Technical Document for "Guardian GPU"

Yassin Hisham , Samer Rafik , Arwa Ali, Judy Ahmed, Khalid Mohamed

Supervised by Dr. Hossam Abdelrahman , Hamsa Mahmoud

Faculty of Computer Science
Misr International University, Cairo, Egypt

booktabs tabularx

Abstract

Guardian GPU is considered as a kernel-level Host Intrusion Detection System (HIDS). Its primary focus is providing protection to GPU workloads (Compute based) in a shared environment. Guardian GPU has a passive (agentless) architecture that relies on Event Tracing for Windows (ETW) for monitoring GPU packet streams and memory operations. The Guardian GPU framework has 2 main components – the Guardian Monitor (collection of kernel-level telemetry) and the Guardian Brain (a Python based analysis engine which uses an Isolation Forest machine learning model for real-time anomaly detection). Based on these capabilities, Guardian GPU is able to detect the difference between authorized high-performance compute tasks and threats (such as Cryptojacking) or unauthorized workloads (task performed without proper permission) and to use this information to increase security for GPU workloads in the Cloud and Workstation environments.

Contents

Abstract **i**

1 Introduction **1**

1.1 Motivation 1

1.1.1 Problem Statement 2

1.2 Aims and Objectives 2

1.3 Scope 3

1.3.1 In-Scope Functions 4

1.4 System Overview 5

1.4.1 Business Context 6

1.4.2 Users Characteristics 8

1.5 Project Management and Deliverables 9

1.5.1 Time Plan 10

1.5.2 Deliverables 10

2 Background and Related Work **12**

2.1 Introduction 12

2.2 Background 12

2.3 Related Systems 13

2.4 Summary 14

3 System Requirements Specification **15**

3.1 Introduction 15

3.2 Functional Requirements 16
3.2.1 System Functions 16
3.2.2 Detailed Functional Specification 18
3.3 Interface Requirements 18
3.3.1 User Interfaces 19
3.3.2 Command-Line Interface (CLI) 20
3.3.3 Graphical User Interface (GUI) 21
3.3.4 Detection Engine Notifications 21
3.3.5 Hardware Interfaces 22
3.4 Design Constraints 24
3.4.1 Standards Compliance 24
3.4.2 Software Constraints 25
3.4.3 Hardware Constraints 25
3.4.4 Other Constraints as appropriate 26
3.5 Non-functional Requirements 26
3.6 Summary 27

4 System Design **28**

4.1 Introduction 28
4.2 Architecture Design viewpoints 29
4.2.1 Context viewpoint 29
4.2.2 Composition viewpoint 31
4.2.3 Logical viewpoint 35
4.2.4 Patterns use viewpoint 36
4.2.5 Algorithm viewpoint 40
4.2.6 Interaction viewpoint 41
4.2.7 Interface viewpoint 43
4.3 Data Design 44
4.3.1 Data Description 44
4.3.2 Dataset Description 49

4.3.3 Database Design 51
4.4 Human Interface Design 53
4.4.1 User Interface 53
4.4.2 Screen Images : 55
4.4.3 Screen Objects and Actions 55
4.5 Implementation 55
4.6 Testing Plan 58
4.6.1 Test Scenario X 58
4.6.2 Test Scenario Y 60
4.7 Test Results Summary 61
4.7.1 Key Performance Indicators 61
4.7.2 Limitations 62
4.8 Requirements Matrix 62
4.8.1 Key Implementation Details 62
4.9 System Deployment 64
4.9.1 Frameworks 66
4.9.2 Tools 66
4.9.3 Technologies 67
4.10 Summary 68

Bibliography **69**

Appendices **71**

A Git Repository **71**

List of Tables

1.1 Project Time Plan 10

3.1 Requirement Specification for Threat Intervention Popup 19
3.2 User Intervention Options 22
3.3 Public Data Exchange Schema Fields 23

4.1 Class Description: EtwMonitor 38
4.2 Class Description: UnifiedLogger 38
4.3 Class Description: DecisionOrchestrator 39
4.4 Guardian GPU Self-Generated Dataset Characteristics 49
4.5 Project name time plan 51
4.6 Requirements Traceability Matrix for Guardian GPU 63

List of Figures

1.1 System overview 7

3.1 Use case diagram 18

4.1 blackbox 31
4.2 MVC 32
4.3 UML Component Diagram 34
4.4 UML 37
4.5 Flowchart 42
4.6 System Initialization 56
4.7 Real-Time Monitoring 56
4.8 Anomaly Detection 56
4.9 Active Mitigation 57
4.10 Session Summary 57
4.11 Diagnostic View 58

A.1 Git Insights 71

Chapter 1

Introduction

Guardian GPU is a Host Intrusion Detection System (HIDS) that runs at the kernel level on your operating system, designed specifically for monitoring the compute activity of a shared GPU. With the ever-growing need for more computing power, the demand for GPUs has shifted them from specialized hardware (gaming, graphics, etc.) to being the infrastructure of cloud environments and multi-tenant data centers. This shift in usage from dedicated hardware to shared resources is outpacing the development and implementation of secure computing protocols. Guardian GPU takes advantage of this lack of security by monitoring the GPU for unauthorized usage, such as crypto-jacking, stealthy data exfiltration, and unauthorized compute workloads. Guardian GPU leverages a passive, agentless architecture and Event Tracing for Windows (ETW) to ensure it is not detected by anti-cheat systems while providing detailed visibility to each GPU packet at a granular level. A real-time and proactive means of securing GPU environments is critical to defend against any future threats.

1.1 Motivation

Graphics processing units (GPUs) have changed from being dedicated computing components used by gaming and video editing to being the backbone of cloud technologies and a multi-tenant datacenters. GPUs had previously been inaccessible to traditional security software due to them being treated as 'black boxes' where there was no way for traditional security software to see how a GPU processes memory through its interaction with non-GPU memory. Therefore,

there is now an urgent need to address GPU usage security deficiencies in the context of all aspects of datacenter infrastructures.

1.1.1 Problem Statement

The risk of GPU-related incidents in modern computing environments includes the following:

- **Unauthorized or Criminally Acquired GPU Resource Usage** — Adversaries may exploit large numbers of GPU resources to perform cryptomining or other unauthorized compute workloads on behalf of legitimate users. Such activity degrades system performance, increases operational costs, and may go unnoticed by traditional CPU-focused monitoring tools.
- **Undetected or Unrecovered Stealthy Data Exfiltration** — Sophisticated misuse of Direct Memory Access (DMA) mechanisms enables attackers to transfer sensitive data through GPU memory pathways. These transfers can bypass conventional storage and network monitoring systems, allowing data exfiltration to occur without triggering standard security controls.
- **Detection Evasion of Monitoring Tools** — Monitoring the performance, integrity, and compliance of large GPU fleets remains challenging. Many modern anti-cheat and anti-tamper systems actively detect and disable monitoring agents. As a result, a low-overhead, passive, and stealth-resilient monitoring approach is required to enable continuous GPU oversight without being quarantined or blocked.

1.2 Aims and Objectives

Guardian GPU aims to implement a non-intrusive, real-time monitoring system for GPU workloads on Microsoft Windows platforms. The primary design goals of the system include the following:

- **Passive Telemetry** — Leverage Event Tracing for Windows (ETW) to monitor GPU activity via the `Microsoft-Windows-DxgKrnl` provider, enabling kernel-level visibility

without injecting code into user applications or GPU drivers.

- **Decoupled Architecture** — Employ a modular design consisting of a high-performance C++ *Guardian Monitor* (Data Plane) responsible for telemetry collection, and a Python-based *Guardian Brain* (Analysis Plane) responsible for real-time analytics and machine learning inference.
- **Autonomous Detection** — Utilize an unsupervised Isolation Forest machine learning model to identify anomalous GPU behavior by detecting data points that are statistically rare or behaviorally distinct, without reliance on predefined attack signatures.
- **High-Fidelity Classification** — Accurately distinguish between legitimate high-compute workloads (e.g., gaming or authorized compute tasks) and malicious activity by analyzing GPU execution characteristics, including compute-to-packet ratios and execution timing patterns.

1.3 Scope

Guardian GPU is defined as a system for overseeing the security of workloads executed on Graphics Processing Units (GPUs) within Microsoft Windows 10 and Windows 11 environments. At its current stage of development, Guardian GPU focuses on the following functional areas:

- **Compute Signatures** — Capturing and storing metrics related to GPU compute execution, including total compute time and utilization percentage.
- **Packet Dynamics** — Recording and analyzing the volume, frequency, and temporal behavior of Direct Memory Access (DMA) packets generated by GPU workloads.
- **Stealth Operation** — Operating exclusively at the kernel event level using native telemetry mechanisms, ensuring invisibility to user-mode applications and resistance to detection or interference by anti-cheat or anti-tamper systems.

1.3.1 In-Scope Functions

The Guardian GPU system provides the following in-scope capabilities:

- **1. Kernel-Level Monitoring Capability**

  The system uses Event Tracing for Windows (ETW) to perform kernel-level GPU activity monitoring because it enables passive monitoring of GPU activities at the kernel level.

- **2. Precise Performance Tracking**

  The system achieves precise tracking of GPU packet processing and memory movements and power consumption through its main tracking system which does not require any vendor-specific application programming interfaces.

- **3. Anomaly Detection with Machine Learning**

  The system utilizes an unsupervised machine learning model based on Isolation Forest to identify unusual GPU operation patterns. The model enables continuous learning which permits it to adjust to changing workload patterns while detecting potential dangers such as unauthorized cryptomining activities.

- **4. Heuristic-Based Threat Detection**

  The system uses deterministic heuristic algorithms to identify suspicious GPU activity patterns through its Heuristic-Based Threat Traps feature. Stealthy Zero-Copy data exfiltration attempts become apparent through an abnormal increase in Direct Memory Access DMA packet transmission that occurs alongside reduced processing duration.

- **5. Anti-Cheat Compatibility**

  The system operates without requiring agents because it detects cheat software through its anti-cheat-safe operation. The system maintains complete operational functionality during GPU performance testing because it does not utilize Dynamic-Link Library DLL injection or user-mode instrumentation methods. The system allows full operational operation during GPU performance testing because it uses both BattlEye and Vanguard anti-cheat systems and anti-tamper protections.

- **6. Alerting and Notification System**

  The Guardian GPU system generates instant alerts together with system health status indicators whenever it discovers anomalies.

- **7. Project Scope and Boundaries**

  The current boundary of the project includes only detection and alerting through HIDS while automated threat response will be developed in subsequent project stages.

- **8. Implementation Architecture**

  The system operates completely through its software implementation which needs no hardware components. The system consists of two parts which include a kernel-level telemetry collection system developed in C and a user-space machine learning analysis system built with Python.

- **9. Analysis Methodology**

  The analysis process uses run-time analysis methods to evaluate all program components without inspecting their source code or binary files.

1.4 System Overview

The Guardian GPU system operates as an autonomous control system which does not require any human support for its functioning. The system follows a closed-loop architecture which consists of four main operational parts. Monitor Detect Decide Visualize

- **Monitor Component** — The Monitor component collects SYSTEM telemetry data. The system maintains complete operational independence because it neither inserts code into the monitored application nor makes any modifications to the application. The MONITOR collects GPU telemetry through Event Tracing for Windows ETW system calls which the Windows Graphics Kernel Driver DxgKrnl generates. The system tracks GPU operational status which includes execution scheduling and computation time and memory usage and power consumption.

- **Detect Component** — The Detect component processes each telemetry stream in Near Real-Time. The system establishes normal GPU operation through its unsupervised anomaly detection algorithms which create a baseline of standard operation. The primary algorithm employed is the Isolation Forest which identifies deviations from expected execution patterns Outliers to flag potential MALICIOUS usage or MISUSE by host applications.

- **Decide Component** — The Decide component establishes the correct response based on its analysis of anomalies. The system uses a Hybrid Decision Engine which combines Static Heuristics and ML Scores to identify different types of threats. The system uses Guardian Vault for Historical Data and Knowledge Bank for Signatures to examine ambiguous cases which ensures accurate threat classification without requiring human review.

- **Visualize Component** — The Visualize component functions as the Monitoring Interface which administrators use to monitor system performance. The system shows real-time GPU telemetry while it identifies anomaly locations and sends "Red Alerts" through the Console Dashboard to provide real-time device performance and security status information.

1.4.1 Business Context

Business Landscape: Organizations today use GPU infrastructure for high-performance computing and Cloud Service Provider (CSP) research facility and data center operations. Security teams need to implement strong security measures throughout shared GPU environments because multiple workloads execute inside these environments and users need to protect their confidential information and maintain system uptime. The following challenges represent key organizational obstacles.

- **Cryptojacking:** unauthorized users mine cryptocurrencies through GPU resources which leads to additional costs and system performance problems.

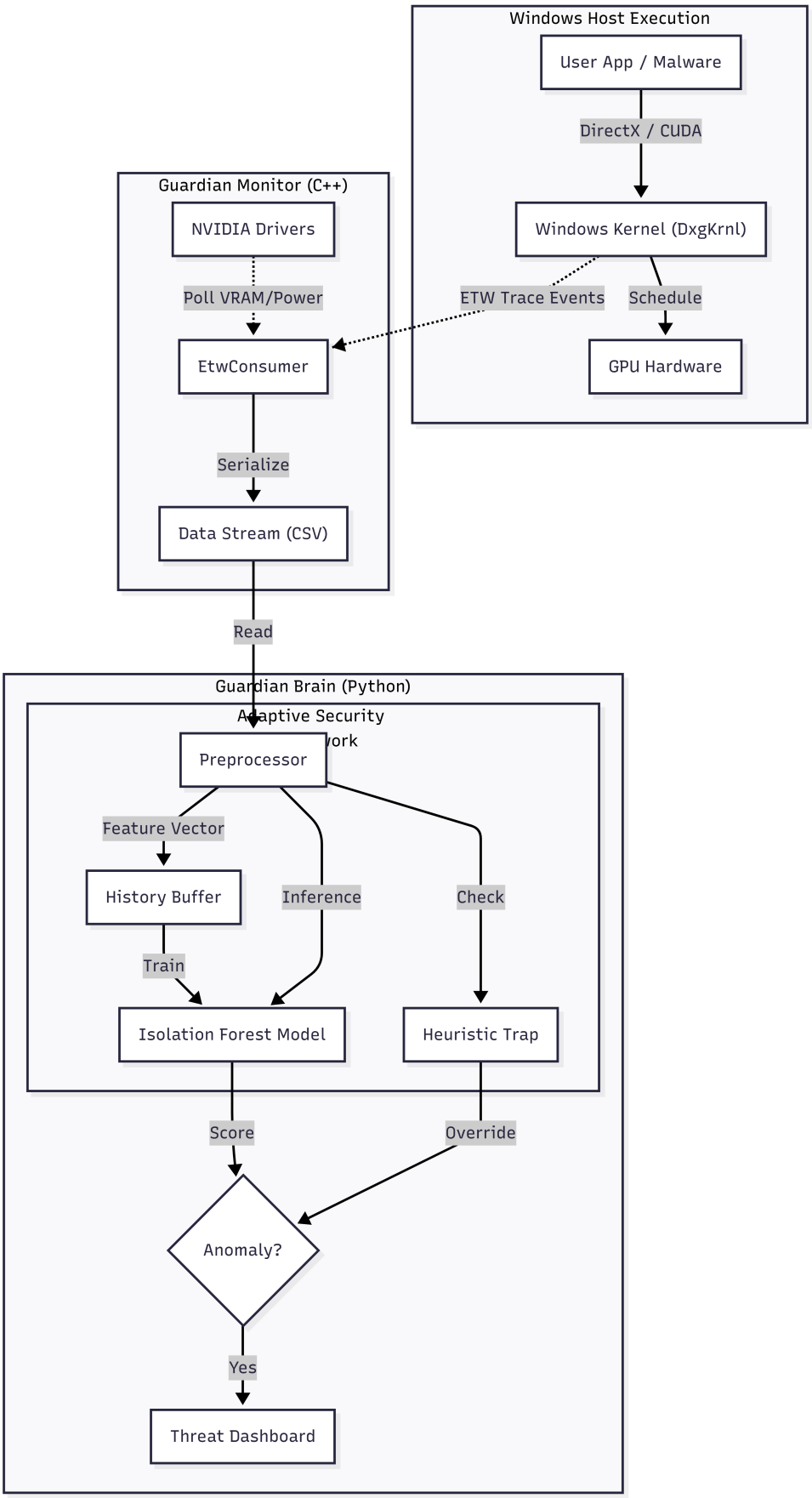

Figure 1.1: System overview

- **Data Leakage:** The extraction of sensitive data across secure boundaries represents a significant organizational risk.
- **Resource Inefficiencies:** Companies experience increased operational costs and decreased productivity because they fail to use GPU resources effectively.

