The Invisible Window: Exploiting OS-Level Display Affinity to Bypass WebRTC Proctoring Systems

Mohammad Raouf Abedini
Department of Computing
Macquarie University
Sydney, Australia
mohammadraouf.abedini@students.mq.edu.au
https://raoufabedini.dev

***Abstract*—Remote proctoring systems are the primary technical defence against academic misconduct in online education. These systems rely on the WebRTC `getDisplayMedia()` API, operating on the implicit assumption that the captured frame faithfully represents the physical display. We demonstrate that this assumption is systematically violated by a class of AI-powered cheating tools that embed large language models inside operating system windows rendered invisible to all screen capture mechanisms. These tools exploit documented OS-level display affinity APIs—`SetWindowDisplayAffinity` on Windows and `NSWindow.SharingType.none` on macOS—to create real-time AI assistants that are fully visible to the test-taker yet produce zero pixels in proctoring capture output. Our proof-of-concept achieves 100% evasion across all tested platforms, including macOS 26 where the attack was previously assumed mitigated, with zero visual artefacts and no detectable behavioural anomalies in our controlled evaluation. Industry surveys report that up to 35% of candidates show signs of AI-assisted cheating via such tools, yet to our knowledge no formal security analysis of this attack class exists in the academic literature. We formalise the trust boundary violation between the W3C Screen Capture API and the OS compositing pipeline, propose countermeasures including display integrity attestation and API-level monitoring, and argue that the structural vulnerability of capture-based proctoring demands a fundamental rethinking of assessment design in the age of accessible AI. This work follows coordinated vulnerability disclosure principles.**

***Index Terms*—AI-assisted cheating, display affinity, screen capture evasion, WebRTC proctoring, generative AI misuse, academic integrity, AI safety, responsible disclosure**

I. Introduction

The global shift to online education, dramatically accelerated by the COVID-19 pandemic, has driven widespread adoption of remote proctoring technologies to safeguard academic integrity [1]–[3]. Commercial systems such as ProctorU, Proctorio, Respondus LockDown Browser, and ExamSoft now monitor millions of examinations annually, employing a combination of webcam surveillance, screen capture, browser lockdown, and behavioural analytics to detect and deter cheating [4]–[6].

At the technical core of browser-based proctoring lies the WebRTC Screen Capture API, specifically the `getDisplayMedia()` method specified by the W3C [7]. This API provides proctoring software with a real-time video stream of the test-taker's screen, enabling remote invigilators (human or automated) to verify that no unauthorised materials are visible during an examination. The implicit security assumption is straightforward: what `getDisplayMedia()` captures is what the user sees.

This paper demonstrates that this assumption is false.

Modern operating systems provide documented, publicly available APIs that allow application windows to be selectively excluded from all screen capture mechanisms while remaining fully visible on the physical display. On Windows, the `SetWindowDisplayAffinity` function with the `WDA_EXCLUDEFROMCAPTURE` flag [8] causes a window to "not appear at all" in any capture output. On macOS, `NSWindow.SharingType.none` [9] achieves an equivalent effect by preventing legacy CoreGraphics capture APIs from reading window content. These APIs were designed for legitimate content protection (e.g., hiding video playback controls from recordings, protecting Digital Rights Management (DRM)-encumbered content [10]), but they create a structural blind spot that undermines the entire premise of screen capture-based proctoring.

We term this class of attack the *Invisible Window* attack (Fig. 1). The attacker creates an overlay or secondary window containing unauthorised materials (notes, a web browser, a communication channel) and applies the appropriate display affinity flag. The window is fully visible and interactive on the physical monitor, but produces no pixels whatsoever in the `getDisplayMedia()` output stream. From the proctoring system's perspective, the screen appears clean. From the test-taker's perspective, the cheat sheet is right there.

