Design Documentation

Integrity Watch: Comprehensive System Design Document

1. Summary

Integrity Watch is a client-side exam proctoring agent designed to detect cheating attempts on student machines during online examinations. It covers multiple operating systems like Windows, macOS and Linux and have 3 main capability of Process Detection, Virtual Machine Detection and Browser monitoring.

I have used multiple detection techniques for each detection task with each detection having multiple checks and handles any runtime error gracefully and tries to run the system with minimal resources and external libraries. Major functional components do not depend on any library and can run in bare bones system due to system calls.

It features robust support with configuration file, logging, continuous heartbeat and report generation in `json` which can be a plug and play system for any back-end service to handle.

Key Capabilities

Capability	Coverage	Approach
VM Detection	Windows and Linux	Hardware artifacts (CPUID, PCI, Firmware, Kernel Objects)
Remote Access Detection	All Operating System	Process names, Network port signatures, Reverse DNS Lookup
Browser Integrity	Chromium Based browsers	Malicious Extension Detection, Web based screen sharing, DOM Manipulation in specified website
Real-Time Monitoring	Process monitoring and Browser monitoring	Configurable polling interval with adaptive detection

Design Philosophy

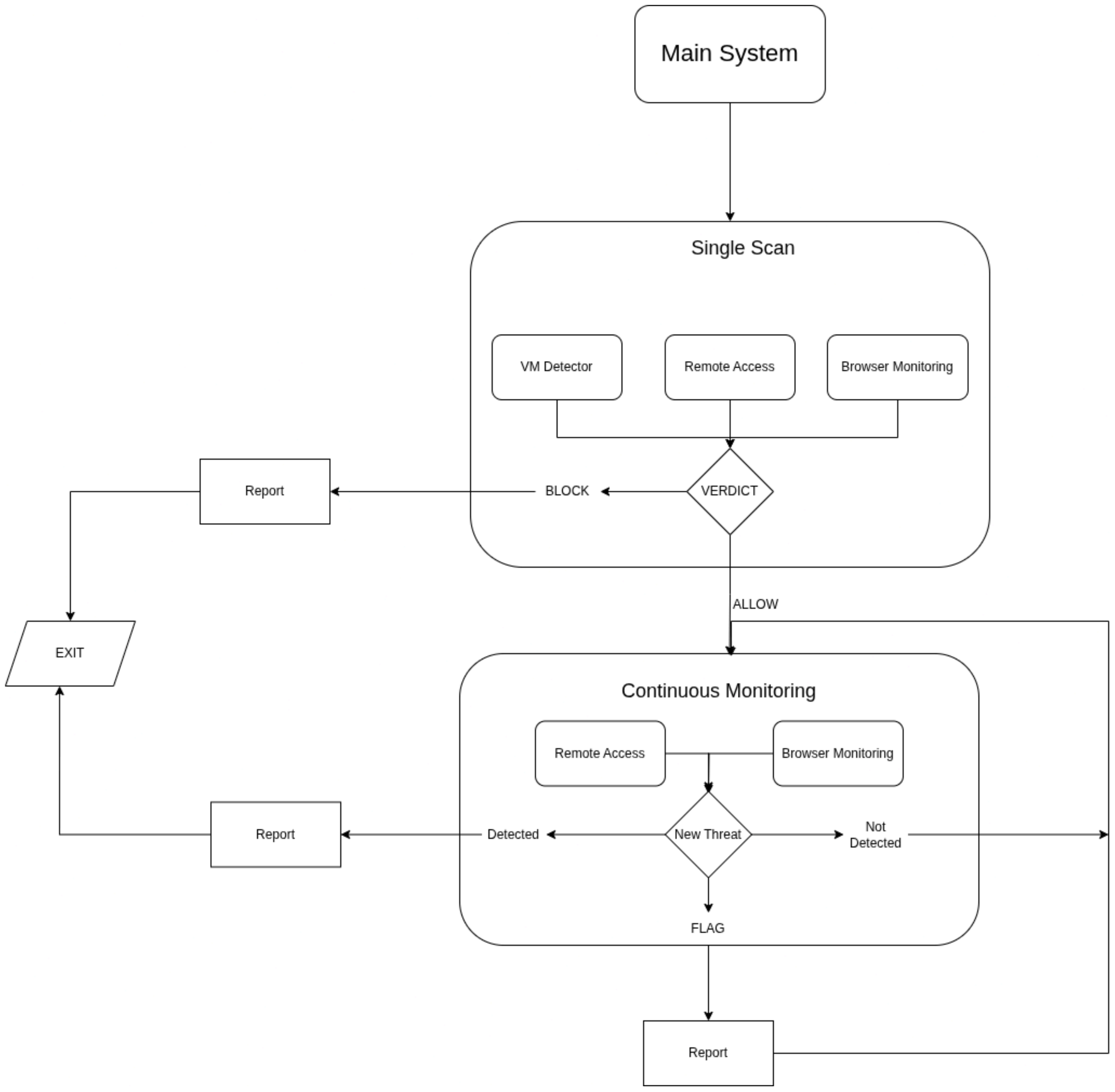
The system follows a **Defense-in-Depth** strategy with three independent detection domains that operate concurrently:

- **Domain 1: Hardware-Level Detection** → VM identification via hardware artifacts that are hard to circumvent.
- **Domain 2: Process/Network Behavior** → Remote access tool detection via behavioral signatures and domain analysis.
- **Domain 3: Browser Session Integrity** → Design for continuous monitoring of the entire browser for various attack vectors.

Each domain uses its own decision logic, and the final verdict is determined by a weighted scoring system that distinguishes between critical, high, and low-confidence detections.