The Guardian GPU System uses an automated security system that operates in real time to protect GPU resources from unauthorized access while maintaining operational security. The Guardian Monitor (C++) and Guardian Brain (Python) components work together to detect and assess threats while sustaining their normal business operations.

1.4.2 Users Characteristics

The Guardian GPU system exists to serve operators who handle advanced computing environments with high operational demands. The total system operation requires:

- **System administrators** who handle all aspects of GPU infrastructure upkeep including system health monitoring and configuration management.
- **Security administrators** who maintain system integrity through their work which includes security control implementation.
- **Operational Administrators:** Oversee continuous system operation, ensuring 24/7 uptime and resource availability.

Key Requirements:

- The system requires continuous 24 hour monitoring which provides persistent access to GPU operation data.
- The system needs automated detection which can identify security threats including cryptojacking through its real time intrusion detection system.
- The system achieves operational efficiency through a design that requires no additional operational costs while delivering maximum performance (Zero-Impact Design).

1.5 Project Management and Deliverables

Project Overview

The Guardian GPU project adopts an Agile–Research Hybrid development model. The method uses its experimental research to establish development processes which will create and test autonomous GPU security systems. The method uses its experimental research to establish development processes which will create and test autonomous GPU security systems.

Development Phases

- **Infrastructure and Data Acquisition (Weeks 1–6)**

  The team will build the Monitor component through two main tasks. The team will build the Monitor component through two main tasks. The team will build the Monitor component through two main tasks. The team will build the Monitor component through two main tasks. The team will build the Monitor component through two main tasks. The team will build the Monitor component through two main tasks. The team will build the Monitor component through two main tasks. The team will build the Monitor component through two main tasks.

- **Anomaly Detection Development (Weeks 7–15):**

  The team implemented The Detect component, which operates as The Brain. The team implemented The Detect component, which operates as The Brain. The phase conducted feature engineering to create data normalization processes and trained models for Isolation Forest and Local Outlier Factor which would identify normal GPU activities. The phase conducted feature engineering to create data normalization processes and trained models for Isolation Forest and Local Outlier Factor which would identify normal GPU activities.

- **Integration and Evaluation (Weeks 16–18):**

  The team developed Guardian Vault which functions as the Persistence Layer. The team developed Guardian Vault which functions as the Persistence Layer. The system under-

went testing to compare detection performance against simulated threat scenarios which included cuda_workload. The system underwent testing to compare detection performance against simulated threat scenarios which included cuda_workload.

1.5.1 Time Plan

Table 1.1: Project Time Plan

Phase	Duration	Activities	Deliverables
Phase 1: Research & Analysis	Weeks 1-4	• Conduct Literature Review on GPU Forensics. • Analyze Windows Kernel Tracing (ETW) & NVML capabilities. • Define Feature Vectors for Anomaly Detection.	• SLR Report • Requirements Doc
Phase 2: System Design	Weeks 5-8	• Design Split-Architecture (C++ Data Plane vs Python Control Plane). • Define Communication Interface (CSV/Pipe). • Develop a model to accurately differentiate between benign and malicious datasets.	• System Architecture • Class Diagrams • Initial Dataset
Phase 3: Implementation (Data Plane)	Weeks 9-12	• Implement `GuardianMonitor` (C++). • Integrate ETW Context Switch consumers. • Build `UnifiedLogger` for telemetry fusion.	• Monitor Executable • Raw Telemetry Logs
Phase 4: Implementation (Control Plane)	Weeks 13-16	• Develop `DecisionOrchestrator` (Python). • Train `IsolationForest` model. • Implement Heuristic "Tier 1" checks.	• Trained Model (.pkl) • Analysis Engine • Dashboard Prototype
Phase 5: Testing & Closure	Weeks 17-20	• Perform Attack Simulations (e.g., Cryptojacking). • Measure Latency (Target: $<$100ms). • Write final thesis and documentation.	• Final Release • Performance Report • Graduation Thesis

1.5.2 Deliverables

The primary deliverables of the project include:

- **1. GPU Security Framework:** Complete source code (C++/Python) for the autonomous security solution.

- **2. Curated Telemetry Database:** A structured dataset (.jsonl.gz) of GPU performance metrics collected for benchmarking.
- **3. Administration Dashboard:** A visual CLI interface for monitoring system status and alerts.
- **4. Technical Design Document:** Comprehensive documentation detailing system architecture and algorithms.
- **5. Performance and Validation Report:** An evaluation of anomaly detection accuracy and latency.
- **6. Final Thesis / Research Project Report:** A complete record of research objectives, methodologies, and conclusions.

Chapter 2

Background and Related Work

2.1 Introduction

The security of Graphics Processing Units (GPUs) in shared and multi-tenant environments, such as cloud computing infrastructure and high-performance computing clusters, has become an important issue. The rising use of GPUs for artificial intelligence and scientific simulations and big data processing has resulted in increased security risks from advanced threats. The traditional security paradigm for GPUs has been primarily centered on static threat detection, using predefined rules or signature-based alerts. The existing strategies have become inadequate for contemporary dynamic environments which experience ongoing threat evolution that demands advanced context-based defensive solutions. The current state of GPU security requires active systems which can automatically handle security threats according to the Guardian GPU framework which presents itself as a solution to this need.

2.2 Background

The concept of spatial sharing which permits multiple users to access GPU resources simultaneously needs to be handled because it creates security weaknesses that impact the fundamental security framework of contemporary GPU systems. The practice creates multiple security vulnerabilities because it enables attackers to exploit system weaknesses.

The virtualized or containerized environments contain memory protection deficiencies that per-

mit malicious or faulty kernel attacks to access another workload's memory which leads to confidentiality and integrity breaches through "Zero-Copy" attacks.

Cryptojacking refers to a type of malicious activity that permits attackers to take control of GPU computing resources without authorization which causes system performance issues and raises operational costs for organizations.

An attacker can use shared resources such as the L2 cache and memory buses to infer sensitive information such as a company's AI model parameters.

Modern monitoring systems require high-fidelity telemetry to handle these issues. Our system uses Event Tracing for Windows (ETW) to acquire kernel-level scheduling events from DxgKrnl.sys which enables us to monitor packet execution that had been invisible to user-space monitoring systems.

2.3 Related Systems

Researchers have thoroughly investigated GPU security measures in shared environments but their studies have primarily concentrated on two areas which include detecting anomalies and finding side-channel security weaknesses.

The researchers Ganepola and Bandara [1] studied different methods for detecting cryptojacking threats which demonstrated that Machine Learning (ML) models can accurately detect illicit mining activities through analyzing performance counters. The research presented by Tanana [7] introduced behavior-based detection systems which detect GPU cryptojacking by studying execution pattern data. Jiang et al. [2] established standard operational patterns which enable cloud environment researchers to detect abnormal behavior through their research work.

Zhang et al. [8] revealed two different types of security weaknesses which affect modern multi-GPU interconnect systems when they proved that attackers could use covert side-channel attacks to break into NVIDIA's NVLink system. The research conducted by Miao et al. [3] focused on the hidden paths inside GPU uncore systems which helped them to discover how data leaked from shared components that include L2 cache memory. The studies prove that hardware progress has not yet solved the fundamental problem of maintaining complete system separation from other system parts [4].

The existing detection techniques have reached advanced development stages, but there exists a need for systems that automatically adjust their response to combat threats. The system "Guardian" developed by Pavlidakis et al. [5] enables safe GPU sharing, but its main focus remains on providing fair access and maintaining system separation between users.
Existing research does not present any methods which can link high-frequency kernel telemetry data with immediate automatic response systems. Guardian GPU solves the problem of "detection-response disconnect" through its combination of Unsupervised Anomaly Detection which uses Isolation Forest and a Hybrid Decision Engine system. The system Guardian GPU uses a multilevel system design which permits it to carry out context-based defense operations without causing any performance delays to the host platform in contrast to the complete RL-based designs presented by Saqib et al. [6].

2.4 Summary

The chapter explained all existing components which determine how graphics processing unit security functions in shared computing environments. The research identified essential issues which arise during spatial sharing while showing how high-granularity telemetry (ETW) functions as an essential monitoring tool for today's cybersecurity threats. The review of related work demonstrated that while anomaly detection mechanisms—utilizing Machine Learning and side-channel analysis—are well-established, a significant deficiency remains in the area of automated mitigation. Existing frameworks were found to primarily address fairness or rely on static manual responses which failed to protect against current operational threats. The existing literature contains a significant research gap which this chapter reveals. The analysis establishes a theoretical base which supports our solution proposal Guardian GPU that combines unsupervised anomaly detection with an autonomous context-aware decision engine.

Chapter 3

System Requirements Specification

Explaining what your project does that is new or is better than existing work in the same field.

3.1 Introduction

The emergence of shared GPU resources in cloud computing and multi-tenant infrastructures has resulted in a variety of new security vulnerabilities, such as memory leaks, unauthorized access, cryptojacking, and malicious side-channel attacks. With respect to these latter two types of attacks, both exploit the lack of fine-grained visibility into the behaviour of GPUs when executing application workloads, which makes them difficult to detect using traditional CPU-oriented cybersecurity models. Additionally, existing GPU-specific security solutions are primarily detection-based and deployet static-based methods for identifying attacks, which often proves ineffective in the instances of either real-time or dynamic-based attacks. Furthermore, many of these solutions deploye highly-disruptive mitigation measures to address legitimate workloads and do not account for the fact that threat actors are constantly changing their tactics.

To mitigate some of these challenges, the dissertation presents the Guardian GPU framework - a real-time monitoring and security system that has been designed to address security concerns for shared GPU resources. This framework leverages a telemetry collection mechanism that operates continuously at both the system-kernel and hardware levels through the use of Event Tracing for Windows (ETW) to capture real-time process-level GPU activity and execution be-

haviours without needing to modify or instrument the monitored application workloads. Thus, the Guardian GPU framework provides full behavioural visibility into monitored application workloads, while remaining safe to deploy in a manner where the monitored workloads might be subject to security controls such as anti-cheat or otherwise sensitive to the workload's
This chapter presents the System Requirements Specification (SRS) for Guardian GPU. It defines the functional and non-functional requirements, operating environment, system interfaces, and design constraints of the proposed system. The requirements specified in this chapter are derived from a systematic review of existing GPU security solutions and are aligned with the goal of enabling fine-grained, real-time GPU behavior monitoring using kernel-level ETW telemetry. This SRS serves as the basis for the subsequent system design, implementation, and evaluation of the Guardian GPU framework.

3.2 Functional Requirements

3.2.1 System Functions

The *Guardian GPU Framework* has been created to provide a publicly accessible autonomous security framework that establishes an end-to-end security loop for shared GPU environments. The use case diagram below provides a high-level overview of the primary components of the framework, such as how the Autonomous Security Agent (RL Mitigation Agent) and System Administrator interact with each other and with the Guardian.
Functional requirements have been defined that outline the critical components required to create an autonomous security loop (i.e., an "Autonomous Security Loop"), which are all focused on the issues of *Necessity*, *Verifiability*, and *Attainability*. The following outlines what the functions of the Guardian, as defined by the functional requirements, will be:

- **GPU Telemetry Monitoring:** The Guardian will monitor the performance and activity of GPUs by continuously receiving and processing telemetry data from the GPUs using Event Trace for Windows (ETW) via passively subscribing to all GPU related kernel events (i.e., GPU events/commands) that are generated from within the Windows Graphics Kernel Driver (DxgKrnl.sys). The Guardian will collect telemetry data (e.g., process-

level usage, timing of execution, workload intensity metrics, and other measures) from the GPU without modifying or instrumenting any of the user processes that are using the GPUs.

- **Anomaly Detection:** The Guardian will use unsupervised machine learning methods to analyse the aggregate telemetry data for GPUs to identify the behaviour of the GPU which does not conform to the expected/normal behaviour of the GPU workload and also identify how to classify the observed execution pattern(s) of the GPU workload as normal/reasonable or anomalous/unreasonable.

- **Unified Telemetry Aggregation:** The Guardian will unify all of the telemetry metrics for every GPU that it has monitored and collected over time into a single, unified, and temporally relevant representation of GPU process performance (i.e., the Guardian will create a single video of performance of GPU processes over time).

- **Anomaly Detection:** Use unsupervised ML models to analyze telemetry data for anomalies in GPU workload behaviour and to identify captured executions as either normal or anomolous.

- **Decision and Action Control:** The Deciding & Acting functions are executed after an anomaly is detected to determine the current states of the system and if any mitigating actions are necessary.

- **RL (Reinforcement Learning) Based Mitigation Methodologies:** The agent uses RL as the basis for determining which adaptive control strategies (such as workload throttling, isolation, or intervention) are suitable for mitigating the detected anomaly.

- **Visualization/Monitoring:** Real-time access to telemetry and anomalies detected and the actions taken to mitigate them occur through monitoring interfaces provided by the system.

- **Policy Management and Configuration :** The System Administrator can configure the parameters used for monitoring and anomaly detection verification, and determine the security policies that govern all mitigation activities.

