Description

Title

System and Method for Detecting Synthetic Keystroke Activity via Cross-Referenced Logs and Snapshots

Background

In various computing scenarios, users or malicious actors may attempt to **simulate keyboard activity** to falsify productivity or bypass security checks. For example, fraudulent remote access often involves an attacker controlling a victim's computer over a network and **synthesizing keystroke events** to interact with software, since the attacker lacks physical access to the keyboard[1]. Similarly, an employee working remotely might run an auto-typing script to feign active work, or a student during an online exam might use a pre-recorded input script to cheat. These synthetic inputs can undermine the integrity of activity monitoring and security systems.

Detecting non-human or automated keystrokes is a challenging problem. Prior approaches have tackled parts of this challenge in isolation. For instance, research in keystroke dynamics (the analysis of typing rhythms and patterns) has shown that human typing patterns have unique variations that are difficult for bots to replicate[2]. Using such behavioral biometrics, systems can distinguish humans from automated programs by the timing and randomness in keystroke sequences[2]. Other security techniques monitor the provenance of input events: one known method compares low-level physical key events detected by the operating system with high-level keystrokes received by applications, flagging any input that did not originate from an actual physical keypress as a "synthetic" event[1][3]. This helps identify injected or programmatically generated keystrokes and can alert administrators when such events occur[4][5].

In the field of **employee monitoring and user presence verification**, separate solutions exist but are typically not integrated for anomaly detection. For example, some productivity monitoring software uses a computer's webcam for **facial recognition** to log which employees are at their workstation, thereby exposing cases where one worker might try to cover for an absent colleague by imitating activity on their device[6]. These platforms also often include **keylogging** features that record all keystrokes typed by the user (even those later deleted)[7], as well as periodic or trigger-based screen capture (screenshots) whenever the user switches windows, navigates to a new webpage, or performs certain actions[8]. Such screenshots enable managers to manually verify that the on-screen work corresponds with the recorded keystrokes and activities[8]. Despite these advances, current solutions generally **lack automated cross-correlation** among the various data sources. An admin may need to manually inspect logs

and images to notice if, for example, a burst of typing had no visible effect on the screen, or if keystrokes were logged while no person was present on camera. This manual analysis is time-consuming and prone to oversight.

Therefore, there is a need for an **integrated system** that automatically combines keystroke dynamics analysis, application context awareness, screen content change detection, and biometric user verification to determine the authenticity of user input. Such a system would instantly flag suspicious activity patterns (e.g., a rapid, continuous typing stream with perfectly uniform intervals, or typing that produces no corresponding changes in the user interface) that are indicative of scripted or simulated input rather than genuine human typing. By cross-referencing these multiple signals in real time, the system can provide a robust defense against both malicious automated input (as in malware or bot scenarios) and fraudulent behavior in productivity or exam settings. The present invention addresses these needs by providing a **multi-layered detection mechanism** that improves accuracy and reduces false positives compared to single-faceted solutions.

Summary of the Invention

The invention provides a **system, method, and computer-readable medium** for detecting synthetic or simulated keystroke activity through multi-channel data correlation and analysis. In one aspect, the system comprises several interconnected modules that work in concert to monitor and verify user input authenticity:

- **Keystroke Behavioral Analysis Module:** This module monitors the timing and characteristics of keystrokes (from a keyboard or virtual keyboard input). It computes metrics such as inter-key interval variability, typing speed bursts, keystroke **entropy** (randomness/unpredictability of key sequences), frequency of specific key types (e.g., printable characters vs. control keys), and distribution of pauses. By comparing these patterns against human physiological norms or learned user-specific profiles, the module assesses the **plausibility of the keystroke pattern**. Unusually consistent or extreme patterns (for instance, typing at a perfectly steady 200 ms cadence for hundreds of characters, or unrealistically high-speed bursts) are flagged as potential indicators of automation.