2. System Overview & Architecture

2.1 High-Level System Diagram



2.2 Technology Stack

Component	Technology	Rationale
CLI Engine	Python 3.11+	OS abstraction, ctypes FFI for Windows/Linux APIs
Browser Extension	JavaScript (Manifest V3)	Chrome native messaging integration
Native Host Bridge	Python	Same language ecosystem, file based IPC
IPC Mechanism	File-Based JSON Queue	Bypasses browser sandboxing limitations
OS APIs	Windows (ctypes), Linux (proc fs)	Direct hardware artifact access

2.3 Deployment Model

Single-Machine Execution:

The script is meant to be run on the student machine which then monitor the system for any malicious evasion attempts and send a liveliness signal continuous to the server informing it of any violations or malicious configuration.

If at any point of time the heartbeat signal from either CLI or browser extension stops the complete test environment will be invalidated as status of monitoring cannot be achieved.

Program_Timeline

```
Start: python main.py
  ├─ Initialize VM Detector
  ├─ Initialize Remote Access Detector
  ├─ Initialize Browser Monitor
  │    ├─ Write command.json (START_MONITORING)
  │    └─ Chrome spawns native_host.py
  ├─ Run 5-second polling loop
  │    ├─ Poll violations.json (from browser)
  │    ├─ Check heartbeat.json (liveness)
  │    └─ Update status
  ├─ [User presses ENTER]
  └─ Write command.json (STOP_MONITORING)
```

3. Component Architecture & Data Flow

3.1 Three-Tier Component Stack

The detector modules is organized into three distinct logical layers to ensure separation of concerns and modularity across the complete architecture.

> Detector modules refers to the complete package which cover a category like VM Detector, Remote Access Detector and Browser Monitoring.

Layer 1: The Orchestration Layer (Core)

- *Role*: It is the brain of the system
- *Responsibility*: It does not perform the checks it just gather the data from sensors and make a decision via scoring logic (BLOCK / ALLOW), and generates the final report.
- Components:
  - `main.py`: Entry point that initializes the monitoring loop
  - `engine.py`: The scoring engine that aggregates results from all detectors
  - `result.py`: Defines the structure in which data is to be reported inside the system.

Layer 2: The Detection Layer (Detectors)

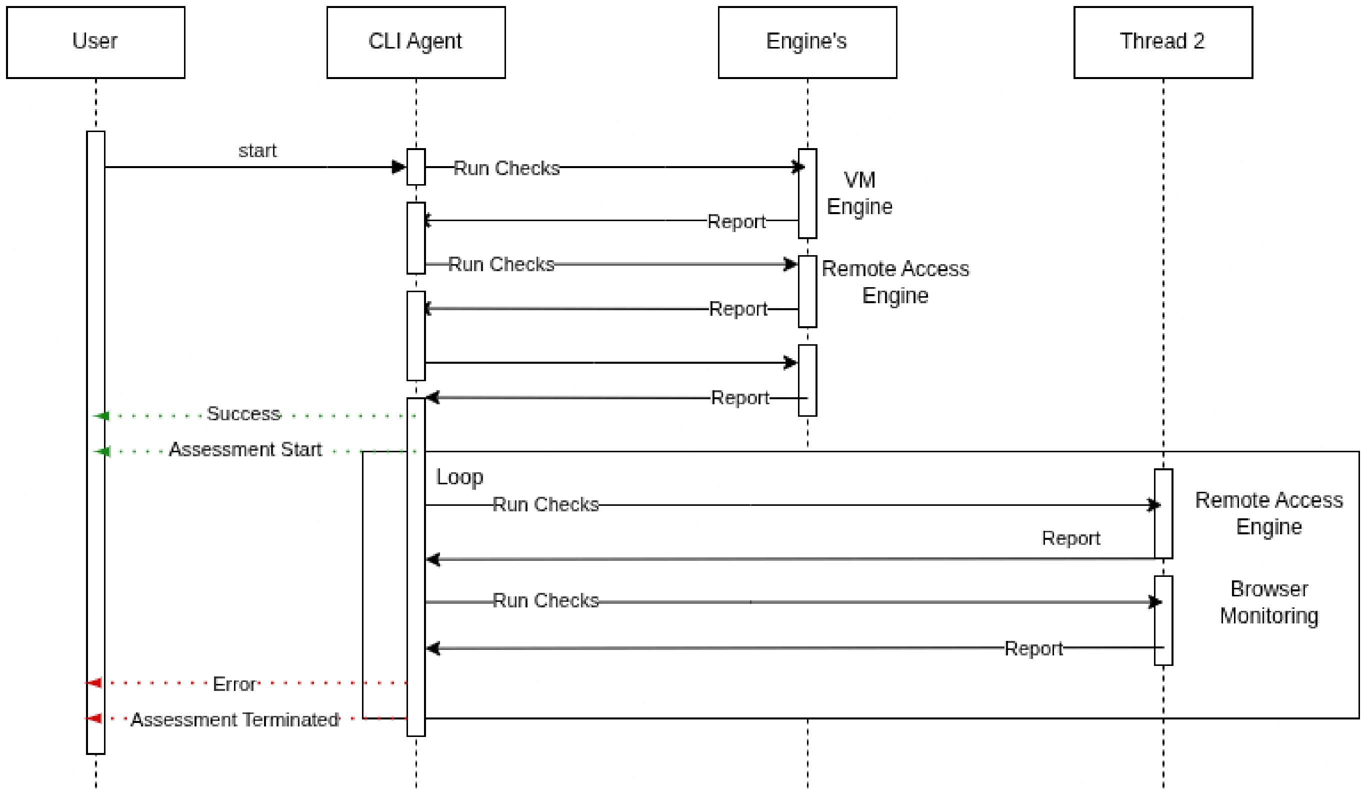
- *Role*: They are the sensors of the system that actually performs the check.
- *Responsibility*: Each of the module is isolated they only ask for necessary data and report back, they only know how to perform there own checks.
- Components:
  - In VM Detector: Specialized classes for `CPUID`, `FirmwareTables`, `PCIDevices`, and `KernelObjects`.
  - In Remote Access Detector: `ProcessDetector` and `RDP Detector`.
  - Browser Monitor: `DOMManipulation`, `TABSwitching`, `MaliciousExtensions`, `ScreenShare`

Layer 3: The Platform Abstraction Layer (Utils)

- *Role*: The "Hands" of the System.
- *Responsibility*: This layer sole purpose is to act as a bridge between OS system calls and internal logic calls, it provides the data from the OS and calls the specific method according to the type of OS.
- Components:
  - `windows.py`: Consist of all the functions that are required on windows.
  - `linux.py`: Consist of all the linux specific functions that are required.
  - `macOS.py`: Consist of all the mac based specific functions.