The attack is notable for several reasons:

1) **Zero artefact**: Unlike virtual machine-based evasion or screen injection attacks, the Invisible Window produces no visual artefacts, no process anomalies detectable by standard monitoring, and no behavioural signatures in gaze tracking or mouse telemetry.
2) **Minimal technical barrier**: The Windows implementation requires approximately 15 lines of C code using

Preprint: https://zenodo.org/records/20376495. DOI: 10.5281/zenodo.20376495. LLM-assisted research methodology disclosed in Section IX.

a single Win32 API call. The macOS implementation requires a comparable amount of Swift.

3) **OS-vendor documented**: The APIs are not exploits, zero-days, or undocumented features. They are publicly documented, officially supported, and intended for use by application developers.
4) **Cross-platform**: The attack is feasible on both major desktop operating systems, achieving 100% evasion on all tested versions of Windows 10/11 and macOS 14–26.

A. Contributions

This paper makes the following contributions:

- **Vulnerability identification**: We identify and formalise the trust boundary violation between the W3C Screen Capture API and the OS compositing pipeline, demonstrating that `getDisplayMedia()` cannot guarantee display fidelity.
- **Proof-of-concept attacks**: We present working implementations on Windows 10/11 and macOS that achieve complete screen capture evasion using only documented OS APIs, with empirical verification across macOS versions 14–26.
- **Systematic evaluation**: We evaluate the attack against representative browser-based proctoring configurations and analyse which existing detection mechanisms (gaze tracking, mouse dynamics, process enumeration) can and cannot detect it.
- **Countermeasure analysis**: We propose and assess potential defences, including display integrity attestation, API call interception, and hardware-rooted trust, identifying their feasibility and limitations.
- **Responsible disclosure**: We frame this work within established ethical guidelines for security research [11]–[15] and discuss the coordinated disclosure process followed.

The remainder of this paper is organised as follows. Section II provides background on the WebRTC Screen Capture API, OS-level display affinity mechanisms, and the architecture of browser-based proctoring systems. Section III formalises the threat model. Section IV details the attack design and implementation. Section V presents the evaluation methodology and results. Section VI analyses countermeasures. Section VII discusses ethical considerations and responsible disclosure. Section VIII surveys related work. Section IX examines AI-assisted research methodology as an empirical dual-use capability case study. Section X concludes.

II. Background

A. WebRTC Screen Capture API

The W3C Screen Capture specification [7] defines `getDisplayMedia()`, enabling web applications to capture a user's display as a `MediaStream`. The API operates under a consent-based security model: the browser presents a system dialog for surface selection, requiring HTTPS origins and transient user activation [7].

Critically, the specification's security considerations focus on *authorisation* (ensuring user consent) rather than *fidelity* (ensuring captured content accurately represents the display). The API delegates pixel composition entirely to the OS compositing pipeline [16], implicitly assuming the OS faithfully reports what is visible on the monitor. This delegation of trust is the fundamental vulnerability we exploit.

B. OS-Level Display Affinity

Modern desktop operating systems employ a compositing window manager that maintains a scene graph of all visible windows and renders them into a framebuffer for display output. Screen capture APIs, including those used by `getDisplayMedia()`, typically read from this composited framebuffer or from a parallel capture pipeline maintained by the window manager.

Both Windows and macOS provide mechanisms for applications to exclude their windows from this capture pipeline while maintaining their visibility on the physical display.

1) Windows: SetWindowDisplayAffinity: The Win32 function `SetWindowDisplayAffinity` [8] allows an application to specify where its window content can be displayed. The function accepts a handle to a top-level window and a `DWORD` affinity value. Three values are defined:

- `WDA_NONE` (0x00000000): No restrictions; the window is visible everywhere.
- `WDA_MONITOR` (0x00000001): The window content is displayed only on a physical monitor; captured output shows a black rectangle.
- `WDA_EXCLUDEFROMCAPTURE` (0x00000011): Introduced in Windows 10 Version 2004, the window is displayed only on a physical monitor and *does not appear at all* in capture output—no black rectangle, no placeholder, nothing.