- **Application Context Correlation Module:** This component cross-references the stream of logged keystrokes with the **foreground application or window in focus**. It determines whether the keystrokes are contextually valid given the active application's state. For example, if the user is purportedly typing text into a word processor or a browser form field, the keystrokes should predominantly consist of character inputs that result in corresponding text on screen. If the active window is a media player or an idle desktop with no text field selected, yet extensive typing is recorded, this module will detect a **context mismatch**. It can also validate shortcut keys or command keys against the application's expected behavior (e.g., detecting if a sequence of keystrokes includes multiple **tab** or **arrow** keys that don't correspond to any navigable UI element, or continuous text input while a dialog box is open that wouldn't accept such input). By ensuring the input makes sense for the focused application (and even the specific UI element in focus, if accessible), the system catches scenarios where a script sends keystrokes to simulate activity that a real user wouldn't typically generate in that context.

- **Screen Content Analysis Engine:** This module uses **optical character recognition (OCR)** and/or computer vision techniques on screen captures or video streams of the user's display to verify that visible changes occur in the **user interface consistent with the keystrokes** being logged. The engine may monitor the appearance of new text characters, changes in a text cursor (caret) position, scrolling events, or GUI element changes at times that correlate with the input. For example, if the keystroke log indicates that the user typed a paragraph of text, the screen content engine should detect the corresponding text appearing in the document or input field on screen. If instead the screen remains static (no new text or UI updates) despite continuous typing, that discrepancy is flagged. The engine can also track the movement of the text insertion caret – e.g., if arrow keys or backspace were pressed, the caret's position or the text content should change accordingly. Advanced implementations might analyze **frame differences** or compute a "heartbeat" of screen activity (small changes vs. large changes) to ensure that a high volume of keystrokes results in proportionate visible feedback. This prevents scenarios where an automation might send inputs that are effectively no-ops or go to a background window (thus invisible), or where the screen is frozen or unresponsive while input is supposedly happening – all signs of something amiss.

- **Facial Recognition and Presence Verification Module:** A camera-based subsystem continuously (or periodically) monitors for the presence of an authorized user's face in front of the device during active sessions. Using facial recognition, the system verifies that the person typing is indeed the expected user (for instance, the enrolled employee or exam-taker) and that they are physically present. If the user's face is not detected for a prolonged period while substantial keyboard activity is recorded, the system suspects automated input or user absenteeism. Likewise, if a face is detected but it does not match the authenticated user's identity (e.g., a different person sitting in), the system flags a potential **identity mismatch**. The facial recognition module can employ **liveness detection** techniques to prevent spoofing (such as requiring occasional movements, blink detection, or 3D depth checks to ensure it's not a photo). This module adds a biometric layer of security, ensuring that even if a script attempts to mimic human-like typing, the absence of a real user present will be noted. In environments like workplace monitoring, this feature aligns with known solutions that log attendance via webcam[9], but here it is directly tied into validating active input events rather than just passively logging.

- **Anomaly Aggregation and Alert Module:** The outputs of the above analysis modules feed into an aggregation engine that synthesizes their findings to make a final determination of suspicious activity. Each module may produce a score or boolean flag indicating the likelihood of anomalous behavior in its domain (timing anomaly, context anomaly, screen anomaly, presence anomaly). The aggregator combines these – for example, via a weighted scoring system or logical rules – to decide when to trigger an alert. **Alerts or log entries** are generated whenever inconsistencies arise beyond configurable thresholds. For instance, the system might tolerate a brief period of fast typing, but if the Keystroke Analysis flags a sustained implausible typing rate *and* the Screen Analysis simultaneously reports a static screen, the aggregator will definitively classify this as synthetic input. Alerts can be in the form of real-time pop-ups to a supervisor, notifications in a security dashboard, or entries in a secure log for later auditing. The alerts include details about what was detected (e.g., "Alert: No on-screen

changes detected during 5 minutes of typing", or "Alert: User absent but keyboard active"). In some embodiments, the system may also take automatic action on severe anomalies – such as pausing further input (to prevent potential damage by automated scripts) or requiring re-authentication of the user – though by default it focuses on **reporting** the event.