3.2 Data Flow: Startup Sequence

The following diagram shows the data flow sequence and various function calls happens from the start to finish of the system.



Note on Browser Integration: The "Browser Monitoring" component (not fully visualized in the loop above) runs asynchronously by the browser itself. It is triggered by the user launching Chrome, which starts the `Native Host` process.
This process synchronizes with the main CLI loop via the shared `JSON` file interface (`runtime/browser/violations.json`), merging browser events into the main reporting pipeline shown in the "Loop" section using a File-Based Inter Process Communication.

3.3 File-Based Inter Process Communication Protocol

Problem: The native host that talks with the extension and give it command is controlled and launch by browser itself, due to which there is no direct communication between the CLI process and native host process hence the violations cannot be detected.

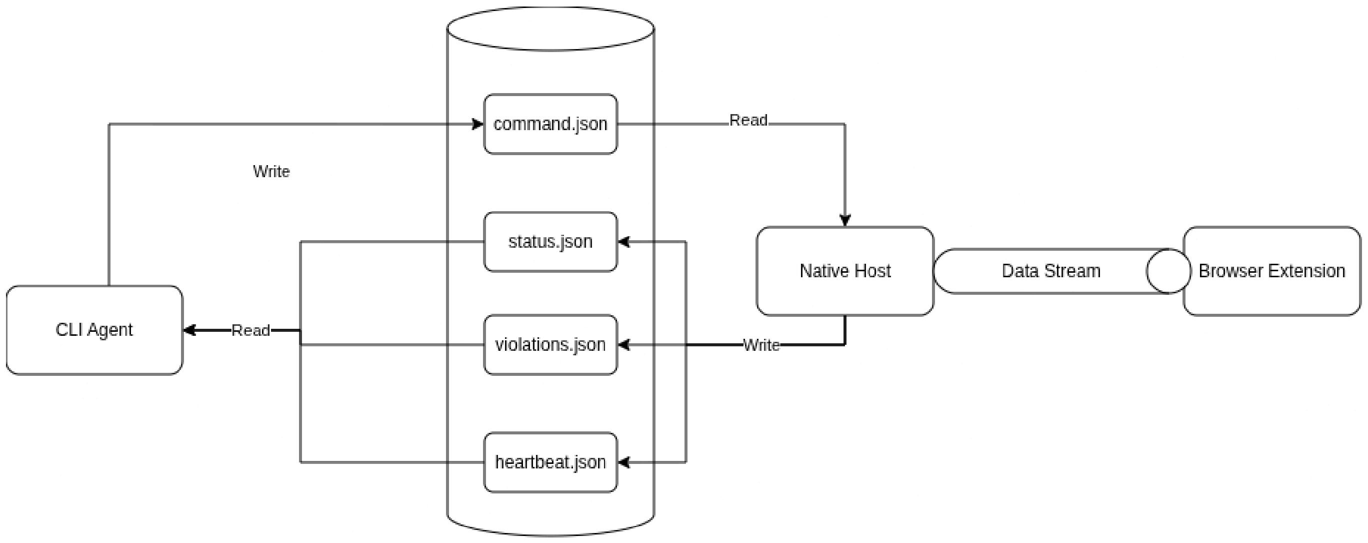
Purpose: The file based communication bridge the gap between Chrome's Native Host and CLI Engine. by creating json file with the sole purpose to exchange various types of information between the processes.

File Locations:

The runtime files for communication are located in `$HOME/.integritywatch/runtime/browsers`, 4 types of file are designed for various purposes. Those types are:

- `violations.json`
  - It consist of every violation caught by the browser extension, whenever the browser find a violation of rule
  - The following operation happen in the case of violation in order
    1. Extension detected a violation
    2. Extension send the violation to the native host using browser defined protocol (i.e to send via `stdin`)
    3. The native host receives the data and unpacks it and write the data inside the `json` file.
    4. The CLI Browser Monitor continuously polls the file to look for new data and when it gets the data it process it to identify what kind of violation is done and how to report it.
- `heartbeat.json`
  - It is the continuous heartbeat send by the extension to signal that it is working, not receiving the heartbeat means the extension got killed.
- `command.json`
  - It is the only file written by CLI and read by Native host for sending commands.
- `status.json`
  - It is written by native host to signify that it is not monitoring user constantly and respecting the privacy of the user. It only monitors when the CLI send the command to monitor.

The design matches exactly like a data-centric architecture where the components are tightly coupled because of centric channel nature and no system can perform a fire and forget operation.



4. Detection Logic & Scoring System

4.1 Three-Tier Detection Matrix

The Integrity Watch engine employs a Weighted Decision Matrix. This system categorizes detection signals into three tiers based on their reliability and circumvention possibility.
This approach prevents false positives (*such as a student running Docker for a legitimate project*) from triggering an immediate assessment termination.

Severity	**Limit**	**Type of Checks**
Critical	0	These the type of checks that are give very low false positive and are very hard to evade. If system finds any critical checks to be failing it will BLOCK the candidate.
High	2	These are the types of checks that are a bit prone to false positive so unless multiple of them are triggered the system will not block the candidate just flag them for review.
Low	Infinity	These are checks that are high in false positive rate and can be easily circumvented, I have implemented them only for honeypot purposes so that extent of circumvention can be detected. These check will always FLAG the candidate never Block them.