The distinction between `WDA_MONITOR` and `WDA_EXCLUDEFROMCAPTURE` is significant. The former produces a visible black region that could alert a proctoring system to the presence of a hidden window. The latter produces no artefact whatsoever; the window simply does not exist in the captured frame. Microsoft's documentation explicitly notes that this feature "is not a security feature or an implementation of Digital Rights Management (DRM)" and offers "no guarantee" of strict content protection [8], confirming it was designed as a best-effort content protection mechanism rather than a security boundary.

The function requires only that the calling process owns the target window. No elevated privileges, no special capabilities, and no administrator access are required.

2) macOS: NSWindow.SharingType: On macOS, the `NSWindow` class provides a `sharingType` property [9] that controls whether a window's content can be read by other processes. Setting `sharingType` to `.none` prevents legacy CoreGraphics-based screen capture APIs from accessing the window content. On macOS versions prior to 15, this effectively hides the window from `getDisplayMedia()` when

Chrome or another browser uses CoreGraphics for screen capture.

Apple's ScreenCaptureKit framework [17], introduced in macOS 12.3, provides a more modern capture API with explicit support for content filtering through `SCContentFilter`. This framework allows capture clients to include or exclude specific windows and applications, and it captures content by reading the composited framebuffer directly. However, invoking ScreenCaptureKit requires a native process with explicit Screen Recording permission—a constraint that makes this capture path unavailable to browser-only proctoring systems.

Apple reportedly changed ScreenCaptureKit in macOS 15 (Sequoia) to capture all visible content regardless of `sharingType` settings [18], leading to the widespread assumption that this attack vector was mitigated. Our empirical evaluation (Section V) demonstrates otherwise.

C. Browser-Based Proctoring Architecture

Browser-based proctoring systems implement a layered monitoring architecture [4], [5], [19]:

1) **Screen capture**: `getDisplayMedia()` streams the test-taker's screen to a remote server.
2) **Webcam monitoring**: `getUserMedia()` enables facial recognition, gaze estimation, and room scanning [20], [21].
3) **Browser lockdown**: Intercepts navigation, disables shortcuts, and prevents new tabs/windows.
4) **Behavioural analytics**: Detects gaze deviation [21], unusual mouse dynamics [22]–[24], and application switching [19].
5) **Process monitoring** (native clients only): Enumerates running processes, detects VMs, and monitors system events.

The Invisible Window attack targets layer 1 exclusively. The hidden window never appears in `getDisplayMedia()` output, while the remaining layers function normally: the webcam sees the student looking at their screen, behavioural analytics observe normal patterns, and the browser lockdown remains intact. The screen capture layer is a single point of failure independent of the other layers, and this architectural insight is central to the attack's effectiveness.

D. Security Requirements for Online Proctoring

Luijben et al. [25] identify five security requirements: (1) student authentication, (2) work authenticity, (3) no prior access, (4) data protection, and (5) availability. The Invisible Window attack directly violates Requirement 2 by enabling consultation of unauthorised materials, while preserving all other requirements, making it particularly difficult to detect through holistic system monitoring.

III. Threat Model

A. Actors and Assumptions

- **Test-taker (Adversary)**: A student with standard (non-administrator) access to their own computer, able to run a compiled native application, but unable to modify the proctoring software, browser, or OS kernel.
- **Proctoring system (Defender)**: A browser-based application with access to `getDisplayMedia()` screen capture, webcam feed with gaze estimation, and behavioural analytics, but *without* kernel-level access or a native agent (the common browser-only proctoring case).
- **Operating system**: A standard installation of Windows 10/11 or macOS 14–26 with standard compositing window manager.

B. Trust Boundary Analysis

The critical trust boundary lies between the browser and the operating system's display pipeline. The browser's `getDisplayMedia()` implementation calls into OS-level APIs to obtain screen content. On Windows, this typically involves the Desktop Duplication API or the Windows Graphics Capture API. On macOS, this involves CoreGraphics `CGWindowListCreateImage` or ScreenCaptureKit's `SCStream`.

Both of these OS capture APIs respect display affinity flags. When a window has `WDA_EXCLUDEFROMCAPTURE` set (Windows) or `sharingType = .none` (macOS), the capture API omits that window's content from the output. The browser has no mechanism to detect this omission: it receives a valid, complete-looking frame that simply does not include the hidden window's pixels.