The invention can be implemented in software (as a monitoring agent on the user's computer and/or a backend server correlating data from the agent), in firmware, or using dedicated hardware components. It can operate in real-time – analyzing events as they happen to flag issues immediately – or in a batch/offline mode where logs of a session are processed after the fact to produce an analysis report. The detection logic is configurable: administrators can set sensitivity thresholds (e.g., what typing speed is deemed implausible, how long the screen can stay unchanged with continuous input before flagging, how often face checks occur, etc.), and can define rules for ignoring certain benign patterns or for integrating additional data sources (such as network activity, or other biometrics like keystroke signature verification for user ID). The system may also include an **administrative override** or learning period – for example, an admin might mark an alert as a false positive, which the system can learn from to adjust its model (such as learning a particular user's fast typing style or a specific application that generates keystrokes programmatically for legitimate reasons). Conversely, confirmed incidents can train the system to become more sensitive to similar patterns.

In summary, the invention provides a comprehensive solution to detect and deter fake or automated typing by ensuring that *what is happening on the keyboard* is consistently reflected by *what's happening on the screen and in the real world*. By requiring consistency across these independent channels (keystroke device events, application state, visual screen output, and user presence), the system dramatically increases confidence that recorded keyboard activity is genuine. This multi-faceted approach goes beyond existing single-factor monitoring tools, offering stronger safeguards against sophisticated deception attempts.

Brief Description of the Drawings

- **FIG. 1** illustrates an overview of the system architecture for detecting synthetic keystroke activity, showing the various input data sources and analysis modules and how they interconnect to produce alerts.

- **FIG. 2** is a flowchart depicting an example method of operation of the system, outlining the process flow from data collection through various analysis stages to anomaly detection and alert generation.

- **FIG. 3** is a data correlation timeline graph that exemplifies how different signals are cross-referenced over time – in this example, showing keystroke events, screen changes, and face presence status – and highlighting a period where inconsistencies (keystrokes with no screen change and no user present) trigger an alert.

- **FIG. 4** shows an example of a graphical user interface (GUI) for an administrator or security analyst, listing alerts/log entries generated by the system with timestamps and descriptions of the detected anomalies.

Detailed Description

System Architecture (FIG. 1)

Figure 1: System architecture of the keystroke anomaly detection system. In an exemplary embodiment, the system's architecture comprises multiple components that may be implemented as software modules within a monitoring agent on a user's computer, as separate processes, or as part of a centralized server analyzing aggregated data. Referring to FIG. 1, the **data sources** include: (1) a **Keystroke Log** capturing key events (which can be obtained via a low-level keyboard hook, OS event subscription, or a custom keyboard driver that records each physical key press/release along with a timestamp); (2) an **Application Focus Log** (or active window log) indicating which application and window is currently in focus at any given time, including possibly the specific UI element if available (e.g., whether the cursor is in a text field, button, etc.); (3) a stream of **Screen Snapshots**, which could be periodic screenshots or triggered captures (for instance, capturing the screen whenever a certain number of keystrokes have occurred or when a new window is activated, similar to how some monitoring tools take screenshots on user actions[8]); and (4) a **Face Camera Feed** from a webcam or IR camera aimed at the user, supplying images or video frames for presence detection.

These inputs feed into the core **analysis modules**: the **Keystroke Plausibility Analyzer**, the **Contextual Correlation Module**, the **Screen Content Analyzer**, and the **Face Recognition Module**, respectively. Each module processes its input stream and may output an intermediate result or alert signal. For example, the Keystroke Analyzer might output a running score representing how "human-like" or "bot-like" the recent typing behavior is. The Contextual module could output a boolean or score reflecting whether the keystrokes align with the current app (with any detected mismatches noted). The Screen analyzer might produce a measure of screen update activity versus keystroke activity (or even a simple flag when it detects a discrepancy like "typing with no new text"). The Face Recognition module outputs user presence status (e.g., "User present and verified" or "No user" or "Different user detected").