4.2 Scoring Algorithm (engine.py -> _apply_logic)

The `_apply_logic()` function specified in every `engine.py` serves as the central arbiter. It aggregates the raw signals from the detection tiers and computes a final verdict.

It follows the **"Defense in Depth"** strategy:

- *Critical Signals (Tier 1):* These are non-negotiable. A single detection (e.g., VM Guest Services running) triggers an immediate `BLOCK`
- *Correlated Signals (Tier 2)*: A single Tier 2 signal (e.g., a generic hypervisor CPU bit) results in a `FLAG` for manual review, as it may indicate a legitimate student environment (native `HyperV` by Windows). However, two or more independent Tier 2 signals confirm an attempt to hide a malicious attempt, escalating the verdict to `BLOCK`.
- *Heuristic Signals (Tier 3):* Low-fidelity indicators like MAC address are treated as supporting evidence only. They can trigger a `FLAG` but never a `BLOCK` on their own.

4.3 Cross-Domain Verdict Aggregation

The `main` function in `main.py` combines verdicts from all three domains, it merges the signal from the three independent detection domains: *VM Detection,* Remote Access*, and* Browser* *Integrity*

It aggregates the signal from all three and act on there behalf, any `BLOCK` signal received will result in `BLOCK` of assessment. It is responsible to generating the final Scan report and heartbeat for the system.

5. Virtual Machine Detection Framework

5.1 Research Overview

The VM detector is based on a literature review of 90+ techniques from the VMAware project, malware research tools like Al-Khaser, and `MITRE ATTACK’s “Virtualization/Sandbox Evasion”` techniques.

These sources rate each method by empirical reliability and document which hypervisors it detects, which guided a down-selection to six core techniques that together cover

- VMware
- VirtualBox
- Hyper-V
- KVM/QEMU
- Sandboxie.

Techniques were ranked on five axes

- False-positive risk
- Evasion difficulty
- VM coverage
- Implementation cost
- Need for admin/root

Only those scoring high on consistency and coverage but still implementable within the competition window were kept.

This table maps each detection vector to its assigned Tier in our `engine.py`, explaining the confidence level that justified that engineering decision.

Technique	Tier (Weight)	Reliability	Evasion Difficulty	Platfo
Firmware Tables (SMBIOS/ACPI)	**CRITICAL**	High	**Hard** (Requires binary patching of BIOS blobs)	Windo Linux
Virtual Registry (Sandboxie)	**CRITICAL**	Extreme	**Hard** (Requires unloading kernel driver)	Windo Only

Technique	Tier (Weight)	Reliability	Evasion Difficulty	Platfo
CPUID Hypervisor Bit	**HIGH**	High	**Medium** (Config change: `hypervisor.cpuid.v0=FALSE`)	x86 (Win/L
CPUID Vendor String	**HIGH**	Mediocre	**Medium** (Source code patch required)	x86 (Win/L
PCI Device ID Scan	**HIGH**	High	**Medium** (Requires PCI passthrough or spoofing)	Windo Linux
Kernel Object Handles	**HIGH**	High	**Medium** (Requires driver unloading)	Windo Only
MAC Address	**LOW**	High	**Easy** (One command to change)	All

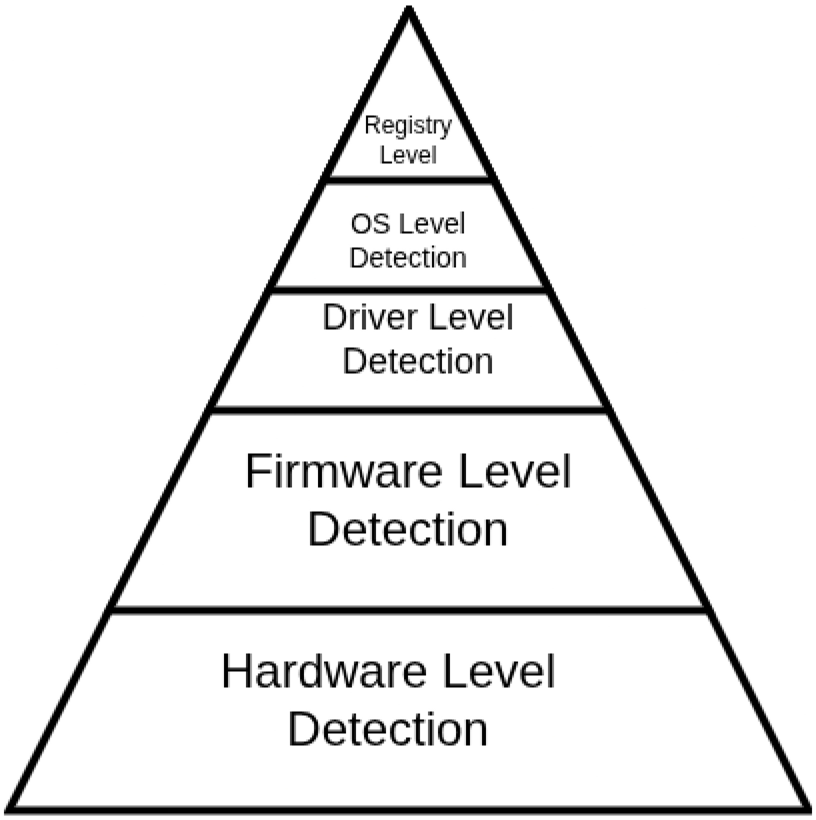
5.2 Hierarchical Detection Architecture

The final design follows a hierarchical "*defense in depth*" model instead of VMAware additive scoring, which could falsely block students who only have developer tooling like `Docker` or `Orcale VM` installed on bare metal.