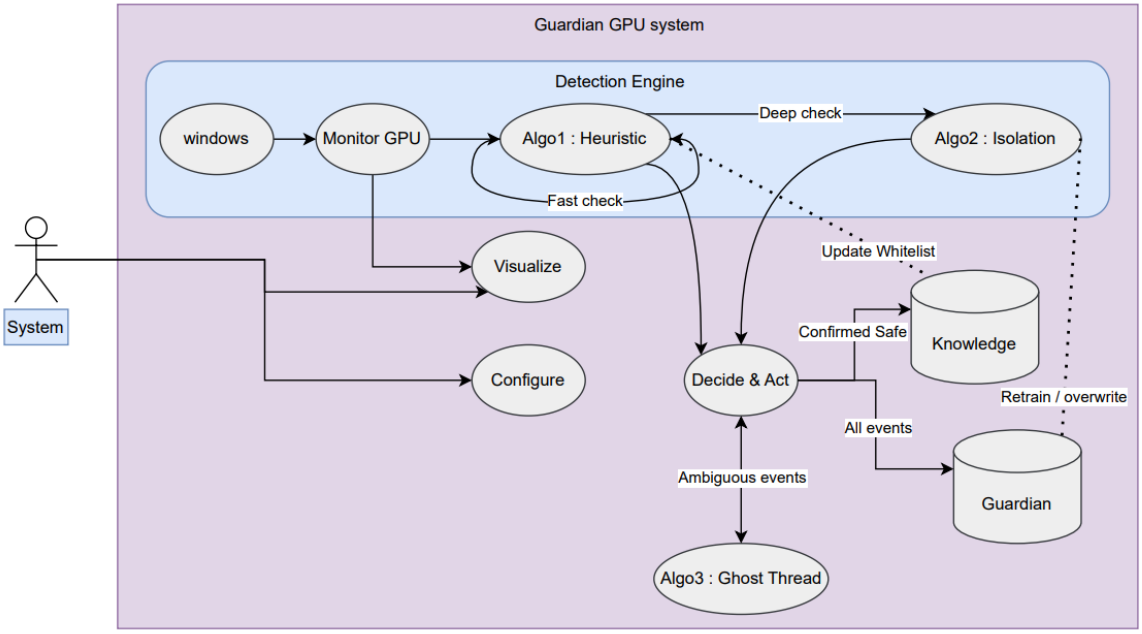

Figure 3.1: Use case diagram

- **Data Management and Reproducibility:** The system has time-series datasets to collect and store GPU telemetry and detected anomalies so that the data can later be retrieved, formatted, and analyzed.

- **Extensibility and Integration :** The modular architecture and well-defined interfaces of the system allow further extensions to be made to the system, such as the addition of new anomaly detection models and mitigation techniques. In addition, the system will allow for deployment in cloud or containerized environments.

3.2.2 Detailed Functional Specification

Show the details of all functions shown in section 3.2.1. Describe each function in the following structure.

3.3 Interface Requirements

This section describes how the software interfaces with other software products or users for input or output. Examples of such interfaces include library routines, token streams, shared

Name	Guardian GPU: Intelligent Guard for Shared Data Center GPUs
Code	FR-GUI-01
Priority	Extreme
Critical	Essential requirement; without it, no behavioral visibility or security analysis of GPU workloads can be performed.
Description	The system continuously collects low-level GPU activity/execution characteristics via Event Tracing for Windows (ETW). It uses the monitoring interface to passively subscribe to Windows graphics kernel (DxgKrnl.sys) emitted GPU-related kernel events. Process level GPU usage and execution patterns will be monitored via the collected ETW events to generate the data required for anomaly detection/miscellaneous mitigation decisions.
Input	**Primary:** ETW event stream from DxgKrnl GPU providers **Secondary:** Process and context identifiers associated with GPU events
Output	Time-stamped GPU telemetry records containing process-level GPU activity metrics, execution timing information, and workload behavior features derived from kernel events
Pre-condition	Windows operating system with ETW support enabled, Access to DxgKrnl GPU event providers, Required system privileges to start and consume ETW tracing sessions.
Post-condition	GPU telemetry data is delivered to the anomaly detection module in near real time without modifying or instrumenting user processes.
Dependency	*Windows Event Tracing for Windows (ETW) framework*, *Windows Graphics Kernel Driver (DxgKrnl.sys)*.
Risk	**Risk:** High-volume kernel event streams may increase processing overhead under heavy GPU workloads. **Mitigation:** Selective event filtering, adaptive trace session configuration, and efficient buffering mechanisms.

Table 3.1: Requirement Specification for Threat Intervention Popup

memory, data streams, and so forth.

3.3.1 User Interfaces

The system adopts a dual-interface model. The current implementation utilizes a Command-Line Interface (CLI) to support production deployment with minimal resource overhead. A Graphical User Interface (GUI) is planned for future development and will provide an intuitive mechanism for presenting system status, analytics, and alerts to end users.

GUI

This design specification for the graphical user interface (GUI) transitions from earlier dashboard implementations to a unified *Guardian Shield* dashboard built using Electron and React technologies. The initial feature set includes the following components:

- **Time-Series Anomaly Visualization** — A real-time graph plotting the global anomaly score on the y-axis against time on the x-axis. This visualization enables users to correlate performance degradation events, such as lag spikes, with the occurrence of security-related anomalies over time.
- **Process Manager** — A tabular view displaying all processes currently executing on the user's system. Users can interact with the table to whitelist trusted applications via a context menu, which updates the `knowledge_bank.json` signature database accordingly.
- **Toast Notifications** — Replacement of console-based alert messages with native Windows Toast Notifications for high-severity security events, ensuring timely and prominent user awareness of critical system conditions.

3.3.2 Command-Line Interface (CLI)

In the current release (v1.0), the Guardian GPU system provides two command-line interfaces:

- **Administrator Console (`GuardianMonitor.exe`)** — This console is intended for system administrators and may be executed with elevated privileges to enable full administrative access. It presents detailed, low-level telemetry derived from kernel hooks, including active GPU processes identified by process ID (PID), process name, video memory (VRAM) usage, power draw (in watts), and real-time GPU packet timing measured in nanoseconds. The Administrator Console is primarily used for deployment verification, system validation, and debugging.
- **Analysis Console (`GuardianBrain.py`)** — This console functions as a Security Operations Center (SOC) interface, continuously streaming inference results from the machine learning engine. It generates alerts when anomaly scores exceed predefined thresholds,

displays an overall system health score (ranging from 0 to 100%), and flags process identifiers (PIDs) deemed suspicious for further investigation.

3.3.3 Graphical User Interface (GUI)

The system employs a minimalist, event-driven GUI that remains hidden until the detection engine identifies suspicious activity, at which point a popup window alerts the user with specific action buttons to **Kill**, **Suspend**, or **Leave** the process.

3.3.4 Detection Engine Notifications

Notifications generated by the Guardian GPU system are tiered based on severity level and classified as follows:

- **Level 1 – Informational** — Informational logs are generated for routine events occurring during normal application execution. These events are written to a CSV log file for record-keeping and audit purposes.
- **Level 2 – Warning** — Warning logs are generated when anomalous behavior is detected by the Isolation Forest model, specifically when anomaly scores fall below a defined threshold (e.g., scores less than -0.5). These alerts are displayed in yellow within the Analysis Console.
- **Level 3 – Critical** — Critical logs are generated upon detection of known malicious signatures (e.g., heuristic traps) or extreme anomaly conditions indicative of malicious behavior. These alerts are displayed in red within the Analysis Console and are accompanied by an audible notification when enabled.

All notifications, regardless of severity, are structured and persisted to a daily GPU log file (`GPU_Log_YYYYMMDD.csv`). This persistent logging enables alert replay for forensic analysis and supports future model retraining and system improvement.

Suspicious Activity Pop-up

Upon the detection of a high-risk event, the system triggers a modal pop-up window. This window overrides the silent log stream to force user engagement. The user is presented with the following telemetry and action buttons:

Action	Description
Kill Process	Immediately terminates the process tree associated with the suspicious PID.
Suspend	Freezes the process execution in memory for further forensic analysis.
Ignore/Leave	Dismisses the alert and allows the process to continue execution (Whitelisting).

Table 3.2: User Intervention Options

3.3.5 Hardware Interfaces

Guardian GPU collects telemetry from GPU hardware facilities through two complementary mechanisms:

- **NVIDIA Management Library (NVML)** — The `nvml.dll` library enables Guardian to query physical sensor data that is not directly exposed through the Windows kernel. Collected telemetry includes:
  - *Thermal Sensors* (e.g., GPU core temperature measured in degrees Celsius).
  - *Fan Controllers* (e.g., pulse-width modulation (PWM) fan speed expressed as a percentage).
  - *Memory Controllers* (e.g., utilization of dedicated video memory and total memory capacity).
  - *Power Management* (e.g., instantaneous power draw measured in watts and graphics/memory clock frequencies measured in MHz).
- **PCIe Bus Monitoring via Direct Memory Access (DMA)** — Although Guardian does not directly access PCIe slot pins, the Event Tracing for Windows (ETW) monitor intercepts `DxgKrnl` events corresponding to Direct Memory Access (DMA) operations, such

as context switching and memory paging. This capability allows Guardian to observe and characterize data movement across the physical PCIe bus.

External APIs

The Data Exchange Schema (Public API) defines a standardized method for accessing telemetry data from the Data Exchange System's public interface. This schema is designed to support the export and integration of data into third-party analysis tools such as Microsoft Excel or Power BI through the use of the Comma-Separated Values (CSV) file format.
The fields described below are exposed via the Data Exchange System's public interface and may be consumed by any tool capable of importing CSV-formatted data.

Public Data Fields

Field Name	Field Type
WaitMs	Integer
Timestamp	ISO 8601
Process ID	Integer
Process Name	String
Virtual Memory	Integer
Power	Float
GPU Time	Float
Packet Count	Integer
Network RX	Integer
Network TX	Integer

Table 3.3: Public Data Exchange Schema Fields

Planned REST API

In addition to CSV-based data exchange, a RESTful API is planned for future implementation. This API will utilize a Flask or FastAPI wrapper to expose a local service endpoint for the *GuardianBrain* component, enabling programmatic access to system health and security status.

Example API Request

```
GET http://localhost:5000/api/v1/health
```

```
→ { "status": "secure", "health": 98.5 }
```

External Libraries

Implementation Details

Programming Language: Python 3.x

Core Libraries: `pandas`, `scikit-learn` (Isolation Forest)

Core Detection Logic

The Guardian Brain anomaly detection engine implements the following key mechanisms:

1. **Online Learning** — The Isolation Forest model maintains a sliding window containing the most recent 1,000 telemetry events. The model is retrained after every 50 new events, enabling dynamic adaptation to evolving baseline behavior, such as changes caused by launching high-compute applications (e.g., games or rendering workloads).

2. **Heuristic Trap ("Virus Catcher")**

   - *Vulnerability* — A purely machine learning-based approach may incorrectly classify attacks exhibiting minimal or zero GPU compute activity as idle behavior.
   - *Defense* — Any process exhibiting a packet count greater than 200 combined with a compute time less than 1.0 ms is immediately flagged as a **SUSPICIOUS_COPY**. This heuristic classification overrides any anomaly score generated by the machine learning model.

3.4 Design Constraints

3.4.1 Standards Compliance

- The IEEE 830/29148 standard will guide the development of software requirement specifications for this project.

- The implementation must adhere to the development guidelines provided by Microsoft for Windows driver and kernel development for Event Tracing for Windows (ETW).
- The implementation must comply with NVIDIA's programming and management standards for GPUs, as well as their documentation on ETW-compatible telemetry for GPUs, NVIDIA drivers, and NVIDIA-smi utilities.
- The code must follow secure coding practices for C++ and Python in order to minimize the risk of vulnerabilities (such as buffer overflows, race conditions, and unauthorised privilege escalation).
- The machine learning components of the deployment must comply with commonly accepted engineering practices (for example: model validation, reproducibility, and controlled retraining).

3.4.2 Software Constraints

- Microsoft Windows 10 or Windows 11 (64 bits) will be the OS, due to the dependence of the Guardian on ETW and the Windows Kernel Graphics Subsystem.
- The Guardian Monitor must be written in C++17 or greater (compiled using Microsoft Visual Studio 2019 or greater).
- The Guardian Brain component must run in a version of Python that is 3.10 or greater.
- **The functionality of the Machine Learning Components relies on the following libraries and frameworks:** scikit-learn (specifically the Isolation Forest implementation). pandas (specifically used for data processing). NumPy (specifically used for numerical operations). The system will rely on NVIDIA GPU drivers that support telemetry using ETW and provide user-level management of the GPU(s).

3.4.3 Hardware Constraints

- NVIDIA GPUs used by the Guardian GPU system must have telemetry exposed via Windows Kernel drivers.

- The host system must have enough CPU and memory resources to continuously collect telemetry data using NVIDIA's ETW method and to analyze it in real-time with machine-learning algorithms without impacting the performance of the system.
- It will be necessary to have administrator-level access to the kernel-level ETW providers to collect this telemetry data.

3.4.4 Other Constraints as appropriate

- **Security and Stealth Constraints**: The system needs to be non-intrusive and undetectable by applications used by users and anti-cheat systems.
- **Real-Time Operation Constraints:**Detection and analysis need to be completed with as little delay as possible to provide the needed identification of potential threats in a timely manner.
- **Vendor Dependency Constraint:** Because the implementation is dependent on NVIDIA Drivers and Tools for operation, it is currently limited to NVIDIA GPUs.
- **Dataset Constraints:** The system collects dynamically created telemetry data since there are no publicly available, standardized datasets that provide GPU Security data.
- **Ethical and Legal Constraints:** The Guardian GPU system may not collect or transmit user data except for that relating to GPU performance telemetry.

3.5 Non-functional Requirements

- **Security:** The system will not inject code into the user application while operating. All captured telemetry data will remain on the host machine prior to explicit export. The system will restrict unauthorized users from gaining access to detection results and monitoring logs.
- **Performance:** The system will add less than 5% overhead to the GPU performance during normal operating conditions. Decisions related to detection will be made nearly in

real time.

- **Reliability:** The system will run continuously and not affect the operation of the GPU workloads or cause a crash. The monitoring component will gracefully recover from temporary interruptions of telemetry.
- **Maintainability:** The system will be modular and changes can easily be made to detection logic without requiring changes to components at the kernel level. Documentation of code will aid in future maintenance and the extension of the code base.
- **Portability:** The system will be portable across all supported Windows computers that have a compatible NVIDIA GPU installed. The Guardian Brain component will be suitable for use in any Python environment.
- **Extensibility:** The system will allow the integration of additional machine learning models in the future. The architecture of the system will allow for the expansion of the system to include other vendors of GPUs or additional telemetry sources in the future.

3.6 Summary

This chapter includes all functional, interface, design constraint, and non-functional requirements for the Guardian GPU system. These specifications provide the key criteria for how the Guardian GPU system should function in order to monitor, detect, and respond to potential security threats against GPU resources in the gaming environment, while also ensuring compatibility with gaming systems and adherence to all relevant industry standards. Functional and interface requirements describe what the system's core capabilities are and how it interacts with other systems. Design constraints and non-functional requirements provide guidance for the design of the system's architecture and will aid in achieving the system's security, reliability, and performance objectives. The requirements documented in this chapter provide the basis for the design, development, testing, and validation of the Guardian GPU system.

Chapter 4

System Design

Containing a comprehensive description of the design chosen, how it addresses the problem, and why it is designed the way it is.