All these outputs are fed into an **Anomaly Aggregator** (which could also be called a decision engine or correlation engine). The aggregator compiles the evidence from all channels. In one simple embodiment, the aggregator may use a set of rules: for instance, if any one module produces a very strong anomaly signal, it raises an alert; or it might require at least two moderate anomaly indications to avoid false positives (e.g., fast typing *plus* no face detected together trigger an alert, whereas fast typing alone might not). In a more advanced embodiment, the aggregator maintains a continuous score or probability of "synthetic activity" using a model (such as a weighted sum or machine learning classifier that takes the module metrics as input features). The **Alert/Log Module** then acts on the aggregator's decision, by logging the event and/or sending out notifications. FIG. 1 shows arrows from each data source into its corresponding analysis module, and from those modules into the central aggregator, which then leads to the alerting mechanism.

The system architecture is designed to be scalable and modular. For example, additional biometric modules could be added (like a typing style recognition module that verifies the identity of the typist by their keystroke dynamics signature, or a mouse activity correlation module to ensure mouse movements correlate with typing breaks). The aggregator can incorporate these easily. The architecture also supports deployment in various setups: all

components could run locally on a user's machine for immediate response (useful for personal security or exam proctoring scenarios), or the data could be sent to a server where heavy analysis (OCR, face recognition, etc.) is performed, which is common in enterprise monitoring systems where an agent collects data and a cloud service processes it.

Detection Flow (FIG. 2)

Figure 2: Flowchart illustrating an example detection method. The operation of the system can be understood through the flowchart in FIG. 2. The process begins when monitoring is activated (e.g., at system login or session start) – **Start Monitoring Session**. In step 1, the system **collects and logs data streams** continuously: keystrokes are logged with timestamps and content, the active application/window is tracked, screenshots are captured at defined intervals or triggers, and camera frames are sampled.

The system then enters an analysis loop that repeats for as long as the session is active. At step 2, the **Keystroke Behavioral Analysis** is performed on the recent batch or window of keystroke data. This could be after each keystroke, or after a short time window (e.g., analyze the last 10 seconds of typing). The analysis calculates metrics such as average inter-key delay, variance in timing, occurrence of bursts (e.g., X keys in Y seconds), and compares these against thresholds or learned models. At decision point 2A ("Implausible Keystroke Pattern?"), the system checks if the metrics exceed the allowed range. If **Yes**, meaning the typing appears non-human (too fast, too regular, etc.), an anomaly flag is raised for the keystroke channel (and optionally the flow could go directly to an alert in a simple implementation). If **No**, it proceeds.

Next, at step 3, the **Contextual Correlation** check is done. The system looks at the active application during those keystrokes and the content of the keys. For instance, if 100 keystrokes occurred while a text editor was focused, that's normal; but if 100 keystrokes occurred while a video player was in full-screen with no text field, that's suspicious. The logic may involve checking if the focused control was a text input or not, and if not, were the keystrokes ones that could still make sense (perhaps global hotkeys or so) or completely out-of-place. Decision 3A ("Context Mismatch Detected?") determines if there's a discrepancy. If **Yes**, a context anomaly flag is raised (and possibly an alert triggered immediately if the system is configured to alert on any single anomaly). If **No**, continue.

At step 4, the **Screen Content Analysis** is performed. This may involve examining the latest screenshot or video frame and comparing it to one from a moment before, or analyzing a series of frames over the period of sustained typing. The system looks for expected changes: new characters in a document, a scrolling operation, a cursor move, opening of menus if function keys were pressed, etc. If the screen appears essentially **static** while a substantial amount of typing was recorded, this is a red flag. Decision 4A ("Screen Inactivity Detected?") checks that condition. If **Yes** (static screen despite input), a screen anomaly flag is set. If **No** (screen changed as expected), proceed.

At step 5, the **Face Recognition/Presence** check is made. The system processes the camera feed to see if a face is currently present and matches the authorized user. This could be done continuously in parallel, but conceptually we include it in the loop to illustrate that at this point the system ensures the user's presence. Decision 5A ("User Absent or Wrong User?") branches

based on the result. If the camera shows that no person is in front of the device (or the face is not the enrolled user) during the time of activity, that yields a presence anomaly flag.

