shoumeng Yan, Zhengyu He, Jiannong Cao, and Fengwei Zhang. StrongBox: A GPU TEE on arm endpoints. In *CCS*, 2022.

[29] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding, 2019.

[30] Dinakar Dhurjati, Sumant Kowshik, and Vikram Adve. SAFECode: Enforcing alias analysis for weakly typed languages. In *PLDI*, 2006.

[31] Whitfield Diffie and Martin E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 1976.

[32] Morris Dworkin, Elaine Barker, James Nechvatal, James Foti, Lawrence Bassham, E. Roback, and James Dray. Advanced encryption standard (AES), 2001.

[33] Petros Efstathopoulos, Maxwell Krohn, Steve VanDeBogart, Cliff Frey, David Ziegler, Eddie Kohler, David Mazières, Frans Kaashoek, and Robert Morris. Labels and event processes in the Asbestos Operating System. In *SOSP*, 2005.

[34] Earlence Fernandes, Jaeyeon Jung, and Atul Prakash. Security analysis of emerging smart home applications. In *IEEE S&P*, 2016.

[35] Tobias Grosser, Armin Groesslinger, and Christian Lengauer. Polly - performing polyhedral optimizations on a low-level intermediate representation. *Parallel Processing Letters*, 2012.

[36] Hanieh Hashemi, Yongqin Wang, and Murali Annavaram. DarKnight: An accelerated framework for privacy and integrity preserving deep learning using trusted hardware. In *MICRO*, 2021.

[37] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *CVPR*, 2016.

[38] Hugging Face Inc. BERT base model. https://huggingface.co/bert-base-uncased.

[39] Hugging Face Inc. GPT-2. https://huggingface.co/gpt2.

[40] Ralf Hund, Carsten Willems, and Thorsten Holz. Practical timing side channel attacks against kernel space ASLR. In *IEEE S&P*, 2013.

[41] Galen C. Hunt and James R. Larus. Singularity: Rethinking the software stack. *SIGOPS Oper. Syst. Rev.*, 2007.

[42] Tyler Hunt, Zhipeng Jia, Vance Miller, Ariel Szekely, Yige Hu, Christopher J. Rossbach, and Emmett Witchel. Telekine: Secure computing with cloud GPUs. In *NSDI*, 2020.

[43] Intel. Intel trust domain extensions. https://www.intel.com/content/www/us/en/developer/articles/technical/intel-trust-domain-extensions.html.

[44] Insu Jang, Adrian Tang, Taehoon Kim, Simha Sethumadhavan, and Jaehyuk Huh. Heterogeneous isolated execution for commodity GPUs. In *ASPLOS*, 2019.

[45] Jianyu Jiang, Ji Qi, Tianxiang Shen, Xusheng Chen, Shixiong Zhao, Sen Wang, Li Chen, Gong Zhang, Xiapu Luo, and Heming Cui. CRONUS: Fault-isolated, secure and high-performance heterogeneous computing for trusted execution environment. In *MICRO*, 2022.

[46] Evan Johnson, David Thien, Yousef Alhessi, Shravan Narayan, Fraser Brown, Sorin Lerner, Tyler McMullen, Stefan Savage, and Deian Stefan. Trust but verify: SFI safety for native-compiled Wasm. In *NDSS*, 2021.

[47] Chiraag Juvekar, Vinod Vaikuntanathan, and Anantha Chandrakasan. GAZELLE: A low latency framework for secure neural network inference. In *USENIX Security*, 2018.

[48] Andrej Karpathy. NanoGPT. https://github.com/karpathy/nanoGPT.

[49] Jehandad Khan, Paul Fultz, Artem Tamazov, Daniel Lowell, Chao Liu, Michael Melesse, Murali Nandhimandalam, Kamil Nasyrov, Ilya Perminov, Tejash Shah, Vasilii Filippov, Jing Zhang, Jing Zhou, Bragadeesh Natarajan, and Mayank Daga. MIOpen: An open source library for deep learning primitives, 2019.

[50] Sangpyo Kim, Jongmin Kim, Michael Jaemin Kim, Wonkyung Jung, John Kim, Minsoo Rhu, and Jung Ho Ahn. BTS: An accelerator for bootstrappable fully homomorphic encryption. In *ISCA*, 2022.