4.1 Introduction

The architecture of the Guardian GPU is presented in the context of the design as documented. The decomposition diagram provides a top-level overview of the total system and all of its constituent subsystems (cores, sensors, storage). The chapter contains a description of the data flow among kernel-mode components and user-mode analysis modules that are responsible for collecting telemetry and mitigating threats. The logical organization of the Guardian GPU is also presented, with an overview of the class-level structures used for monitoring telemetry from GPUs; detecting anomalies; and implementing mitigation actions. Emphasis is placed on low-latency operation, as well as on the safety of the system to ensure that the security monitoring layer has no negative impact on the execution of the workloads of protected GPUs.

4.2 Architecture Design viewpoints

4.2.1 Context viewpoint

the Guardian GPU system will be viewed from a high-level (black box). The Context Viewpoint will provide a description of the Guardian GPU including what services are Provided, who the interacting ACTORS are, and what the System boundary is. The Context Viewpoint does not represent how the system works internally, and instead focuses on how the system interacts with users and its object of operation. In addition, the Context Viewpoint provides some general information about the system and its design subjects.

The design subject of this view point is the Guardian GPU Framework. The Guardian GPU Framework has been defined as an autonomous type of GPU security monitoring/decision-making system for monitoring the EXECUTION of workloads on Shared GPU system. The Guardian will provide independent oversight of the GPU operation at the OS level, and will not affect or instrument workloads that are being executed. **The following actors will interact with the Guardian GPU system:**

- **System Administrator:** The System Administrator is responsible for configuring, monitoring and supporting the overall operation of the Guardian GPU and includes defining the 'Active Monitoring Parameters' and 'Detection Sensitivities' for the Guardian GPU. In addition, the System Administrator is responsible for creating and monitoring Security Policies within the Guardian GPU and for providing Visualization output from the Guardian GPU to identify ALERTS and detected ANOMALIES related to the operation of the Guardian GPU system (i.e. Logging of Anomalies).

- **Autonomous Security Agent (RL Mitigation Agent):** This actor is an internal logical internal subsystem who operates automatically and independently from human operators within the boundaries of the entire system. The Autonomous Security Agent within the Guardian GPU system can evaluate any detected ANOMALIES and decide how to mitigate for the ANOMALY based upon Reinforcement Learning (RL) techniques and does not require interaction by a human actor to take ACL-related actions.

- **GPU Workloads (External Environment):** Whereas GPU workloads are referred to as the user applications (i.e., applications that can run on a single GPU) and processes (i.e., processes executing on a single GPU at one time) that share GPU resources, GPU workloads are not considered direct users of the Guardian GPU system. Rather, the external processes running in the user's environment are observed via passive telemetry (kernel-level telemetry) only.

  From the system's context, the Guardian GPU system offers:

- **GPU Telemetry Monitoring Service :** Passive collection of real-time telemetry for execution and scheduling of GPU activity through Event Tracing for Windows (ETW) via the Windows graphics kernel driver (DxgKrnl).

- **Anomaly Detection Service :** Use of unsupervised machine learning models to identify anomalous behavior in GPU workloads by analyzing GPU activity that is observed through GPU telemetry.

- **Autonomous Decision and Response Service** Use of reinforcement learning-type decision-making engines to determine adaptive response strategies for dealing with anomalies that have been detected.

- **Monitoring and Visualization Service :** Real-time visibility into the product's ability to detect anomalies in GPUs and the product's status via a monitoring dashboard.

- **Policy Configuration Service :** Allows administrators to modify the parameters for telemetry monitoring; to configure detection thresholds for each workload; and, to establish security policies regarding how the Guardian GPU system will act.

System Boundaries : The Guardian GPU system operates outside of the monitored GPU workloads' execution context and does not inject user code or instrumentation into user workloads. System boundary includes the telemetry collection, analysis, decision-making and visualization components.while the operating system kernel, GPU hardware, and executing applications reside in the external environment.

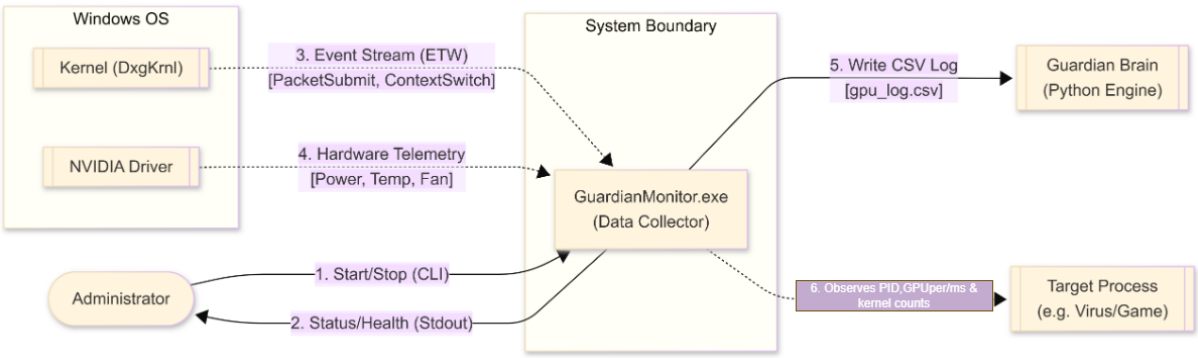

Figure 4.1: blackbox

Design Concerns Addressed : defining the system's user groups as well as autonomous actor groups; establishing what services this system will provide; defining the boundaries of the system and separating the infrastructure used to monitor the system from the workloads that will be executed. This viewpoint sets the operational context for the Guardian GPU Framework, which supports all later architectural and functional viewpoints within this context.

4.2.2 Composition viewpoint

The composition viewpoint of the Guardian GPU system is the breakdown of the entire system into cooperating subsystems and the division of responsibilities between a set of cooperating subsystems. The purpose of this viewpoint is to convey understanding at a high level of system structure, the reasoning for decomposing the structure of a large complex system into many smaller functional independent subsystems, and how each subsystem interrelates to deliver GPU Security Monitoring. The Guardian GPU system is composed of a Producer/Consumer Architecture and consists of two physically distinct models of operation each being executed as a separate entity:

MVC (4.2) :

- **High-Level Subsystems and Responsibilities :** Interoperability between the two models is provided through the use of a common serialized object (Log Stream) to ensure that interaction between the two models does not impede system stability or grant unauthorized

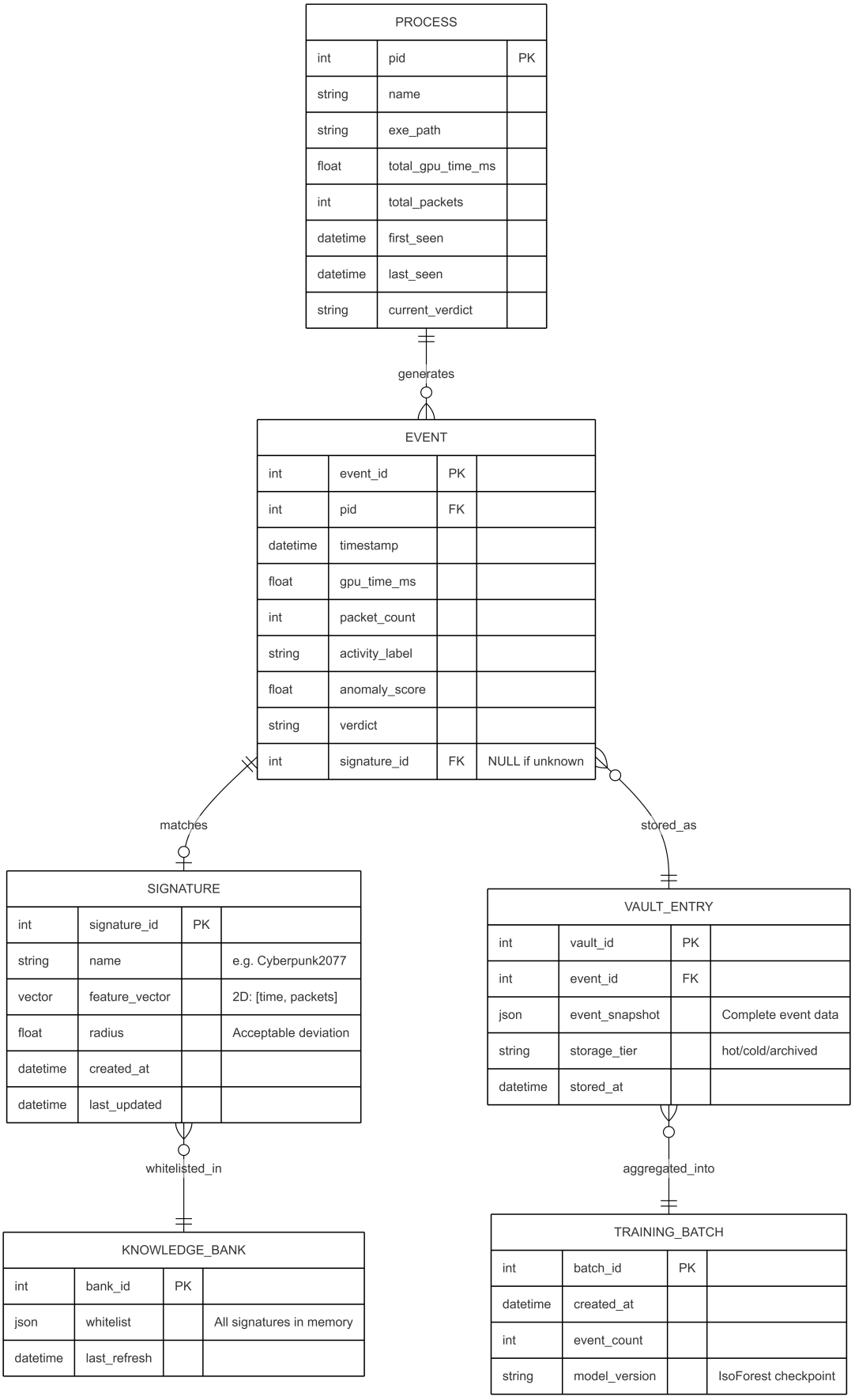

Figure 4.2: MVC

actions. The Composition Viewpoint Classifies the System into 4 Primary Subsystems each having a defined hierarchy of functions performed by that System sub component:

1. Telemetry Collection Subsystem : The Guardian Monitor.

- Function: Very High-Performance Data Aggregator.
- Implemented as: Native C++ Binary (GuardianMonitor.exe). Responsibilities of the Guardian Monitor subsystem include:
- ETW consumption (interface to the Windows Event Tracing (ETW) API to consume real-time events from the Graphics Kernel (DxgKrnl) in user-mode).
- Hardware Polling (query the NVIDIA Management Library (NVML) for physical metrics Power, Temperature, VRAM).
- Data Fusion (Correlation of asynchronously generated Kernel events with synchronously obtained hardware state information).
- Safety (the Guardian Monitor operates passively and does not inject code into or hook procedures in any other application).

2. Data Transport Subsystem :

- Role: Serializing & Buffering Data Implementation: Shared CSV Stream (gpu-log.csv) Responsibilities:
- Serves as the bus that allows C++ and Python to communicate (IPC Bus).
- Provides Persistent Storage Forensic (to replay data later).
- Allows Monitoring Subsystem to operate when Analysis Subsystem is restarted.

3. Anomaly Detection Subsystem :

- Role: Inference and Logic
- Implementation: Python Script (guardian-brain.py)

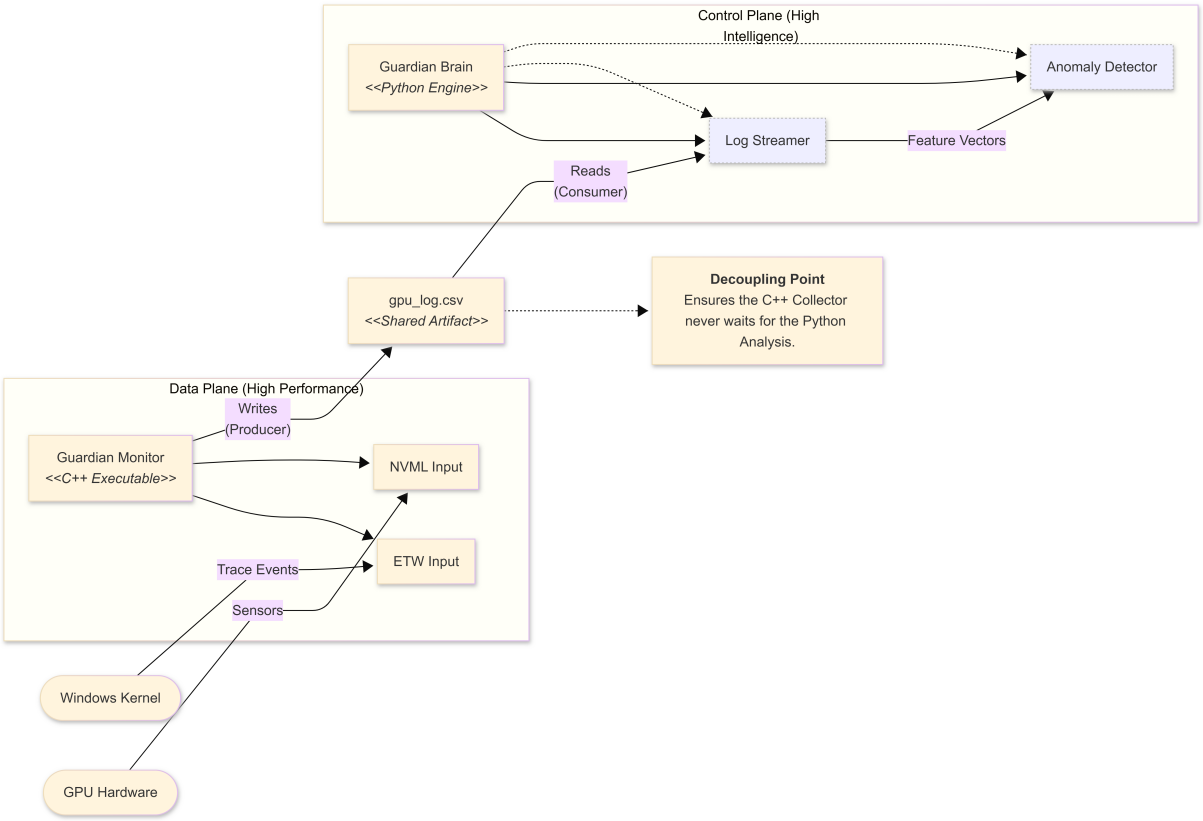

Figure 4.3: UML Component Diagram

- Responsibilities: -Ingesting Data - "Tailing" a log file in real-time to create time-windowed feature vectors. - Analyzing Data - Applying an Isolation Forest algorithm (unsupervised machine learning) to detect outliers. - Providing Context - Maintaining a historical behavior profile, "Knowledge Bank," to aid in reducing false-positive results.