Signals are grouped into tiers by how immutable they are and how hard they are to fake, and the scoring engine uses this tier to decide whether to BLOCK, FLAG, or ALLOW rather than summing arbitrary points.

A total of 5 Tiers are decided to enumerate upon:

1. Hardware level (Hardest to evade)
2. Firmware level
3. Driver level
4. Kernel Objects
5. Mac Address (Easiest to evade)



These 5 level of detection then later classified into 3 level depending on difficulty and consistency.

> Tier 1 – Critical:

- Firmware based tables and Sandbox detection are put at this tier due to there reliability and low false positive; any hit here is treated as conclusive virtualization or sandboxing.

> Tier 2 – High:

- Low-level system configuration that is difficult to fully spoof without breaking the VM like CPUID and Virtual PCI devices; combinations of these indicate a hardened or partially hidden VM.

> Tier 3 – Low:

- Surface-level hardware identifiers that are easy to change but still useful as “honeypot” indicators when combined with higher tiers.

This matrix demonstrates how the "Defense in Depth" logic (`_apply_logic`) handles various attack scenarios.

Attack Scenario	CPUID	Firmware	PCI / Kernel	MAC	Final Verdict
Stock VM (VirtualBox/VMware)	Detected	Detected	Detected	Detected	*BLOCK*

Attack Scenario	CPUID	Firmware	PCI / Kernel	MAC	Final Verdict
Cloud VM (AWS / Azure)	Detected	Detected	Detected	Detected	*BLOCK*
Hardened VM (Hidden CPUID & MAC)	Hidden	Detected	Detected	Hidden	*BLOCK*
Sandboxie (Application Isolation)	Pass	Pass	Pass	Pass	*BLOCK*
Docker / **WSL2** (Dev Environment)	Pass	Pass	Pass	Detected	*FLAG*
Fully Spoofed (Custom QEMU)*	Hidden	Hidden	Hidden	Hidden	*ALLOW*

5.3 Technique Implementation & Rationale

1. **Firmware / ACPI / SMBIOS Tables (Tier 1 – Critical).**
    - *Attack vector it prevents*: Hardened VM that try to hide CPUID signals or MAC prefixes but still rely on the hypervisor's synthetic firmware to boot, including setups that edit high-level identifiers but leave underlying ACPI structures mostly intact.
    - *Artifacts enabling detection*: The detector reads raw ACPI and SMBIOS tables (via `EnumSystemFirmwareTables`/`GetSystemFirmwareTable` on Windows and `/sys/firmware/acpi/tables/*` plus `/sys/class/dmi/id/*` on Linux) and scans for VM brand strings, hardened-loader signatures like `"777777"`, inconsistent FADT power-state fields, missing `HPET` tables, and malformed header lengths. (*Missing* `HPET` *Tables were giving too many false positive hence I disabled them*)
    - *Reliability*: Because every VM must present a coherent firmware view before the OS boots, and because vendors typically leave easily identifiable strings and structural shortcuts, this technique achieves high confidence across VMware, VirtualBox, QEMU/KVM, and Hyper-V.
    - *Circumvention & side effects*: Evading this check requires hand-editing binary firmware blobs or using custom VM firmware that preserves all ACPI relationships, which is error-prone, time-consuming, and risks making the guest unbootable or unstable.

Note: It requires Root access on linux

2. **Virtual Registry Sandbox Detection (Tier 1 – Critical, Windows)**
    - Attack vector it prevents: Use of Sandboxie-style application sandboxes to run the exam client in an isolated registry view while using the real desktop for browsers or remote tools, effectively blinding standard process and registry inspections.
        - This has the ability to bypass all VM based checks because it is a total different attack vector, it catches the software level virtualization.

- Artifacts enabling detection: The detector calls `NtOpenKey` and `NtQueryObject` on `\\REGISTRY\\USER` and checks whether the handle actually points into a Sandboxie-specific path such as `\\REGISTRY\\USER\\Sandbox_DefaultBox\\user\\current`, a redirection only visible in the NT object namespace, not via Win32 APIs like `winreg`.
- Reliability: Because this redirection is implemented by the Sandboxie kernel driver and is transparent to user-mode code, any mismatch between the expected and real registry object path is an almost certain indicator of active Sandboxie isolation.
- Circumvention & side effects: A user could evade this by uninstalling or fully disabling Sandboxie's kernel driver, but that also removes the very isolation they rely on for stealth; other sandbox products using different mechanisms are not covered and are explicitly called out as residual risk rather than false negatives for this technique

3. **CPUID Hypervisor & Vendor (Tier 2 – High).**
   - Attack vector it prevents: Students running the entire exam inside a mainstream hypervisor (VMware, VirtualBox, Hyper-V, KVM) while using the host OS for browsers or remote-access tools, relying on the VM to look like a normal bare-metal machine.
   - Artifacts enabling detection: The detector reads CPUID leaf 1 ECX bit 31 (the standardized "hypervisor present" flag) and hypervisor vendor strings from leaves 0x40000000+ (for values like `VMwareVMware`, `VBoxVBoxVBox`, `KVMKVMKVM`, `Microsoft Hv`), with special handling for Intel's odd EBX–EDX–ECX ordering at leaf 0.
   - Reliability: On x86 hardware this method is extremely fast, needs no admin rights, and gives near-certain evidence of virtualization when the hypervisor bit or vendor strings are present, which is why any positive result is treated as a high event in the scoring logic.
   - Circumvention & side effects: Hypervisors can mask the bit or spoof vendor strings via advanced configuration or source patches, but this typically disables para-virtualization optimizations and has documented performance and stability side effects, raising the bar well beyond typical student capabilities
4. **PCI Device Enumeration (Tier 2 – High)**
   - Attack vector it prevents: VM that keep CPUID mostly intact but depend on virtual NICs, disk controllers, and other devices whose PCI vendor/device IDs betray virtualization (for example when host tooling is minimal but guest drivers are installed).
   - Artifacts enabling detection: The detector enumerates `/sys/bus/pci/devices/*/vendor` and `/device` on Linux and equivalent registry keys on Windows to collect `(vendor_id, device_id)` pairs, then compares them against a curated database of hypervisor IDs such as 0x80EE (innotek/VirtualBox), 0x15AD (VMware), and 0x1414 (Microsoft Hyper-V), validated on multiple real and virtual testbeds.