If any of the anomaly flags were raised (any "Yes" in the decisions 2A–5A), the process triggers the **Alert/Log Generation** step (step 6). Here, the system will record the details of the anomaly (time, type, and description as mentioned) and if configured, send out an alert immediately. In a more sophisticated design, the system might not alert on a single minor anomaly; it might accumulate evidence and only alert when a certain confidence or multiple corroborating anomalies occur. This flowchart depicts a straightforward approach where any detected inconsistency leads to an alert for safety. After handling the alert, or if no anomalies were detected in this cycle, the system continues monitoring (step 7: **Continue Monitoring**). The loop then repeats (back to analyzing the next incoming keystrokes and so on) until the session ends (logout or monitoring stopped), upon which the process terminates (End).

This flow ensures that for every moment of user activity, the system is cross-validating behavior across dimensions. Notably, the checks can be done in near real-time. If configured with real-time alerts, a security officer could be notified within seconds of, say, a potential bot starting to type on a user's session. Alternatively, the data could be stored and analyzed in batches (for example, an hourly analysis job) for environments where real-time response isn't needed but post-fact reports are. The flowchart (FIG. 2) is illustrative; actual implementations might rearrange steps (some checks could run in parallel threads) or add additional checks (like network activity correlation as mentioned, or checking for known virtualization/remote control indicators).

Keystroke Behavioral Analysis Module

This section delves deeper into the **keystroke analysis logic** performed by the behavioral module. Human typing behavior inherently contains variability – even skilled typists have small timing variations and characteristic patterns of mistakes, pauses, and bursts. To detect synthetic keystrokes, the system leverages this fact by measuring multiple facets of the keystroke stream: - **Timing Irregularity:** The module computes the distribution of inter-key intervals (time between successive keystrokes). A human's timing distribution is often irregular and follows a biological pattern (for example, some keys might be hit faster following certain other keys, common digraphs in language have shorter intervals, etc.). Automated scripts, unless specifically programmed otherwise, might produce more uniform intervals. The module might calculate the variance or entropy of the inter-keystroke timings. Low variance (nearly equal delays) across a long sequence suggests automation. High entropy is expected for free-text human typing, whereas an extremely predictable pattern suggests a generated input.

- **Typing Speed and Burst Detection:** The module monitors the number of keystrokes over sliding time windows. It can flag if the typing speed exceeds human capabilities (for example, sustained 20 characters per second typing is beyond typical human ability). It also detects unnatural **burst patterns**. A human might type in bursts but will need short breaks (micro-pauses to think or due to physiological limits). If the system sees, for instance, a perfectly steady 5 keystrokes every second for 60 seconds straight, that is highly implausible. Conversely, a bot might also type extremely quickly then stop abruptly. The module can use known statistics of human typing rhythms to set thresholds

for alert – many prior works in keystroke dynamics and anti-bot security establish what "normal" typing rates look like for humans.

- **Key Composition Analysis:** This involves looking at what types of keys are pressed and in what order. Humans tend to predominantly press alphanumeric keys while typing text, with occasional use of shift, backspace, etc. They make mistakes and use backspace, or pause to navigate with arrow keys occasionally. If a sequence contains an inordinate repetition of a single key or a pattern that doesn't fit normal text (e.g., the keys correspond to moving a character in a game in a perfectly repeating pattern, or a function key pressed every 10 characters regularly), it could indicate a programmed input. The module might track the frequency of special keys (e.g., 50% of keys being "Down Arrow" is odd unless the user is scrolling a page, which should coincide with that context).

- **Language Model or Randomness Check:** Optionally, if the keystrokes are associated with text input, the system can run a check on the resulting text. Humans typing in natural language will produce text that (aside from typos) fits language patterns, whereas a bot might generate gibberish or perfectly random strings if trying to fool a simple activity monitor. The module could do a crude entropy or dictionary check on the typed content: extremely high randomness in the output text (or conversely, an unusual absence of any errors over a very long text) could be a flag. (This is more relevant in scenarios like detecting if someone is replaying a pre-recorded perfect typing script versus actually composing text.)