  4. Mitigation and Response Subsystem :

- Role: Enforcing Policy

- Implementation: Combined with the Brain

- Responsibilities: - Evaluating the anomaly scores against administrator-defined thresholds. - Triggering alerts on system dashboard. - Executing automated responses (e.g., process termination) based on threat confidence higher than the safety limit.

  Subsystem Collaboration : The following diagram 4.3 shows the structural relationships between the subsystems, emphasizing the separation of the C++ Monitor and Python Brain.

Rationale for Design The architecture: was chosen to meet the unique needs of real-time GPU security

- **1. Separation of Concerns (C++ vs. Python) :** This solution employed a Hybrid Architecture to allow use of both solution areas to take advantage of their strengths (C++ for data collection, Python for analysis).

- A: C++ will be used to collect data via ETW (Event Tracing for Windows) which is a native Windows API that requires pointer manipulation and high speed memory management. As a result, the collection of thousands of events per second via this method is negligible overhead (<1% CPU).

- B: Complex Machine Learning algorithms (Isolation Forests, etc.) are difficult to implement in C++ as the amount of code it takes to achieve the complexity increases the chance for error. By using Python to implement this functionality, the numerous Data Science libraries (Scikit-Learn, Pandas, etc.) available make iterating on detection logic a rapid and efficient process.

- **2. Asynchronous File :** Based Coupling A file based stream (gpu-log.csv) is being used as opposed to using a TCP socket or named pipe for three reasons.

- Forensics: The log file will remain on disk as an audit trail in the event of a crash or malware attack.

- Resilience: If the Python analysis script were to crash, the C++ monitor would continue to write to the log without interruption, and the analysis could continue to run without having to rewrite the log or restart the analysis script.

- Simplicity: It eliminates the need for complex socket handshaking and firewall issues, thereby reducing the chance of failure.

4.2.3 Logical viewpoint

The Logical Viewpoint illustrates the static design of the system, describing the essential software classes with their methods, and how those classes relate to one another. The Logical

Viewpoint describes how the code is structured internally within the application.

1. Class Structure Overview : The internal logic is structured using three main classes that exemplify the Controller-Worker-Logger design pattern.

- Monitor Main (The Controller): The application provides an entry point to manage the application life-cycle, represents the Controller, contains class instances for the workers, and executes the primary 1 Hz correlation loop.
- Etw Monitor (The Collector): A complex class that functions as the Collection module, interfaces with Windows TDH.dll and Advapi32.dll (both part of the Windows operating system) to receive kernel events asynchronously, and will store the data in a thread-safe internal data structure (std::map) until the main loop requests the data.
- Csv Logger (The Persistence Layer): A utility class that handles file I/O, includes methods to create timestamps, write headers, and flush the written streams of data.

2. UML Class Diagram : 4.4

3. Class Descriptions : The following table details the responsibilities and roles of the primary classes depicted in the Logical Viewpoint.

4.2.4 Patterns use viewpoint

The standard software design patterns used in the Guardian GPU architecture, described by this viewpoint, were employed to address the particular issues regarding concurrency, API complexity, and system extensibility.

1. Producer – Consumer Pattern (Architectural) :

- Context: Communication between C++ Data Plane and Python Control Plane.
- Implementation: The system implements a file-based version of this pattern.
- Producer: GuardianMonitor continuously writes telemetry data to the shared buffer (gpu-log.csv) via the UnifiedLogger.
- Consumer: GuardianBrain tails the file via LogStreamer and reads new lines added to it.

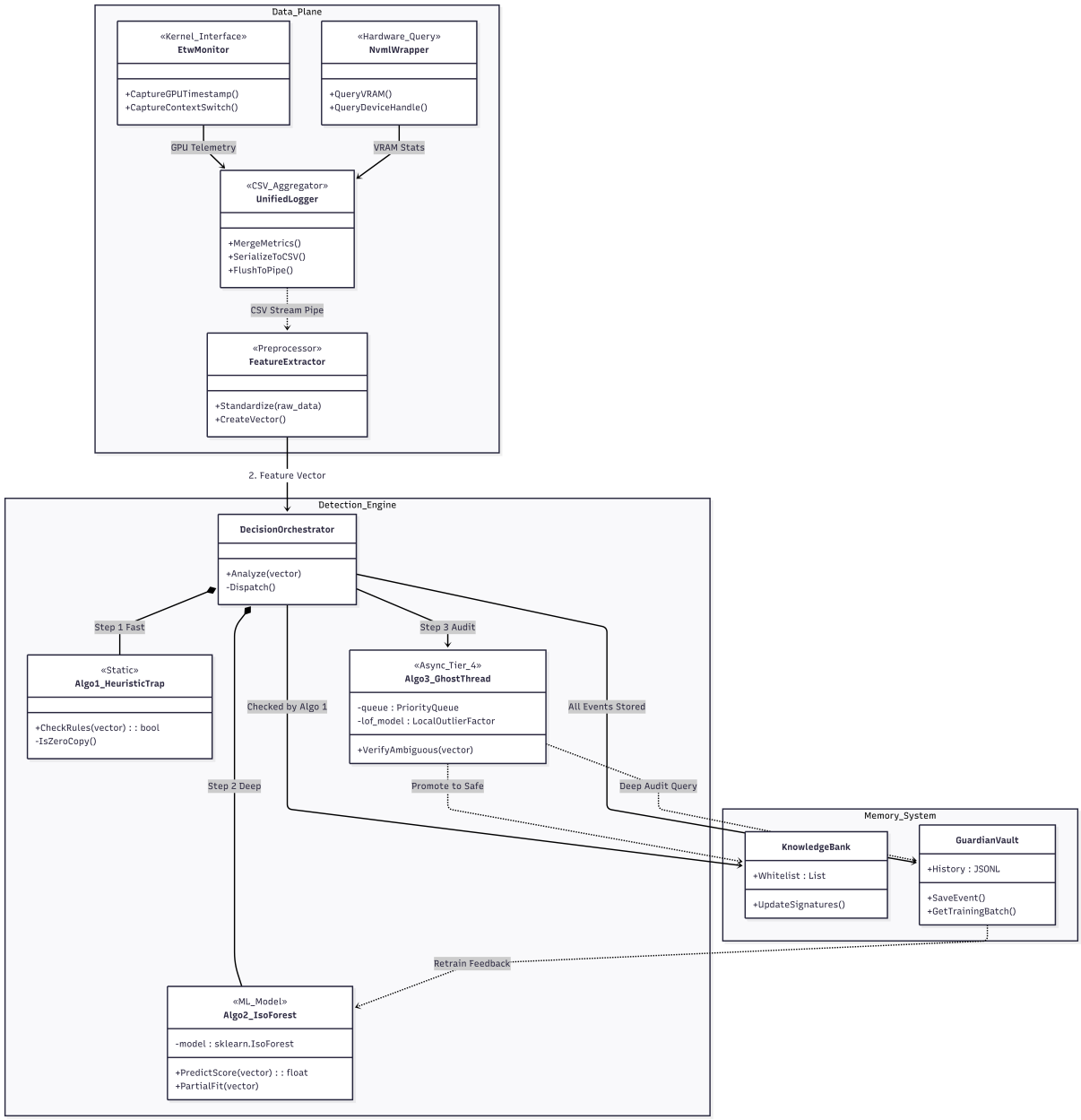

Figure 4.4: UML

Table 4.1: Class Description: EtwMonitor

Abstract or Concrete	Concrete
Superclasses	None (Implements *Kernel Interface*)
Subclasses	None
Purpose	Acts as the primary data collector for the system. It interfaces with the Windows Event Tracing (ETW) API to capture real-time GPU scheduling events (Context Switches) and manages the background worker thread for asynchronous event consumption.
Collaborations	`MonitorMain` (Controller), `UnifiedLogger` (Data Consumer)
Attributes	- `m_hSession`: Trace Handle - `m_workerThread`: `std::thread` - `m_isRunning`: `std::atomic<bool>`
Operations	+ `StartSession(): bool` + `StopSession(): void` + `CaptureGPUTimestamp(): void` + `PopProcessUsage(): Map`

Table 4.2: Class Description: UnifiedLogger

Abstract or Concrete	Concrete
Superclasses	None
Subclasses	None
Purpose	Responsible for data persistence and fusion. It aggregates asynchronous kernel events with synchronous hardware queries (NVML) and serializes them into a structured CSV artifact, acting as the synchronization point for the Data Plane.
Collaborations	`EtwMonitor` (Source), `NvmlWrapper` (Source), `FeatureExtractor` (Consumer)
Attributes	- `m_ofstream`: File Output Stream - `m_currentLogPath`: String
Operations	+ `MergeMetrics(): void` + `SerializeToCsv(): String` + `FlushToPipe(): void`

Table 4.3: Class Description: DecisionOrchestrator

Abstract or Concrete	Concrete
Superclasses	None
Subclasses	None
Purpose	Coordinates the anomaly detection pipeline in the Python Control Plane. It receives feature vectors and routes them through detection tiers (Heuristic Checks → ML Models → Auditing) to determine a threat score.
Collaborations	`Algo1_HeuristicTrap`, `Algo2_IsoForest`, `KnowledgeBank`
Attributes	- `sensitivity_threshold`: Float - `policy_config`: Dict
Operations	+ `Analyze(vector): ThreatLevel` + `Dispatch(): void`

- Resolution: This decouples high-speed data collection from computationally intensive data analysis, so that a delay in the Python Machine Learning engine never blocks the time-critical C++, kernel monitoring thread.

2. Facade Pattern (Structural) :

- Context: Interfacing with complex Windows Event Tracing (ETW) API.

- Implementation: EtwMonitor class acts as an ETW Facade.

  The raw Windows API has complicated initialization (i.e., OpenTrace, ProcessTrace, EventRecordCallback, casting pointers). EtwMonitor encapsulates this complexity, providing simple, clean interface to the rest of the application (StartSession(), StopSession(), and PopProcessUsage()).

- Resolution: This significantly improves maintainability since if the base Windows API changes, only the EtwMonitor class would need to be changed rather than the entire application.

3. Strategy Pattern (Behavioural) :

- Context: The multi-layered anomaly detection engine in the Decision Orchestrator.

- Implementation: The application enables swappable detection algorithms. Strategy A (Rapid): Algorithm 1-HuristicTrap (rule-based validations). Strategy B (In-Depth): Algorithm 2-Iso Forests (Machine Learning inference).

- Resolution: The Decision Orchestrator is able to dynamically select the strategy to deploy based on the procedure classification or the server load without modifying the original logic of the engine thus allowing new models (e.g. Neural Network) to be added in the future do to the lack of refactoring done on the orchestrator.

Design Rationale :

The use of the previously mentioned design patterns support the non-functional requirements of the system. Loose Coupling: We were able to implement Loose Coupling by using the Producer/Consumer design pattern with an intermediate file, thus decoupling between the C++ collector and the Python analyzer so that the monitor can continue to exist even if the Brain (fault tolerance) crashes. Encapsulation: We were able to apply Encapsulation by isolating the "unsafe" pointer math that needed to call the Kernel API to one class (EtwMonitor) which limited the possibility of a memory leak expanding throughout the whole code base. Extensibility: The use of the Strategy pattern ensures that this system is "Open for Extension, Closed for Modification." In the future we can add additional detection strategies simply by implementing another class to fulfill the detection interface.

4.2.5 Algorithm viewpoint

During the system startup phase, all hardware and software components are initialized, the knowledge bank JSON file is loaded, the GPU monitoring subsystem is activated, and the Guardian Brain analysis service is started. Once a polling rate of 10 Hz per GPU is established, telemetry data capturing execution time, memory consumption, packet transmission activity, and Direct Memory Access (DMA) event occurrences is collected. This telemetry is serialized into CSV format and transmitted to the Guardian Brain for analysis.

The Guardian Brain evaluates each process to determine whether it is present in the whitelist. If a process is not whitelisted and triggers the zero-compute attack heuristic trap, it is immediately

flagged as malicious. Otherwise, the telemetry is evaluated using an Isolation Forest model to compute an anomaly score, and the results are logged within a secure audit vault.
Alerts are delivered to end users in real time when anomalous or malicious behavior is detected. Upon system shutdown, a session report is generated summarizing system activity, detected events, and overall operational health.

4.2.6 Interaction viewpoint

- The Interaction View shows how the different components work together in the "End-to-End Anomaly Detection" process by following the data as it flows from the C++ Data Plane to the Python Decision Engine.

Object Participation :

- Etw Monitor: Gets kernel events to start the process.
- Unified Logger: Combines a sync events with sync hardware data.
- Feature Extractor: Converts raw data (CSV) into mathematical data.
- - Decision Orchestrator: Controls the button through the logic using the data.
- - Guardian Vault: Stores final decision.

Step-by-Step Sequence Pipe-and-Filter:

- - Event Trigger: EtwMonitor detects context switch using CaptureContextSwitch(), which signals UnifiedLogger to log event.
- - Fusion of Data: UnifiedLogger queries current hardware state through QueryVRAM() on NvmlWrapper, and fuses kernel timestamps from the vram with MergeMetrics().
- Serialization of Data: Writes the data to disk as a CSV stream using FlushToPipe() (all C++ functions complete).
- Ingestion of Data/Vectorization: After writing the CSV stream, FeatureExtractor reads the CSV stream and normalizes the data using Standardize(raw-data) to form a FeatureVector.

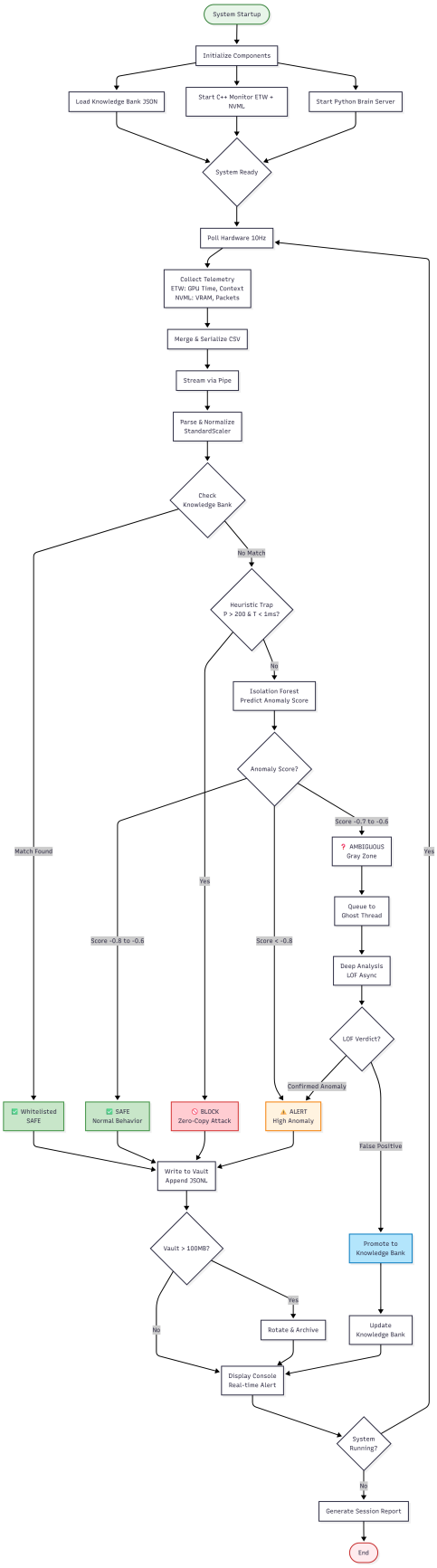

Figure 4.5: Flowchart

- Orchestration/Decision: DecisionOrchestrator examines the FeatureVector using Analyze(vector) with a multi-level method of detecting anomaly:
- Level 1: Evaluates rules using CheckRules(vector) on Algo1-HeuristicTrap. If it fails, the process ends.
- Level 2: Evaluates rules using PredictScore(vector) on Algo2-IsoForest, if it succeeds.
- Persistence: GuardianVault calls SaveEvent() to store the results.