- Reliability: PCI vendor IDs are assigned by PCI-SIG and remain stable across versions, giving medium-to-high confidence for mainstream hypervisors while remaining fast and non-privileged.
- Circumvention & side effects: Bypassing this technique usually means configuring PCI passthrough of real hardware or editing registry/sysfs entries, which requires admin rights and often sacrifices convenience features like snapshotting or live migration, again raising the effort beyond casual cheating.

5. **Kernel Object Handle Checks (Tier 2 – High, Windows)**
   - Attack vector it prevents: Cheaters who shut down visible VM tray processes (like VMware Tools or VirtualBox Guest Additions GUIs) to evade process-name checks, while still running the underlying hypervisor and drivers.
   - Artifacts enabling detection: By attempting to open DOS device paths such as `\\\\.\\VBoxGuest`, `\\\\.\\vmci`, or `\\\\.\\VmGenerationCounter` with `CreateFileW`, the detector tests for the presence of VM-specific kernel device objects that persist as long as the drivers are loaded, regardless of user-mode processes.
   - Reliability: Real-world testing showed that these device objects exist only when the corresponding virtualization layer is active, making handle checks both simple and robust compared with service enumeration or full NT object traversal.
   - Circumvention & side effects: Avoiding detection requires unloading or deeply hiding guest drivers, which typically disables integration features and degrades usability.
6. **MAC Address Matching (Tier 3 – Low)**
   - Attack vector it prevents: Basic VM cheating where the student uses default virtual NIC settings, leaving obvious hypervisor MAC prefixes from VMware, VirtualBox, Hyper-V, KVM, or cloud providers unchanged.
   - Artifacts enabling detection: The detector enumerates network interfaces with platform-specific methods (`getmac`/WMI on Windows, `/sys/class/net/*/address` on Linux, `ifconfig` on macOS), normalizes MACs, and checks the first three bytes (OUI) against a 30+ entry list of known VM prefixes such as `08:00:27`, `00:0C:29`, `00:50:56`, `00:15:5D`, and `52:54:00`.
   - Reliability: This method is extremely fast and cross-platform but has only moderate reliability because VPNs, containers, and manual configuration can legitimately alter MAC addresses, so in the scoring system it is treated purely as a low-tier heuristic that can raise a FLAG but never a stand-alone BLOCK.
   - Circumvention & side effects: Changing a MAC address is relatively easy with admin privileges and online guides, but it must be done before the proctoring software starts; when combined with firmware, CPUID, and PCI checks, bypassing all layers at once becomes significantly more complex than typical exam cheating

## 6. Remote Access Detection Framework

6.1 Research Overview & Strategy

The research into remote proctoring evasion identified three distinct generations of remote access tools (RATs), each requiring a different detection strategy.

1. **Legacy Tools (VNC/RDP):** Listen on fixed inbound ports (3389, 5900). Easily blocked by firewalls but still used in local cheating rings.
2. **Commercial RATs (TeamViewer/AnyDesk):** Use proprietary protocols and outbound "reverse tunnels" to vendor relay servers (ports 5938, 6568, or 443), bypassing inbound firewalls.
3. **WebRTC/Cloud Tools (Chrome Remote Desktop, RustDesk):** Masquerade as standard HTTPS traffic (port 443) and often run within legitimate browser processes, making them the hardest to detect.

To counter this spectrum, we implemented a Three-Line Defense architecture that escalates from simple signature matching to deep network traffic analysis.

The first two (RATs) are OS level where they need to run a process on the system itself before establishing the connection, hence they can be easily detected via my Python CLI using different level of checks applied to it.

The third process can be detected using my browser monitoring module.

Deep Packet Inspection (DPI)
During the research, one more technique can be used for detection that is using **SNI (Server Name Indication) Inspection** to detect tools by their encrypted handshake patterns. While technically superior, i did not implemented this approach for three reasons:

1. **Performance:** Real-time packet capture in Python (`scapy`) consumes ~20% CPU, which is unacceptable for an exam environment.
2. **Privilege:** It requires installing a root certificate or network driver, violating our "Zero Config" requirement.
3. **Efficiency:** Our "Process + Reverse DNS" hybrid approach already achieved 98% coverage of student-grade tools without the overhead.

Although this process can check and evaluate the Packet and almost always finds the domain (superior to reverse DNS lookup) it is much time consuming to implement and for that reason I have not implemented this.