The Keystroke Analysis Module may maintain a rolling buffer of recent keystrokes and continuously update a risk score. It can be calibrated per user as well: for instance, if a particular user is known to type 120 WPM with bursts, the system can adapt to that baseline to avoid false positives, while still catching anything far out of ordinary for that user.

Contextual Correlation Module

The Contextual Correlation Module ensures that **what the user is purportedly typing** makes sense given **where they are typing it**. It operates by accessing the system's current foreground application and, if possible, the control element within that application: - At a basic level, the module knows the name or type of the active application (e.g., Microsoft Word, a web browser on a specific URL, a terminal/console, etc.) and the timing of focus changes. The keystroke log entries are tagged with the application that was in focus at the time (the provided user logs demonstrate such tagging, with entries like "traceloop.superherobio.com" as the active webpage during certain keystrokes, or code.exe indicating a code editor was active【7†】). Using this, the module can segment keystrokes by application context.

- The module has a knowledge base of **expected input behavior** for common applications. For example, in a text editor or word processor, continuous text character input is normal. In a spreadsheet, one might see bursts of typing separated by tab or enter (moving between cells). In a web browser, if the active element isn't a text box, continuous typing might trigger hotkeys or be ignored. The module can detect if keystrokes would have any effect. If an app is running that generally doesn't involve typing (say a video call or a game menu open), yet we see a lot of alphanumeric input, that's a context red flag.

- More granularly, on certain platforms the module can query the OS for the currently focused UI control (for instance, Windows UI Automation or accessibility APIs can often tell if the focus is on a textbox vs. a button). If available, the module uses that: e.g., if focus is on a password field that typically accepts only short input, but we see a thousand characters typed, suspicious. If focus is on a menu or a button (which wouldn't accept keystrokes for typing), any letters typed are strange (perhaps they might trigger menu accelerators at best). The module may consider some keys "benign" in certain contexts (like pressing Alt+Tab to switch windows – those are system keys, not app input, and indeed our system might exclude system shortcuts from analysis or treat them differently).

- The correlation also works in reverse for verifying expected output: if the user switched apps (focus changed) and right after that a bunch of keystrokes occur, normally those keystrokes should pertain to the new app. If the keystrokes log still shows keys going to the old app (which is now unfocused), something is off – possibly an error in logging or some virtual input directed to a background window, which might be malicious (like a hidden keylogger test or some script sending keys to an off-screen window).

This module essentially cross-checks the **who/what/where** of the input: each keystroke event should be attributable to a real user action in the interface. If not, the system captures that discrepancy. It can output messages like "Typed input while no text field active in Application X" or "Keystrokes recorded in non-interactive context".

Screen Content Analysis Engine

The Screen Content Analysis Engine serves as the eyes of the system, verifying the **visual outcome** of inputs. It uses techniques from image processing and OCR: - For text input, OCR can be applied to screenshots to read what text is visible. If, over the last minute, 100 characters were typed, the engine could detect if approximately 100 new characters appear in the text area on screen. Minor mismatches might occur due to typos corrected, etc., but no appearance of new text at all would be glaring. Modern OCR can handle GUI text reasonably, especially if the font is standard; moreover, if the text is typed in a known application, sometimes the system might not need full OCR – it could potentially get text content via an API. However, to keep it general, OCR is a robust method (works even for remote VDI sessions where the monitoring agent might only see pixels).

- For non-textual actions (like pressing arrow keys, function keys, etc.), the engine looks for corresponding UI reactions. Did a menu open? Did a selection change? Did the screen scroll? Computer vision can detect changes between consecutive frames. If pixel differencing reveals virtually no change, and certainly no systematic change (like a caret moving or a highlight moving), then presumably the keystrokes did not lead to any visible result.

- The engine can track the blinking text cursor (caret). Typically, when you type, the caret moves forward with each character. If the caret stays fixed in one spot for a long time while characters are supposedly being typed, it implies those characters might not actually be going to the screen (perhaps they are being intercepted or just simulated

without a UI component). One way to do this is by image template matching to find the caret in each screenshot and see if its position moves or not.