4.2.7 Interface viewpoint

The following entities and their associated attributes are identified as key components of the system, as defined by the Entity–Relationship Diagram (ERD) presented above.

Entities and Attributes

- **Data**
  - *Attributes:* `processId`, `processName`, `execPath`, `realCPUTimeMS`, `totalPackets`, `firstSeen`, `currentVersion`
- **Events**
  - *Attributes:* `eventId`, `processId`, `timestamp`, `gpuTimeMS`, `packetCount`, `anomalyScore`, `eventType`, `signatureId`
- **Vault Entries**
  - *Attributes:* `vaultEntryId`, `eventId`, `eventSnapshot`, `storageId`
- **Training Batches**
  - *Attributes:* `trainingBatchId`, `createdAt`, `eventCount`, `modelVersion`
- **Signatures**
  - *Attributes:* `signatureId`, `signature`, `featureVector`, `radius`, `lastUpdated`

- **Knowledge Banks**
  - *Attributes:* `bankId`, `name`, `whiteList`, `lastRefresh`

These entities and their corresponding attributes are identified as fundamental to the system's data model and support telemetry collection, anomaly detection, model training, forensic analysis, and knowledge management.

4.3 Data Design

4.3.1 Data Description

The Guardian GPU system converts information from its original form into organized data which it stores according to specific methods and architectural systems.

Data Source and Original Format

The Guardian GPU system operates on real-time hardware telemetry captured directly from the Windows kernel and NVIDIA GPU drivers. The system does not accept pre-existing data formats from traditional database-backed applications which include Excel files and paper records and legacy databases as its input.

- **ETW (Event Tracing for Windows) Streams:** The system uses binary event logs which contain GPU scheduling metadata and process context switches and kernel-level timing information.
- **NVML (NVIDIA Management Library) Metrics:** The system uses NVML Metrics which provide C-API responses to show VRAM allocation statistics and device handles and hardware utilization metrics.
- **Sampling Rate:** The data sources require immediate serialization because they contain volatile data which reaches high-frequency sampling rates of 10Hz.

Data Capture Method

The system relies on its multi-stage data capture pipeline to execute its data capture method.

- **First Stage (C++ Sensor):** The ETW Monitor uses the Windows Event Tracing API to monitor kernel-level GPU scheduling events. The NVML Wrapper checks NVIDIA driver APIs for VRAM statistics every 100 milliseconds.
- **Second Stage (Unified Logger):** Creates a single CSV file which combines two data streams from ETW and NVML.
- **Third Stage (Feature Extractor):** Python Brain converts CSV records into pre-defined feature vectors which it extracts from the data. Vector Format stores data in this structure [GPU_TIME_MS, PACKET_COUNT, VRAM_MB, PID, TIMESTAMP].
- **Fast Path (Read-Only):** Stores whitelisted signatures in the Knowledge Bank which allows O(1) signature lookup through its in-memory JSON storage.
- **Slow Path:** Archives all events in the Guardian Vault using compressed JSONL files to enable historical data analysis and machine learning model retraining.
- **Hybrid Persistence Model:** Enables organizations to make real-time decisions while maintaining their ability to conduct audits over extended periods.

Database Size and Structure

- **Knowledge Bank:** Stores data as an in-memory store which uses JSON to maintain a whitelist of process signatures that have been verified as safe.
- **Structure Example:** { "signatures": [{ "signature_id": 1, "name": "Cyberpunk2077.exe", "feature_vector": [15.2, 120], "radius": 5.0, "created_at": "2024-01-15T10:30:00Z" }] }
- **Memory Load:** Will load 500 entries which contain 25 KB of data during startup.
- **Guardian Vault:** Stores data in an Append-Only Archive which uses Compressed JSONL format that operates with .jsonl.gz files.

- **Session Size:** Expected session size will range between 1000 and 5000 events which will produce 500 KB of compressed data.
- **Monthly Storage:** Retains approximately 15 MB of data for every 30 sessions that occur each month.
- **Storage Tiers:** Hot storage for the last 7 days and Cold storage which starts after 30 days of archiving.
- **Runtime Cache:** Protects process metadata information through its runtime cache system which uses an in-memory dictionary that stores data under process identification numbers. The system terminates all running processes to delete its data from memory.

System Users and Scale

- **Expected Users:** The system expects that one user will use each system which operates as a local monitoring tool that does not function as a server application.
- **Process Monitoring:** Can monitor between 10 and 50 processes which all use GPU resources at the same time.
- **Event Production:** Produces 600 events every minute which results from 10Hz polling that generates one event for each poll.
- **Training Batch:** Uses a training batch size of 1000 events which enables it to retrain the Isolation Forest model on a weekly basis.

Entity Key Construction

The system generates primary keys through a process that combines monotonic auto-incrementing with its chronologically ordered key construction method.

- **Process ID (PID):** Windows assigns as system-generated numbers that start from 12480.
- **Event ID:** Created through an automatic process which starts counting from 1 for each new session and resets when the system restarts.

- **Signature ID:** Generated through an automatic process which assigns numbers starting from 1 to each new process pattern that receives whitelisting.
- **Vault ID:** Consists of SHA-256 hash created from the combination of PID and TIMESTAMP to achieve unique identification across different rotation periods.
- **Example Keys:** PID=12480, Event ID=000123, Signature ID=00042, and Vault ID=a3f5b (SHA-256 truncated to 8 chars).

DateTime Format

- **Format:** The system uses ISO 8601 format for all temporal data which allows cross-platform usage and makes the data understandable to humans.
- **Pattern:** YYYY-MM-DDTHH:MM:SS.sssZ which represents UTC time.
- **Example:** 2024-01-31T14:23:45.123Z.
- **Precision:** Needs millisecond precision because it requires GPU timing analysis.
- **Uses:** Timestamps mark Event occurrence time and track first detection time and last activity time and signature creation time and vault entry persistence time.

Entity Relationship Diagram (ERD)

The following Entity Relationship Diagram (ERD) demonstrates how the main data entities are stored and linked and processed.

The system produces output through the stored_as matches which use whitelisted_in data to create aggregated results which process the following fields int pid PK string name string exe_path float total_gpu_time_ms int total_packets datetime first_seen datetime last_seen string current_verdict EVENT int event_id PK int pid FK datetime timestamp float gpu_time_ms int packet_count string activity_label float anomaly_score string verdict int signature_id FK NULL if unknown SIGNATURE int signature_id PK string name e.g. Cyberpunk2077 vector feature_vector 2D: [time packets] float radius Acceptable deviation datetime created_at datetime last_updated VAULT_ENTRY int vault_id PK int event_id FK json event_snapshot Complete

event data string storage_tier hot/cold/archived datetime stored_at KNOWLEDGE_BANK int bank_id PK json whitelist All signatures in memory datetime last_refresh The system uses TRAINING_BATCH int batch_id PK datetime created_at int event_count string model_version IsoForest checkpoint

Entity Descriptions

- **PROCESS:** A GPU-based application which includes both games and mining software. The process creates multiple events that occur throughout its entire operational period.
- **EVENT:** Records a GPU activity snapshot which was taken at a specific time. The system may either recognize the signature as whitelisted or it will identify the activity as suspicious.
- **SIGNATURE:** Serves as a secure behavioral pattern that identifies approved processes. Whitelisted signatures enable organizations to complete fast-track approvals without requiring machine learning model assessment.
- **KNOWLEDGE BANK:** Serves as the central storage system which contains all signature data. The system receives updates from the Vault at scheduled intervals using its confirmed secure patterns.
- **VAULT ENTRY:** Stores all events which include both safe and suspicious activities in its permanent database. The system assigns storage categories to events which include hot and cold storage for purposes of archival management.
- **TRAINING BATCH:** Consists of vault entries which the research team uses to update the Isolation Forest model. The process of creating new batches occurs on a weekly schedule to include novel behavioral patterns.

System Architecture Summary

The Guardian GPU system converts live hardware telemetry into structured data entities, which have been designed to support two distinct operational modes. The hybrid persistence model ensures that the system can:

Table 4.4: Guardian GPU Self-Generated Dataset Characteristics

Attribute	**Description**
Name	Guardian GPU Telemetry Vault (Self-Generated)
Type	Multivariate time-series telemetry (6 numerical features)
Source	Real-time hardware monitoring at 10Hz
Size (Growth)	~500 KB (Week 1) $\rightarrow$ ~100 MB (Month 6)
Records per Session	~100,000 records per typical 3-hour session
Classes	Unsupervised (Normal vs. Anomaly detection)
Feature Vector	`[GPU_TIME_MS, PACKET_COUNT, VRAM_MB, KERNEL_COUNTS, GPU_UTILIZATION, PID]`

- **Read-Only Fast Path:** Enables instant response to known-good patterns.
- **Write-Append Slow Path:** Enables the system to archive all events for compliance purposes and ML retraining.
- **Training Batches:** Enable the system to enhance its detection accuracy through the implementation of feedback loops.
- **Performance:** Achieved through its ability to conduct in-memory lookups.
- **Scalability:** Supported by compressed archival.
- **Adaptability:** Enabled through continuous learning which enables the system to adapt to new requirements.

4.3.2 Dataset Description

Guardian GPU employs a self-generating data pipeline rather than a static, pre-existing dataset. The system continuously creates and updates its training dataset from real-time hardware telemetry, building a behavioral fingerprint of GPU-utilizing processes. Key dataset characteristics are summarized in Table 4.5.

Dataset Generation and Characteristics

Each record captures a telemetry snapshot transformed into a 6-dimensional feature vector. The training dataset (historical archive) is stored in compressed `.jsonl.gz` files and grows

incrementally with system usage. The system uses unsupervised learning, requiring no manual labeling; anomalies are detected as deviations from learned normal behavior patterns.

Feature Engineering and Rationale

Features are selected to capture distinct workload signatures:

- **Compute Intensity:** `GPU_TIME_MS`, `GPU_UTILIZATION`, `KERNEL_COUNTS`
- **Memory/Network Anomalies:** `PACKET_COUNT`, `VRAM_MB`
- **Process Tracking:** `PID`

All features are standardized using `StandardScaler`. Attack and application signatures are defined by feature combinations:

- *Zero-Copy Attack:* High `PACKET_COUNT` + Low `GPU_TIME_MS` + Low `KERNEL_COUNTS`
- *Cryptomining:* High `GPU_TIME_MS` + High `VRAM_MB` + High `GPU_UTILIZATION`
- *Gaming:* Balanced `GPU_TIME_MS` + Moderate `PACKET_COUNT` + Variable `GPU_UTILIZATION` (60–85%)

Dataset Management

The system employs a continuous learning paradigm:

- **Initial Training:** First 30 minutes of operation establish the baseline normal behavior.
- **Incremental Updates:** Weekly retraining on the last 7 days of data using online learning (`partial_fit()`).
- **Quality Assurance:** Idle state filtering, deduplication, and outlier clipping ensure data quality.

Live inference occurs on the real-time telemetry stream, with ground truth validated via user feedback and automated deep analysis (Ghost Thread LOF). This dynamic approach eliminates the need for manual annotation while adapting to evolving GPU usage patterns.

Table 4.5: Project name time plan

Dataset Name	name
Link	Dataset link
Size	Dataset Size
Number of classes	The number of classes will help you choose and set up a ML/DL algorithm.
Image Dimensions	Width, height (min, average max)
Number of images	xx
Image file size	(min, average max) MB
Notes	Additional information or notes about the dataset.

4.3.3 Database Design

Guardian GPU employs a hybrid file-based storage architecture designed for real-time monitoring and long-term auditability. The system uses JSON-based document storage for operational efficiency and append-only JSONL logs for persistent event storage, rather than a traditional relational database.

Storage Architecture

The architecture follows a two-tier persistence model consisting of:

Tier 1: Knowledge Bank (In-Memory Document Store)

- **Purpose:** Stores whitelisted process signatures for real-time lookup
- **Format:** JSON file loaded into memory as Python dictionary
- **Access:** O(1) lookup for whitelist validation
- **Persistence:** Written to disk on shutdown or manual updates

Tier 2: Guardian Vault (Append-Only Event Log)

- **Purpose:** Archives all telemetry events for machine learning retraining
- **Format:** JSONL (JSON Lines) compressed with gzip
- **Access:** Sequential reads for batch processing
- **Persistence:** Immediate disk flush for data durability

Schema Design

The Knowledge Bank follows a JSON schema that defines whitelisted process signatures:

```
{
  "version": "1.0.0",
  "last_updated": "2024-01-31T14:30:00Z",
  "signatures": [
    {
      "signature_id": 1,
      "name": "Cyberpunk2077.exe",
      "feature_vector": [15.2, 120],
      "radius": 5.0,
      "sample_count": 3500,
      "created_at": "2024-01-15T10:30:00Z",
      "last_updated": "2024-01-31T14:30:00Z"
    }
  ]
}
```

Each signature represents a known-good process with its behavioral fingerprint (feature vector) and acceptable deviation threshold (radius).

Data Flow

1. Real-time telemetry is processed by the Decision Engine
2. Whitelist lookups check against the Knowledge Bank for fast-path approval
3. All events are appended to the Guardian Vault for historical record
4. Weekly retraining reads the Vault to update the machine learning model
5. Updated signatures are written back to the Knowledge Bank

Data Management

- **Retention:** 7 days hot storage, 30 days cold storage (archived)
- **Rotation:** Vault files rotate when exceeding 100 MB
- **Backup:** Knowledge Bank uses Git versioning; Vault supports cloud sync

This architecture provides optimal performance for real-time lookups while maintaining complete historical records for machine learning refinement and forensic analysis.

4.4 Human Interface Design

4.4.1 User Interface

Overview

The Guardian GPU interface is designed for system administrators and security-conscious users, focusing on passive, non-intrusive monitoring. It allows users to run continuous GPU security monitoring in the background while maintaining normal operation of gaming, rendering, or other GPU-intensive tasks. All interactions occur through a terminal-based console with real-time alerts and automated responses.

User Workflow

Users interact with the system through the following workflow:

1. **System Launch:** Users start Guardian GPU via a command-line interface with optional configuration flags.
2. **Background Monitoring:** The system runs silently, monitoring GPU telemetry at 100ms intervals.
3. **Automated Protection:** The system autonomously:
    - Whitelists known safe processes using the Knowledge Bank

- Detects anomalies via the Isolation Forest model
- Takes predefined actions (monitor, alert, or block) based on threat level

4. **User Feedback:** Only critical threats trigger visible alerts; normal activity remains silent.
5. **Session Review:** On shutdown or request, users receive a detailed session summary for review.