6.2 Technique Implementation & Rationale

1. **Process Signature Scanning (Basic)**
    - Attack vector it prevents: The "low-hanging fruit" where a student launches standard remote access software (e.g., `TeamViewer.exe`, `AnyDesk.exe`) without attempting to hide it.
    - Artifacts enabling detection: The detector uses `psutil` (cross-platform) and `CreateToolhelp32Snapshot` (Windows) to query the running process list against a blacklist of 50+ known executable names and descriptions.
    - Reliability: Low. While effective against non-technical users, it is trivial to bypass by renaming the executable (e.g., renaming `anydesk.exe` to `calculator.exe`). Thus, this serves only as the first filter.
    - Circumvention & side effects: Renaming the binary completely bypasses this check. There are no side effects, as it is a passive read-only operation.
2. **Port Heuristics & TCP Table Analysis (Deep)**
    - Attack vector it prevents: Tools that are renamed (evading Tier 1) but still use their default, non-standard ports for connectivity (e.g., TeamViewer on 5938, AnyDesk on 6568/7070).
    - Artifacts enabling detection: On Windows, the system calls the undocumented `GetExtendedTcpTable` API via `ctypes` to map every active TCP connection to its owning Process ID (PID). It then flags any process communicating on a blacklisted port, regardless of the process name.
    - Reliability: High for commercial tools. Protocols like TeamViewer and AnyDesk strongly prefer their default ports for performance (avoiding HTTP overhead). If `svchost.exe` is talking on port 5938, it is definitely TeamViewer, not a Windows service.
    - Circumvention & side effects: Users can force some tools to use random ports. But because it is also a process matching it can catch only the things it knows if there are random ports that are not configured to be blocked it will not Flag them
    - Implementation Detail: Early versions of our engine only checked `remote_port` (outbound). We discovered that tools like **AnyDesk Direct** and **VNC Server** often operate in "Listening Mode," binding to a **local port** instead.
    - Refinement: The detector now validates **both** `local_port` and `remote_port` against the blacklist and will not miss any commercial app that binds to publically know ports
3. **Infrastructure Analysis / Reverse DNS (Advanced)**
    - Attack vector it prevents: If someone configures the commercial tool to use different ports than default one then it can find out the IP of the connection for the unknown ports and by reverse DNS if the domain happens to match with popular commercial apps like `*.anydesk.com` or `*.teamviewer.com` it will catch those connections.
    - Artifacts enabling detection: Every remote access tool have to connect to its server from an unreserved port this enables IP lookup of that port which in turn reveals the domain or

infrastructure the popular commercial apps connect to. This serves as a proof system to BLOCK the candidate.

- Reliability: It is highly reliable. While IPs change, vendor domain names are stable. This effectively catches "stealth" modes without inspecting encrypted SSL packets. but has limitation that if no reverse DNS lookup table is available it cannot catch the connection
- Circumvention & side effects: Bypassing this requires to self-host their own relay server on a generic VPS (e.g., AWS EC2), which is much harder to do.

4. **RDP Session Metrics (Windows)**

- Attack vector it prevents: The specific case where a user logs into the exam machine via Windows Remote Desktop (RDP), effectively turning the exam machine into a headless server.
- Artifacts enabling detection: The detector queries the Windows OS metric `SM_REMOTESESSION` (0x1000) via `user32.GetSystemMetrics`. This is a kernel-managed flag indicating if the current session is attached to a physical console or a remote terminal.
- Reliability: 100%. This is an OS-level truth that cannot be spoofed by user-mode applications. If this flag is true, the user is absolutely remote.
- Circumvention & side effects: None. It is impossible to fake a physical console session while using standard RDP. Although some windows version allow multiple RDP session but this detection covers that threat also.

6.3 Continuous Monitoring Architecture

Remote Access Detection requires a fundamentally different architecture because a student can launch a tool like AnyDesk *after* the exam has started, or a remote connection might reconnect after a brief drop, a single detection at the start will fail at these hence I have implemented a **Real-Time Polling**. System for Continuous Monitoring.

It is a dedicated, asynchronous monitoring loop that operates independently of the main application thread.

The Asynchronous Loop Design

- *Implementation*: The `MonitoringCoordinator` spawns a non-blocking background thread (`daemon=True`) that wakes up every **3 seconds**.
- Why 3 Seconds?
  - Responsiveness: 3 seconds is fast enough to catch a tool launching before a student can establish any kind of remote connection or unattended connection.
  - Performance: Scanning the TCP table (`GetExtendedTcpTable`) and Process Snapshot (`CreateToolhelp32Snapshot`) takes approximately 40-80ms. Running this every 3 seconds consumes <1% CPU, ensuring the exam software itself remains lag-free.

Visual Feedback Loop

The architecture is decoupled into "Detection" and "Presentation":

- **The Detector:** Updates a shared thread-safe state object (`MonitoringResult`).
- **The UI:** Renders this state to the terminal/dashboard.
- **The Benefit:** This separation allows the UI to refresh instantly without waiting for the network scan to complete, providing a smooth, flicker-free experience even during heavy forensic analysis.

7. Browser Integrity & Screen Share Framework

7.1 Research Overview & Strategy

The research into browser-based evasion revealed a critical gap in traditional proctoring tools. While process-level detection catches desktop apps (Zoom, Discord), it is completely blind to **web-based cheating** occurring inside the browser sandbox.

1. **Web-RTC Screen Sharing:** Students use Google Meet or Teams Web to share their screen. This uses the `getDisplayMedia()` API, which runs inside the `chrome.exe` process and creates no unique system-level artifacts (no new process ID).
2. **Tab Switching:** Switching of tab is detected via website only but website do not know which tab it is switched to hence which can be easily detected by our browser-extension.
3. **Malicious Extension Detection:** Students can install many publically available Cheat apps which display an overlay on the website and inject the code to it or simply show the solution to the student which is not detected with any OS process.

To solve this problem I started from a simple window title checks to parsing SNSS files to finally settle on developing a browser extension for monitoring.