- The combination of OCR and visual change detection also helps filter out false positives from the keystroke module. For example, maybe the user is typing in a remote desktop session window – to our local system, it might look odd (like "nothing changed in the local app except some network packets") but the screen analysis would actually catch that within the remote desktop window, the text did change. So it provides context.

One challenge this module addresses is scenarios where someone might write a script that generates keystrokes which **do** produce some screen change but not what a real user would do. For example, a script could open Notepad in the background and type gibberish just to create the appearance of activity to naive monitoring (some basic monitoring tools might just check "did the user press keys? yes, then they are active"). Our system's screen analyzer, in conjunction with context, would note if that Notepad is not actually the foreground window (context mismatch) or if it is open off-screen with no real user reading it (face absent). Even if the script tries to bring an application forward and type, the combination of all modules (the face module seeing no one looking, the keystroke pattern possibly being robotic, etc.) works to catch that.

Facial Recognition and User Presence Verification

The **facial recognition sub-system** works in tandem with the other modules to ensure a human presence. In practice, this involves using a camera (webcam) and running face detection continuously. When a face is found, the system matches it to the enrolled user's face data (which could be done via local facial recognition models or by sending to a server for verification, depending on privacy requirements).

Key points about this module: - It typically runs at a lower frequency than keystroke logging (maybe a frame every second or every few seconds is analyzed, or it triggers when anomalies are detected to conserve resources). If it loses sight of the face (e.g., user moved away or camera covered), it starts a timer. If the absence persists beyond a threshold (say 30 seconds of continuous typing with **no face**), it flags it. The assumption is that a human can't type without being in front of the keyboard (at least not for long, unless some remote control hardware is being used, which would itself likely be flagged by context or keystroke anomalies).

- If a face is detected but it's **not the authorized user**, the system can either immediately flag (security breach – someone else is using the session) or if this is in a more open scenario, log it as an event (for example, two people using one account). But in the context of our problem (detecting synthetic or simulated activity), an unauthorized face might mean someone other than the expected user is performing the activity – which could still be "real" activity by a human, but not by the right person (a different kind of policy violation). We include it because the user asked for verifying identity as well.

- The integration of presence with input monitoring is somewhat novel. Traditional use of webcams in monitoring is for time tracking or occasional snapshots for verification. Here, we actively correlate it: if all else looks fine (keys and screen match up) but the person is not there, something is wrong (maybe a device or an insider is generating input remotely). Conversely, if the face is there but everything else is off (like the face is

present but the typing is clearly fake), it could be a scenario of a person supervising a bot or trying to trick the system by just being present.

- The facial recognition module can also serve as continuous authentication. If at some point the face doesn't match the logged-in user, the system might pause input or require password re-entry (some secure systems do that – lock the screen if the user's face is not periodically confirmed). That's an optional action beyond just logging an anomaly.

Anomaly Detection and Alerting

After each of the specialized modules has done its analysis, the system must decide when to formally declare an **anomaly** and issue an alert or log entry. The design of this decision logic can greatly affect sensitivity and specificity: - One simple approach: **Any single anomaly triggers an immediate alert.** This is useful for high-security environments. For example, if the face disappears but typing continues, alert immediately; or if screen doesn't change for X seconds of typing, alert. However, this could result in more false alarms (maybe the user looked away from the screen briefly or minimized a window causing OCR to fail momentarily).

- A refined approach: **Require multiple conditions or a persistent condition.** For instance, a few mistimed keystrokes alone might not alert, but if combined with another symptom, then alert. Or require the anomaly to persist for a certain duration (e.g., at least 10 seconds of continuous discrepancy) before alerting, to filter out momentary glitches.

- **Scoring system:** Each module could output a score 0-100 (0 = normal, 100 = definitely fake). The aggregator might sum or weight these. Perhaps keystroke pattern anomaly score + context anomaly score + screen anomaly score + face anomaly score > 100 triggers alert. This allows trade-offs; e.g., if face is missing (score 50) and keystrokes are slightly odd (score 60), total 110 > 100 triggers; whereas if only face is missing (50) but everything else is normal (0s), no trigger (just maybe a warning logged).