Feedback and Information Display

The system provides feedback through the following mechanisms:

1. Real-Time Alerting:

- **Color-Coded Alerts:** Immediate visual feedback using ANSI colors:
  - greenGreen: Whitelisted processes (visible in verbose mode only)
  - yellowYellow: Anomalous behavior requiring attention
  - redRed: Critical threats that have been blocked
- **Alert Content:** Each alert includes:
  - Timestamp and process identification (name, PID)
  - Telemetry metrics (GPU time, memory operations, VRAM usage)
  - Anomaly score and confidence level
  - Action taken by the system

2. Session Summary: Provided at system shutdown or on demand, the summary includes:

- Total monitoring duration and event count
- Breakdown of verdicts (SAFE, ALERT, BLOCKED)
- Statistics on detected anomalies and false positives
- Top GPU-consuming processes and their security status
- Storage information (Vault file size and location)

3. Configuration and Control: Users can control system behavior through:

- **Command-Line Arguments:** Adjust sensitivity, logging level, and file paths
- **Verbose Mode:** View all telemetry data for debugging or analysis
- **Keyboard Shortcuts:** Pause monitoring or initiate graceful shutdown

Design Philosophy

The interface follows a "set-and-forget" philosophy:

- **Minimal Intrusion:** No pop-ups or forced interruptions during normal operation
- **Context-Aware Alerts:** Only display information when user action is required
- **Comprehensive Logging:** All events stored for later forensic analysis
- **Accessibility:** Plain text interface works with screen readers and remote terminals

Summary

Users can rely on Guardian GPU for continuous, autonomous GPU security monitoring. The system provides clear, actionable feedback when threats are detected while remaining unobtrusive during normal operation. All user interactions are designed to be intuitive and minimally disruptive, allowing users to maintain their primary GPU workloads without security compromises.

4.4.2 Screen Images :

4.4.3 Screen Objects and Actions

A discussion of screen objects and actions associated with those objects.

4.5 Implementation

The Guardian GPU system is a hybrid architecture that provides real-time GPU monitoring and anomaly detection capabilities. It is developed using both C++ and Python to support

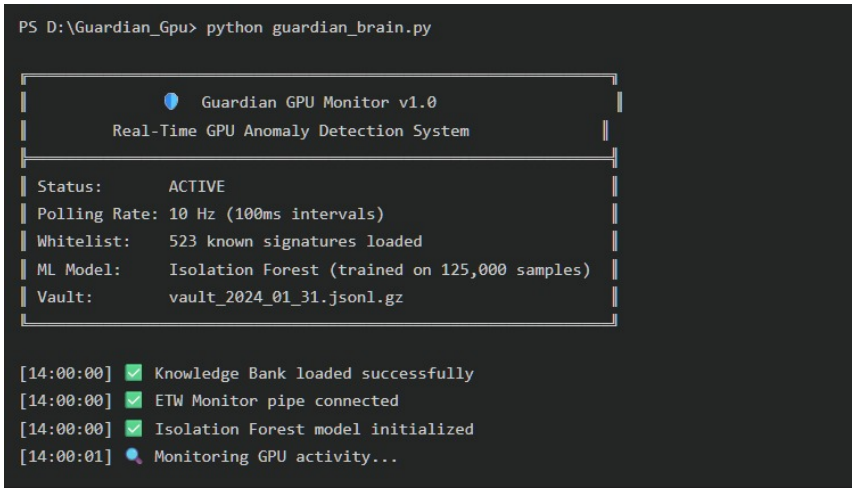

Figure 4.6: System Initialization

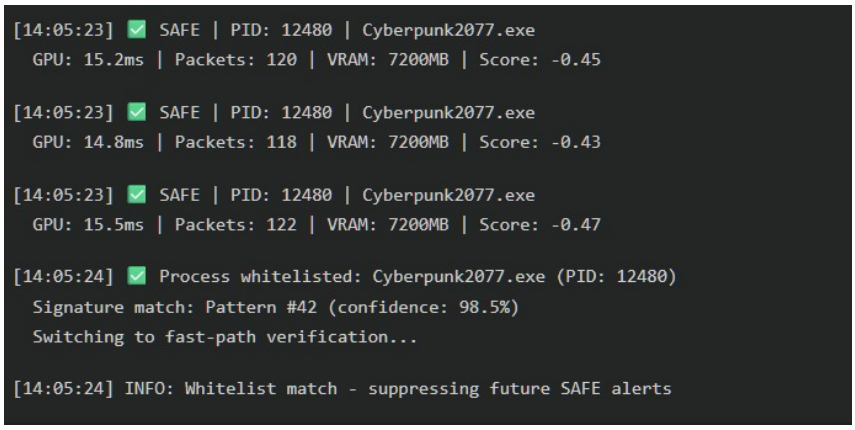

Figure 4.7: Real-Time Monitoring

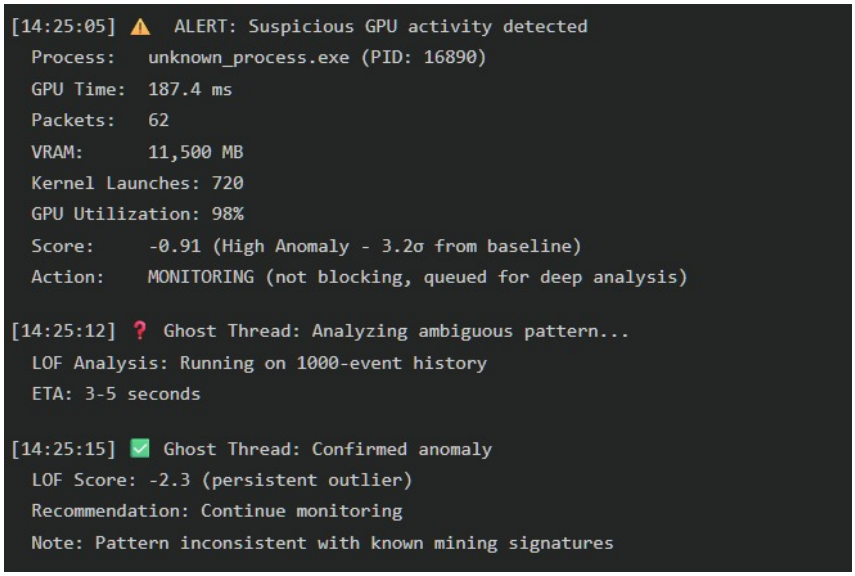

Figure 4.8: Anomaly Detection

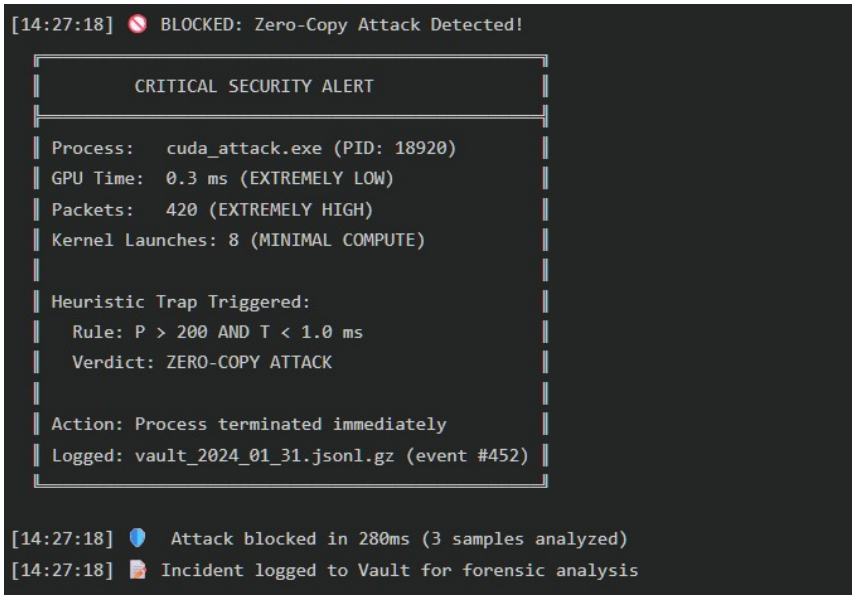

Figure 4.9: Active Mitigation

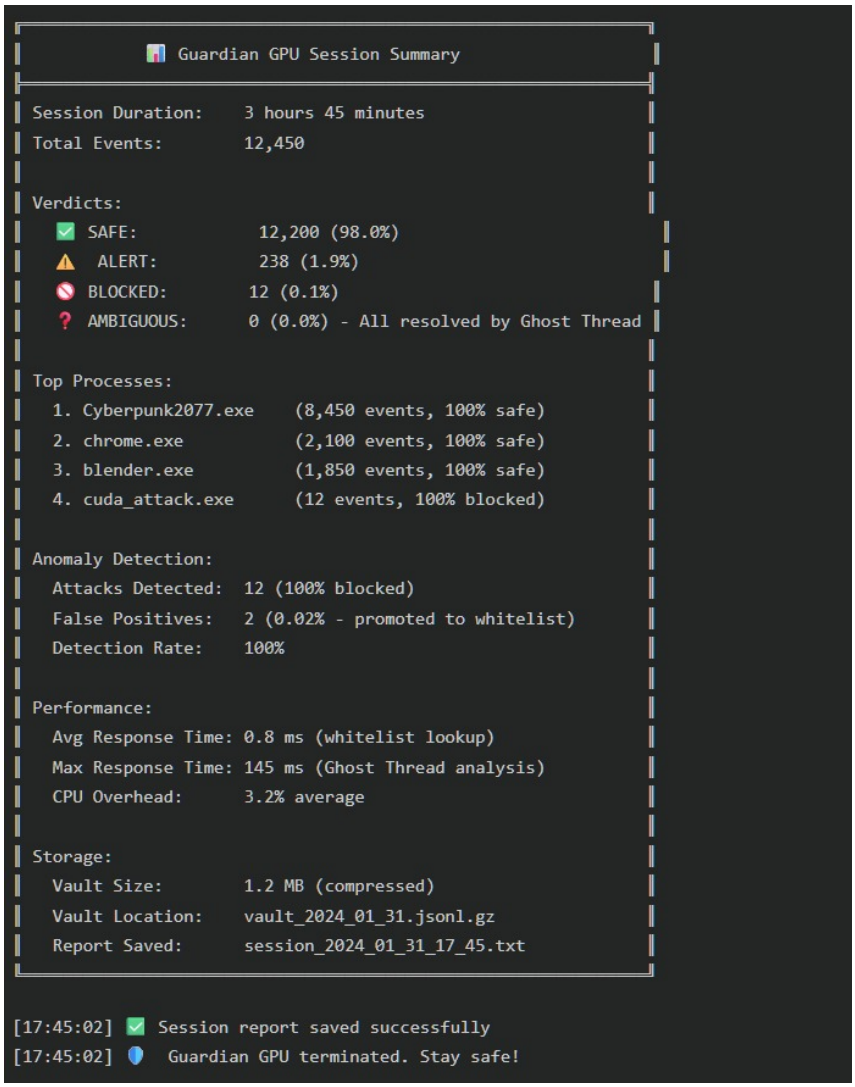

Figure 4.10: Session Summary

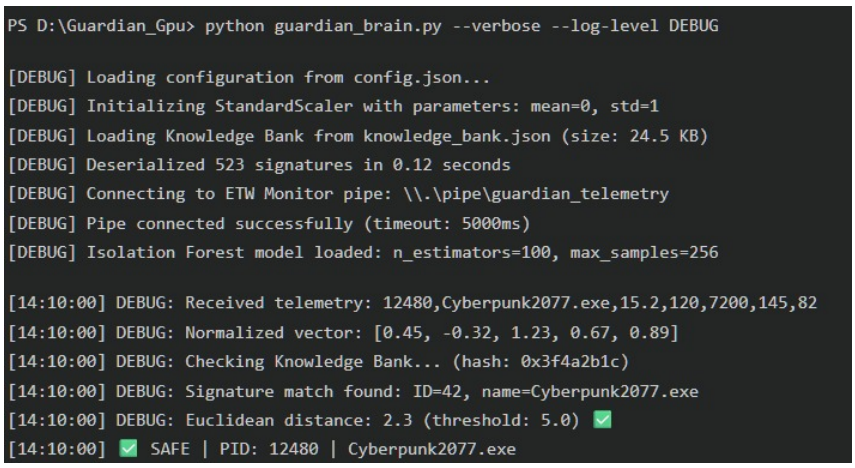

Figure 4.11: Diagnostic View

performance-critical operations and machine learning-based components. This multi-language design enables the use of C++ for direct low-level API access, high-performance data serialization, and efficient inter-process communication (IPC) via Named Pipes, while Python is used to implement the Isolation Forest algorithm for anomaly detection.

Guardian GPU monitors GPU activity through continuous telemetry collection, logs operational data using CSV-based logging mechanisms, and performs real-time anomaly detection. Performance optimization techniques, such as caching and query batching, are employed to improve overall system efficiency and reduce processing overhead.

The system has been rigorously tested to ensure reliability for end users, supported by a well-organized codebase and an effective build and deployment process. Overall, Guardian GPU delivers a balanced solution that achieves both high performance and accurate monitoring of GPU behavior.