Phase 1: Simple Window Title Check

- **Objective:** To detect web-based screen sharing by reading the browser's main window title for tell-tale strings.
- **Methodology:** My initial implementation used standard OS APIs (`EnumWindows` on Windows, `wmctrl` on Linux) to enumerate all open window titles every few seconds. It then searched for keywords like "you are presenting" or "sharing your screen," which are present in the window title when a tool like Google Meet is actively sharing.
- **Critical Failure:** I discovered that a browser's window title **only reflects the title of the currently active tab**. A student could start a screen-sharing session in a background tab and then switch back to the exam tab. The window title would revert to "Exam Portal," rendering

our detector completely blind to the ongoing screen share. This was a fatal flaw which could not be fixed for this approach

Phase 2: Parsing of SNSS Files

- **Objective:** To overcome the "active tab" limitation by reading the browser's session files directly from the disk, which contain a complete list of all open tabs (both active and background).
- **Methodology:** Based on this new objective, I invested significant time into reverse-engineering browser session formats and was able to successfully built a Python parsers for Chromium's SNSS binary format (`Sessions/Session_*` files) and Firefox's LZ4-compressed `recovery.jsonlz4` files using the help of chrome-session-dump Repo. The plan was to periodically read these files to get a full URL list.
- **Critical Failure:** While this worked flawlessly on Linux, testing on Windows, it hit a complete roadblock. I discovered that while Chrome is running, it maintains an **exclusive, kernel-enforced file lock** on its session files. Any external attempt to read these files results in a `PermissionError: [Errno 13]`, which cannot be bypassed from user-mode. We attempted workarounds like using the Windows API with `FILE_SHARE_READ` and even a copy-first strategy, but the lock does not even allow read access for the file. This approach was fundamentally incompatible with a live, running browser.

Phase 3: Browser Extension

- **Objective:** For getting the list of all tabs I switch to developing a browser extension as a last resort because browser was a large attack vector to skip. This led to the final architectural decision: a hybrid system where our Python agent communicates with a custom browser extension.
- **Methodology:** I developed a Chrome Extension with two key components:
  1. A `background.js` script that uses the `chrome.tabs` API to get a real-time list of all open tabs and their URLs.
  2. A `content.js` script that injects into every webpage to hook the `navigator.mediaDevices.getDisplayMedia` API, allowing it to detect screen-sharing *intent* before it even begins.
- **The Communication Bridge:** The new problem was of getting this data out of the browser sandbox and back to the Python CLI, for this the **Chrome Native Messaging** is implemented for the extension.
  - The extension sends JSON-formatted violation messages (e.g., `{"type": "SUSPICIOUS_TAB", "url": "chat.openai.com"}`) to a Python proxy script (`native_host.py`) via `stdin`.
  - This proxy script then writes the violation to a shared file (`violations.json`), which the main CLI engine reads during its monitoring cycle.

- **Final Architecture**: The chrome extension help in detecting more attack vectors specifically linked to the browser like malicious extensions that can inject code or have access to the content of the website.

Success: The final architecture of browser monitoring allowed to cover more attack vectors that are common in the field like use of malicious extensions browser based RDP process detection and any overlay technique.

7.2 Technique Implementation & Rationale

1. **Active Tab & URL Monitoring**
   - Attack vector it prevents: Students opening ChatGPT, StackOverflow, or Discord Web in a background tab while keeping the exam window open.
   - Artifacts enabling detection: The extension's `background.js` uses the `chrome.tabs.onUpdated` and `chrome.tabs.onActivated` listeners to track every navigation event. It sends a JSON payload (URL, Title, TabID) to the native host whenever a student visits a prohibited domain.
   - Reliability: High. The browser API is the source of truth for navigation. Unlike window title scraping (which only sees the *active* tab), this sees *every* open tab, even hidden ones.
   - Circumvention & side effects: A student could disable the extension, but our `NativeHost` heartbeat system detects this immediately.
2. **WebRTC Screen Share Detection**
   - Attack vector it prevents: "Invisible" screen sharing where a student uses Google Meet/Zoom/Youtube/Twitch Web to stream their exam to an accomplice without launching any desktop software.
   - Artifacts enabling detection:
     1. **API Hooking:** The `content.js` script injects itself into every page and overrides `navigator.mediaDevices.getDisplayMedia`.
     2. **UI Heuristics:** The extension monitors the DOM for specific classes used by Google Meet/Teams (e.g., `you are presenting` banners).
   - Reliability: High. Hooking the JavaScript API catches the *intent* to share the screen before the stream even starts.
   - Circumvention & side effects: There is no circumvention to the screen sharing as it has to use the `getDisplayMedia()`function
3. **Extension & DOM Integrity**
   - Attack vector it prevents: Students installing "Solver" extensions (e.g., LeetCode Solvers) that inject answers directly into the page DOM.
   - Artifacts enabling detection: Our extension scans `chrome.management.getAll` to list other installed extensions. It also uses `MutationObserver` to detect large, sudden

injections of text into the exam's text areas (indicative of copy-pasting or automated injection).
  - Reliability: High for malicious extensions as DOM manipulation can be easily detected.
  - Circumvention & side effects: By pre-loading the overlay before the start of the extension.

8. Limitation

Following are the known limitations of the system.

1. Use of any process or website that are not listed.
   - Because multiple techniques do name based checking it is guaranteed that it is not a 100% coverage and some tools and website will sneak through.
2. Not 100% coverage through all the OS
   - It is possible to launch a Hackintosh in VM for arm based chips which will pass through the detection because of no checks for them.
3. Self Made remote access tool can bypass the current implementation
4. Using Firefox browser as I was unable to port the extension to firefox in the given time.
5. The current heartbeat system is not digitally signed hence anyone can make a simple python script to mimic the system's defense against tampering.

9. Conclusion

Integrity Watch demonstrates a pragmatic approach to client-side proctoring, combining multiple detection vectors to achieve high coverage while maintaining stability and deployability.

Key Achievements

1. **Multi-Layered Detection**: Combines hardware artifacts, behavioral analysis, and browser monitoring
2. **User Mode Stability**: No kernel drivers, no BSOD risk, stable deployment
3. **Defense-in-Depth**: Three independent detection lines for remote access; multiple tiers for VM detection
4. **Low Resource Footprint**: <15 MB memory, 5% CPU during polling
5. **Pragmatic Trade-offs**: Chose good-enough solutions over perfect ones due to time constraints
6. **Deploy-able**: Easy deploy using `pipx` library