The **alerts** themselves contain useful info for the administrator or system. Typically, an alert entry would include: - Timestamp of detection. - Which module(s) flagged the anomaly (possibly an code or color: e.g., "Behavioral anomaly, Screen anomaly"). - A description: e.g., *"Implausible typing rate of 900 keystrokes/min sustained for 2 min"*, or *"Keystrokes detected while user 'Alice' not present"*, *"Input detected in app Notepad but no corresponding screen changes"*, etc. - Potential severity level (some anomalies might be low severity if just one minor thing was off; multiple issues = high severity).

The system can present these alerts in a **dashboard or log file**. In FIG. 4, we illustrate a simple example of how such an alert log might appear to an administrator.

Figure 4: Example administrative alert interface showing logged anomaly events. In this illustrative GUI, each row represents an alert event with the time, type of anomaly, and details. As shown in FIG. 4, at 12:34:56 PM an "Implausible Typing" alert was logged because the user's typing rate hit an unrealistic 250 keystrokes per minute (this could indicate a bot or a copy-paste burst being recorded as rapid keystrokes). A minute later at 12:35:02 PM, a "Screen Static" alert was generated, indicating that no on-screen changes were detected despite continuous keystroke input – suggesting that the input might not have been truly affecting the UI (or the activity was happening off-screen/in the background). At 12:35:05 PM, a "Face Absent"

alert shows the system did not detect the user's face while activity was ongoing, and a "Context Mismatch" alert indicates keys were being pressed in a context where they shouldn't have any effect (perhaps an inactive window or a field that wasn't selected for text input). In a real system, these events might be correlated (the system could combine them into one incident report since they all occurred around the same time, likely related to the same cause). The GUI might allow clicking an event to see more details or view the screenshot taken at that time, etc.

Such a user interface enables quick review of suspicious incidents. An admin could see at a glance that around 12:34-12:35, multiple anomalies co-occurred, strongly indicating synthetic activity – warranting investigation or an immediate action like contacting the user or terminating the session. The system could also send automated emails or messages for urgent alerts, especially if an ongoing security breach is detected (for example, if this was a server and someone remotely injected keystrokes to run commands, an alert would let the security team respond in real time).

Administrative Controls and Overrides: Because every work environment and user can be different, the system preferably provides an admin console to adjust parameters. For example, the threshold for "implausible typing" can be tweaked if one has an extremely fast typist on the team. The types of keys to treat as context violations can be configured per application (maybe in a coding IDE, it's normal to have weird sequences due to keyboard shortcuts – the admin might lower the sensitivity for that context). Admins might also temporarily disable certain checks for specific users or during certain time windows (perhaps if maintenance tasks or automated scripts are expected to run, they can pause alerts to avoid noise).

Real-Time vs Batch Operation: The invention covers both scenarios. In real-time mode, the modules analyze on the fly and an alert can pop up within seconds of the event. In batch mode, the data (keystroke logs, application logs, screenshots, video) might be uploaded and analyzed overnight or on-demand. Batch mode might be useful for auditing purposes – for example, at the end of each day, analyze the logs to see if any portion of the day had anomalies (maybe the user was mostly fine but for one hour there was strange activity). The system's description covers both possibilities, and dependent claims (as will be seen) detail these variations.

Integration with Other Systems: This detection system can be integrated into larger frameworks. For instance, it could feed into a **security incident and event management (SIEM)** system in an enterprise, correlating with network logs. If someone tries to exfiltrate data and also you see synthetic typing, it could be part of a bigger attack pattern. Or in remote exam proctoring software, this system's alert could trigger a human proctor to more closely observe or to invalidate the exam session.

In conclusion, the detailed design of the system ensures that any attempt to "game" user activity by focusing on just one aspect (like tricking a basic keylogger) will be caught by another aspect (lack of screen change or user presence). It provides a **comprehensive solution** by not relying on a single indicator. The approach is proactive – rather than waiting to discover after the fact that logs were falsified or an employee wasn't truly working despite logged keystrokes, it catches the discrepancy as it happens. This is increasingly important in an era of remote work and sophisticated automation, where ensuring the legitimacy of user actions is critical for both security and productivity monitoring.