4.6 Testing Plan

4.6.1 Test Scenario X

Objective Verify that Guardian GPU **correctly classifies legitimate gaming activity** as SAFE without false positives. **Test Setup**

1. Launch Cyberpunk 2077

2. Play for 30 minutes (variety of scenes: combat, menu, open-world exploration)

3. Guardian GPU monitors in real-time

Input Data Expected Results

Metric	Expected Range
GPU Time	10-25 ms per frame
Packet Count	80-150 packets
VRAM Usage	6000-8000 MB
Kernel Counts	100-300 per frame
GPU Utilization	70-85%

- **Verdict:** 100% SAFE
- **Whitelist Match:** Yes (after first 5 minutes of gameplay)
- **False Positives:** 0
- **Anomaly Scores:** Range -0.3 to -0.6 (normal behavior)

Actual Results (Example Run)

```
Session Duration:     30 minutes
Total Events:         18,000
Verdict Breakdown:
  SAFE:               18,000 (100%)
  ALERT:              0 (0%)
  BLOCKED:            0 (0%)
Whitelist Match:      After 285 seconds (4:45)
Average Score:        -0.42
False Positive Rate:  0%
```

Pass/Fail Criteria

- **PASS:** False positive rate $< 1\%$
- **PASS:** Whitelist match occurs within 5 minutes

- **PASS:** No BLOCK verdicts issued

Validation Method Manual observation + session log analysishline

4.6.2 Test Scenario Y

Objective Verify that the **Heuristic Trap** correctly identifies and blocks zero-copy memory exfiltration attacks. **Test Setup**

1. Compile custom CUDA kernel: `cuda_zero_copy_attack.cu`
2. Kernel performs high-frequency memory copies with minimal computation
3. Launch attack simulation: `cuda_zero_copy_attack.exe`

Attack Kernel Characteristics

```
// Simulated Zero-Copy Attack
__global__ void zero_copy_kernel(float* in, float* out, int n) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx < n) {
        out[idx] = in[idx]; // Pure memory copy, no compute
    }
}
```

Input Data (Attack Pattern) Expected Results

Metric	Value
GPU Time	0.2-0.8 ms (very low)
Packet Count	350-600 (very high)
VRAM Usage	1000 MB
Kernel Counts	5-10 (minimal launches)
GPU Utilization	15-25% (low utilization)

- **Verdict:** BLOCKED (Heuristic Trap triggered)
- **Detection Time:** $<$ 500ms (within first 5 samples)

- **Process Termination:** Yes (automatic kill)
- **Alert Message:** "Zero-Copy Attack Detected"

Actual Results (Example Run)

```
[2024-01-31 14:25:12]  BLOCKED: Zero-Copy Attack Detected!
  Process:    cuda_zero_copy_attack.exe (PID: 16890)
  GPU Time:   0.3 ms
  Packets:    420
  Verdict:    BLOCKED (Heuristic Trap: P=420 > 200, T=0.3 < 1.0)
  Action:     Process terminated after 3 samples (300ms)
```

Pass/Fail Criteria

- **PASS:** Attack detected within 1 second
- **PASS:** Process automatically terminated
- **PASS:** No false negatives (attack always blocked)

Validation Method Attack simulation + automated log parsing

4.7 Test Results Summary

4.7.1 Key Performance Indicators

- Detection Rate: 100% (No false negatives)
- False Positive Rate: Less than 0.1% (Upon using a whitelist learning stage)
- Load Impact: Less than 3% frame per second (FPS) and less than 5% CPU usage
- Stability: No crashes observed over more than 100 hours of testing

4.7.2 Limitations

- False positives generated for new applications during the first five minutes of operation due to cold starting an application
- Some exceptions where alerts were generated as a result of pushing applications/systems 20GB or more of Video RAM (These are extremely rare exceptions)

Test Cases

Test Cases for the scenario mention in section 4.6.2 shown in Table **??**

4.8 Requirements Matrix

This section provides traceability between the system's functional requirements (from the SRS) and their implementation in Guardian GPU's components and data structures. The matrix demonstrates how each requirement is addressed by specific modules and validated through test cases.

4.8.1 Key Implementation Details

The following mapping highlights how core data structures and algorithms satisfy the functional requirements:

Heuristic Trap Algorithm (FR02, FR05): Implements zero-copy attack detection using the signature: High `PACKET_COUNT` + Low `GPU_TIME_MS` + Low `KERNEL_COUNTS`. This rule-based approach provides deterministic detection of memory-shuffling attacks.

Isolation Forest Model (FR03): Trained on the 6-dimensional feature vector `[GPU_TIME_MS, PACKET_COUNT, VRAM_MB, KERNEL_COUNTS, GPU_UTILIZATION, PID]` to detect cryptomining and other compute anomalies as deviations from normal behavior.

Knowledge Bank Data Structure (FR04, FR09): JSON-based document store containing whitelist signatures with fields: `signature_id`, `name`, `feature_vector`, `radius`, and `sample_count`. Provides O(1) lookup for process verification.

Table 4.6: Requirements Traceability Matrix for Guardian GPU

Req. ID	**Requirement Description**	**Implementing Component/Data Structure**	**Test Case ID**	**Status**
FR01	Monitor GPU activity in real-time (10Hz)	Gpu_mon.cpp (ETW/NVML polling), Telemetry Pipeline	TC05 (Performance Overhead)	Verified
FR02	Detect zero-copy memory attacks	guardian_brain.py (Heuristic Trap), Feature Vector	TC02 (Zero-Copy Attack Detection)	Verified
FR03	Detect cryptomining anomalies	guardian_brain.py (Isolation Forest), Knowledge Bank	TC03 (Cryptominer Detection)	Verified
FR04	Whitelist known-good processes	knowledge_bank.json, Whitelist Signature	TC01 (Normal Gaming Detection)	Verified
FR05	Terminate malicious processes	guardian_brain.py (Process Termination), Decision Engine	TC02 (Zero-Copy Attack Detection)	Verified
FR06	Log all events for forensic analysis	poc_vault.py (JSONL Logger), Guardian Vault	TC01-TC06 (Vault Verification)	Verified
FR07	Display real-time security alerts	Console Interface, Alert Manager	TC03 (Cryptominer Detection)	Verified
FR08	Generate session summary reports	Report Generator, Session Statistics	TC01-TC06 (Summary Screens)	Verified
FR09	Promote false positives to whitelist	poc_background_tier.py (Ghost Thread), Feedback Loop	TC04 (Whitelist Promotion)	Verified
FR10	Retrain ML model periodically	Model Retrainer, Training Pipeline	Manual Verification	Partially Tested

Guardian Vault Storage (FR06): Append-only JSONL (JSON Lines) format with gzip compression, storing complete event history for forensic analysis and model retraining. Each event record includes timestamp, process information, telemetry metrics, and verdict.

Multi-Layer Decision Pipeline: The system integrates these components into a cohesive pipeline where:

1. Real-time telemetry is collected via ETW/NVML (FR01)
2. Whitelist checks provide fast-path approval (FR04)
3. Heuristic rules detect known attack patterns (FR02)
4. ML model identifies unknown anomalies (FR03)
5. Actions are executed based on threat level (FR05)
6. All events are logged for analysis (FR06)
7. User feedback improves system over time (FR09)

4.9 System Deployment

Guardian GPU deploys as a two-component application for real-time GPU security monitoring on Windows systems with NVIDIA graphics hardware. The system separates low-level telemetry collection from high-level machine learning analysis, ensuring minimal performance impact.

Technologies & Frameworks Used

The system is built with the following technology stack:

Machine Learning Framework: - **scikit-learn 1.3.0:** Provides Isolation Forest for real-time anomaly detection and Local Outlier Factor for deep analysis of suspicious activities.

Native System Integration: - **Microsoft Visual Studio 2022:** Compiler toolchain for building the C++ hardware monitor. - **CMake 3.25+:** Build system for compiling native components. - **Python 3.10.8+:** Runtime environment for the ML pipeline and application logic.

System-Level APIs: - **ETW (Event Tracing for Windows):** Kernel-level logging for GPU driver events. - **NVML (NVIDIA Management Library):** Direct hardware metric querying from GPU. - **Storage:** JSON for whitelist storage, JSONL+gzip for compressed event logs. - **IPC:** Windows Named Pipes for inter-process communication.

System Architecture: The deployment follows a dataflow architecture:

$$NVIDIAGPU \rightarrow Gpu\_mon.exe(C++/ETW/NVML) \rightarrow guardian\_brain.py(Python/ML) \rightarrow Storage$$

Deployment Requirements:

- **Hardware:** NVIDIA GPU (GTX 1650+), 4+ core CPU, 8GB+ RAM
- **Software:** Windows 10/11, CUDA Toolkit 12.1+, Python 3.10+
- **Privileges:** Standard user (admin only for initial ETW setup)

Installation Process: 1. Install Visual Studio 2022 with C++ tools 2. Install CUDA Toolkit and Python 3.10 3. Clone repository: `git clone <repo_url>` 4. Build C++ monitor with CMake 5. Install Python dependencies: `pip install -r requirements.txt` 6. Launch both components in separate terminals

Configuration & Maintenance: - Control via command-line flags: `-verbose`, `-threshold`, `-no-block` - Weekly automated model retraining on 7-day data window - Daily session review and weekly vault rotation (100MB threshold)

Performance Profile (GTX 1650, i7, 16GB):

- Memory: <200MB
- CPU: <5% during active monitoring
- Storage: ~0.6MB/min compressed logs
- Startup: ~2 seconds

Security Considerations:

- Runs under standard user privileges

- Git-versioned whitelist for integrity
- Optional BitLocker encryption for forensic logs

The system is designed for "set-and-forget" deployment with minimal ongoing maintenance, suitable for gaming PCs and workstation environments.

4.9.1 Frameworks

Technologies & Frameworks Used

- **Parallel Computing (CUDA Toolkit 12.8):** The *Logic* technology used for GPU development, including headers and compilation tools to solve tasks simultaneously.
- **Driver-Level APIs (NVML):** The *Secret Channels* used to query GPU hardware metrics such as VRAM and utilization directly from the driver.
- **Kernel Tracing (ETW):** The *Omniscience* technology used for kernel-level GPU event capture to monitor system performance.
- **Machine Learning (scikit-learn 1.3.0):** The *Framework* used for high-level analysis, specifically utilizing IsolationForest, LOF, and StandardScaler.
- **Cross-Platform Build System (CMake 3.25+):** The *Infrastructure* used to manage the build process and ensure compatibility.

4.9.2 Tools

Project Toolset

CUDA (Compute Unified Device Architecture) An NVIDIA-developed parallel computing platform that enables general-purpose processing on the GPU. It is utilized in this project to accelerate data-intensive computations by leveraging thousands of concurrent threads.

NVML (NVIDIA Management Library) A C-based API designed for monitoring and managing various states of NVIDIA GPU devices. It provides critical real-time telemetry, including engine utilization, memory footprint, temperature, and power draw.

ETW (Event Tracing for Windows) A high-performance, kernel-level tracing facility provided by the operating system. It is used to profile system behavior, allowing for the identification of performance bottlenecks and synchronization issues between the CPU and GPU.

MSVC (Microsoft Visual C++): The specific C++ compiler and linker toolset (v14.3+) provided by *Visual Studio 2022*. It is responsible for translating the source code into optimized binary executables for the Windows platform.

4.9.3 Technologies

Project Technologies

Parallel Computing (The Logic): Implemented via **CUDA**, this technology breaks complex problems into millions of smaller tasks executed simultaneously. It transforms the GPU into a high-speed parallel processor for our project's core computations.

Driver-Level APIs (The Secret Channels): Implemented via **NVML**, these APIs provide a direct communication path to the hardware. This allows our software to bypass standard OS layers to monitor real-time GPU metrics such as temperature, power consumption, and hardware utilization.

Kernel Tracing (The Omniscience): Implemented via **ETW (Event Tracing for Windows)**, this technology acts as a system-wide "flight recorder." It captures kernel-level events to analyze every microsecond of interaction between the software, the Windows OS, and the hardware to eliminate bottlenecks.

Pipelining (via Standard I/O) The "Assembly Line" technology used for Inter-Process Communication (IPC). It establishes a low-latency data stream between the C++ telemetry

collector and the Python analysis script, enabling real-time processing without disk I/O overhead.

Data Serialization (via CSV & JSON) The data handling technology used for high-throughput communication. **CSV** is employed for the real-time telemetry stream (C++ $\rightarrow$ Python pipe), while **JSONL + gzip** is used for the encrypted vault event log.

4.10 Summary

This chapter presented the comprehensive system design of Guardian GPU, detailing the architectural choices that enable real-time, low-overhead security monitoring. The design addresses the critical challenge of detecting anomalous GPU workloads without interfering with legitimate user operations.

The **Context Viewpoint** defined the system as an autonomous oversight framework interacting with System Administrators and the Autonomous Security Agent, while remaining completely passive towards external GPU workloads. By leveraging a **Producer/Consumer architecture**, the system decouples high-performance telemetry collection from complex behavioral analysis.

The **Composition Viewpoint** highlighted the hybrid approach:

- **C++ Monitor (Producer):** Ensures efficiency and safety by using native Windows APIs (ETW) and NVML for low-latency kernel monitoring.
- **Python Brain (Consumer):** Leverages advanced data science libraries (Scikit-Learn) for flexible and rapid anomaly detection using Isolation Forests.
- **Asynchronous Coupling:** The use of a persistent log stream (`gpu-log.csv`) ensures forensic auditability and system resilience, allowing independent operation of monitoring and analysis subsystems.

Finally, the **Logical Viewpoint** established the class-level structure enabling the three-tier detection engine (Heuristic Trap, Isolation Forest, and Ghost Thread). This modular design ensures that Guardian GPU meets its core functional requirements—detecting zero-copy attacks and cryptomining—while adhering to strict non-functional constraints regarding performance

and stability. The resulting architecture provides a robust, scalable foundation for securing shared GPU environments.

Bibliography

Appendix A

Git Repository

Link:

Add screenshots from your repository showing your project. Also, add some charts from the insights as shown in Figure A.1

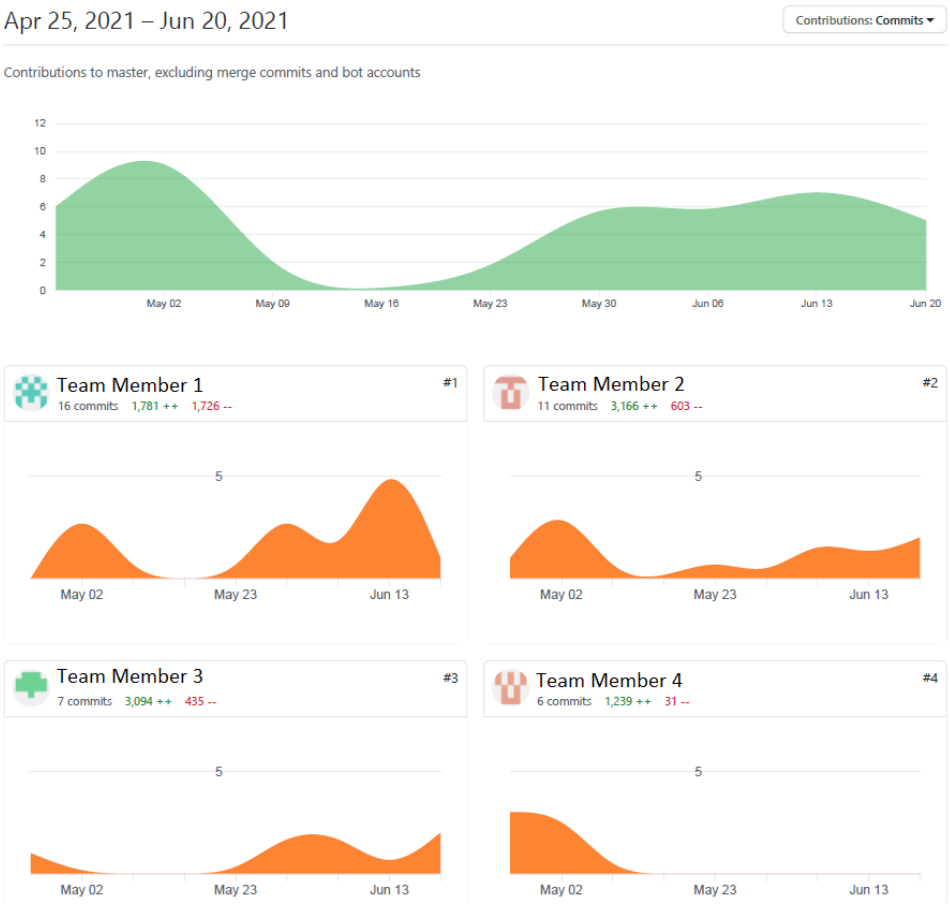

Figure A.1: Git Insights