[51] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors. In *ISCA*, 2014.

[52] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal verification of an OS kernel. In *SOSP*, 2009.

[53] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO*, 2004.

[54] Haohui Mai, Edgar Pek, Hui Xue, Samuel Talmadge King, and Parthasarathy Madhusudan. Verifying security invariants in ExpressOS. In *ASPLOS*, 2013.

[55] Microsoft Inc. Secure boot. https://docs.microsoft.com/en-us/windows-hardware/design/device-experiences/oem-secure-boot.

[56] Pratyush Mishra, Ryan Lehmkuhl, Akshayaram Srinivasan, Wenting Zheng, and Raluca Ada Popa. Delphi: A cryptographic inference service for neural networks. In *USENIX Security*, 2020.

[57] Greg Morrisett, Gang Tan, Joseph Tassarotti, Jean-Baptiste Tristan, and Edward Gan. RockSalt: Better, faster, stronger SFI for the x86. In *PLDI*, 2012.

[58] Ravi Teja Mullapudi, Vinay Vasista, and Uday Bondhugula. PolyMage: Automatic optimization for image processing pipelines. In *ASPLOS*, 2015.

[59] Mayur Naik, Alex Aiken, and John Whaley. Effective static race detection for Java. In *PLDI*, 2006.

[60] George C. Necula. Proof-carrying code. In *POPL*, 1997.

[61] Lucien K. L. Ng and Sherman S. M. Chow. GForce: GPU-friendly oblivious and rapid neural network inference. In *USENIX Security*, 2021.

[62] NVIDIA Inc. FasterTransformer 5.3. https://github.com/NVIDIA/FasterTransformer.

[63] NVIDIA Inc. Developing a Linux kernel module using RDMA for GPUDirect. Technical report, 2022.

[64] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in PyTorch. 2017.

[65] PCI-SIG. Integrity and data encryption (IDE) ECN. https://members.pcisig.com/wg/PCI-SIG/document/16599.

[66] Rishabh Poddar, Ganesh Ananthanarayanan, Srinath Setty, Stavros Volos, and Raluca Ada Popa. Visor: Privacy-Preserving video analytics as a cloud service. In *USENIX Security*, 2020.

[67] Moinuddin Qureshi, Aditya Rohan, Gururaj Saileshwar, and Prashant J. Nair. Hydra: Enabling low-overhead mitigation of row-hammer at ultra-low thresholds via hybrid tracking. In *ISCA*, 2022.

[68] Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. 2019.

[69] Adam Rodnitzky. Sensing breakdown: Waymo jaguar I-pace robotaxi. https://www.tangramvision.com/blog/sensing-breakdown-waymo-jaguar-i-pace-robotaxi, 2022.

[70] Francisco Romero, Mark Zhao, Neeraja J. Yadwadkar, and Christos Kozyrakis. Llama: A heterogeneous & serverless framework for auto-tuning video analytics pipelines. In *SoCC*, 2021.

[71] Nikola Samardzic, Axel Feldmann, Aleksandar Krastev, Srinivas Devadas, Ronald Dreslinski, Christopher Peikert, and Daniel Sanchez. F1: A fast and programmable accelerator for fully homomorphic encryption. In *MICRO*, 2021.

[72] Nikola Samardzic, Axel Feldmann, Aleksandar Krastev, Nathan Manohar, Nicholas Genise, Srinivas Devadas, Karim Eldefrawy, Chris Peikert, and Daniel Sanchez. CraterLake: A hardware accelerator for efficient unbounded computation on encrypted data. In *ISCA*, 2022.

[73] Alan Shieh, Dan Williams, Emin Gün Sirer, and Fred B. Schneider. Nexus: A new operating system for trustworthy computing. In *SOSP*, 2005.

[74] Mark Silberstein, Bryan Ford, Idit Keidar, and Emmett Witchel. GPUfs: Integrating a file system with GPUs. In *ASPLOS*, 2013.

[75] Paul Staat, Johannes Tobisch, Christian Zenger, and Christof Paar. Anti-tamper radio: System-level tamper detection for computing systems. In *IEEE S&P*, 2022.

[76] Standard Performance Evaluation Corporation. The SPEC ACCEL benchmark suite. https://www.spec.org/accel.

[77] Zhendong Su. Refutation unsoundness issue on a QF_UFNIA instance. https://github.com/Z3Prover/z3/issues/6693, 2023.

[78] Sijun Tan, Brian Knott, Yuan Tian, and David J. Wu. CryptGPU: Fast privacy-preserving machine learning on the GPU. In *IEEE S&P*, 2021.

[79] Shuo Tang, Haohui Mai, and Samuel T. King. Trust and protection in the Illinois Browser Operating System. In *OSDI*, 2010.

[80] Mohammadkazem Taram, Ashish Venkat, and Dean Tullsen. Context-sensitive fencing: Securing speculative execution via microcode customization. In *ASPLOS*, 2019.

[81] Florian Tramèr and Dan Boneh. Slalom: Fast, verifiable and private execution of neural networks in trusted hardware. In *ICLR*, 2019.

[82] Jo Van Bulck, Nico Weichbrodt, Rüdiger Kapitza, Frank Piessens, and Raoul Strackx. Telling your secrets without page faults: Stealthy page table-based attacks on enclaved execution. In *USENIX Security*, 2017.

[83] Stavros Volos, Kapil Vaswani, and Rodrigo Bruno. Graviton: Trusted execution environments on GPUs. In *OSDI*, 2018.

[84] Jean-Luc Watson, Sameer Wagh, and Raluca Ada Popa. Piranha: A GPU platform for secure computation. In *USENIX Security*, 2022.

[85] Chenhao Xie, Xie Li, Yang Hu, Huwan Peng, Michael Taylor, and Shuaiwen Leon Song. Q-VR: System-level design for future mobile collaborative virtual reality. In *ASPLOS*, 2021.

[86] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *IEEE S&P*, 2015.

[87] A. Giray Yağlikçi, Minesh Patel, Jeremie S. Kim, Roknoddin Azizi, Ataberk Olgun, Lois Orosa, Hasan Hassan, Jisung Park, Konstantinos Kanellopoulos, Taha Shahroodi, Saugata Ghose, and Onur Mutlu. BlockHammer: Preventing rowhammer at low cost by blacklisting rapidly-accessed DRAM rows. In *HPCA*, 2021.

[88] Bennet Yee, David Sehr, Gregory Dardyk, J. Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. Native client: A sandbox for portable, untrusted x86 native code. In *IEEE S&P*, 2009.

[89] Bo Yu, Wei Hu, Leimeng Xu, Jie Tang, Shaoshan Liu, and Yuhao Zhu. Building the computing system for autonomous micromobility vehicles: Design constraints and architectural optimizations. In *MICRO*, 2020.

[90] Nickolai Zeldovich, Silas Boyd-Wickizer, Eddie Kohler, and David Mazières. Making information flow explicit in HiStar. In *OSDI*, 2006.

[91] Kaihao Zhang, Dongxu Li, Wenhan Luo, Wenqi Ren, Björn Stenger, Wei Liu, Hongdong Li, and Ming-Hsuan Yang. Benchmarking ultra-high-definition image super-resolution. In *ICCV*, 2021.

[92] Jie Zhao, Bojie Li, Wang Nie, Zhen Geng, Renwei Zhang, Xiong Gao, Bin Cheng, Chen Wu, Yun Cheng, Zheng Li, Peng Di, Kun Zhang, and Xuefeng Jin. AKG: Automatic kernel generation for neural processing units using polyhedral transformations. In *PLDI*, 2021.

[93] Xing Zhou, Zhilei Xu, Cong Wang, and Mingyu Gao. PPMLAC: High performance chipset architecture for secure multi-party computation. In *ISCA*, 2022.

[94] Zongwei Zhou, Virgil D. Gligor, James Newsome, and Jonathan M. McCune. Building verifiable trusted path on commodity x86 computers. In *IEEE S&P*, 2012.

[95] Jianping Zhu, Rui Hou, XiaoFeng Wang, Wenhao Wang, Jiangfeng Cao, Boyan Zhao, Zhongpu Wang, Yuhui Zhang, Jiameng Ying, Lixin Zhang, and Dan Meng. Enabling rack-scale confidential computing using heterogeneous trusted execution environment. In *IEEE S&P*, 2020.