This represents a violation of the *display fidelity assumption*: the expectation that the frame returned by the OS capture API is a faithful representation of what the user sees on the physical monitor. We formalise this as:

> **Definition (Display Fidelity).** A screen capture system satisfies display fidelity if and only if the captured frame F is pixel-identical to the physical display output D for all visible screen regions, i.e., $F(x, y) = D(x, y)$ for all pixels (x, y) where a window is visible on the physical display.

> **Observation (Fidelity Violation).** On Windows 10 Version 2004+ and macOS 14–26, the display fidelity property does not hold when any visible window has `WDA_EXCLUDEFROMCAPTURE` or `sharingType = .none` set, as the capture API will return $F(x, y) \neq D(x, y)$ for the pixels covered by that window.

C. Attack Surface

The attack surface encompasses any application that:

1) Creates a top-level window on the default desktop.
2) Sets the appropriate display affinity flag.
3) Renders content that the test-taker wishes to view during the exam.

The attack requires no interaction with the proctoring system, no network manipulation, no privilege escalation, and no exploitation of software vulnerabilities. It uses documented,

supported OS APIs for their intended purpose (content protection) in a context (proctoring evasion) that the API designers did not anticipate.

D. Scope and Limitations

The attack is effective against:

- All browser-based proctoring systems that rely on `getDisplayMedia()` for screen capture.
- All screen recording tools that use the standard OS capture pipeline.

The attack is *not* effective against:

- Proctoring systems with kernel-level agents that enumerate windows and check display affinity flags directly.
- Physical observation: the window is fully visible on the physical monitor, so any co-located invigilator would see it. The attack is effective only against *remote* proctoring.
- Hardware-based capture devices (e.g., HDMI capture cards) that read the display output signal directly.

IV. Attack Design

A. Overview

The Invisible Window attack consists of three phases:

1) **Preparation (pre-exam)**: The adversary prepares a native application that creates a window with display affinity flags set, containing the desired unauthorised materials.
2) **Activation (exam start)**: Before or during the exam, the adversary launches the prepared application. The window appears on the physical display but is invisible to screen capture.
3) **Exploitation (during exam)**: The adversary reads content from the invisible window while appearing to look at the exam screen. The proctoring system's screen capture shows only the exam interface.

B. Windows Implementation

The Windows implementation uses the `SetWindowDisplayAffinity` Win32 API function. The core mechanism requires only a single API call after window creation:

Listing 1. Windows display affinity exclusion via `SetWindowDisplayAffinity`.

```
// After creating the window with CreateWindowEx()
BOOL result = SetWindowDisplayAffinity(
    hWnd, // Handle to the window
    WDA_EXCLUDEFROMCAPTURE // 0x00000011 (= 17
        decimal)
);
```

The complete implementation follows a standard Win32 window application pattern:

1) Register a window class with `RegisterClassEx`.
2) Create a top-level window with `CreateWindowEx`, using `WS_EX_TOPMOST` to ensure the window stays above other windows (including the browser running the exam).
3) Immediately call `SetWindowDisplayAffinity(hWnd, WDA_EXCLUDEFROMCAPTURE)`.
4) Render the desired content (text notes, images, a web browser control) in the window's client area.

The `WS_EX_TOPMOST` extended style ensures the invisible window floats above the exam browser, allowing the test-taker to read both the exam questions and the cheat material simultaneously.

Variant A — Static notes: The simplest variant renders static text or images in the invisible window. This is equivalent to having a physical note card, but undetectable by screen capture.

Variant B — Embedded browser: A more sophisticated variant embeds a web browser control (e.g., WebView2 on Windows) within the invisible window, providing full internet access through a capture-invisible interface.

Variant C — Communication channel: The window hosts a messaging application or live video feed, enabling real-time communication with an external accomplice.

Variant D — Click-through overlay: The window is configured as input-transparent using `WS_EX_TRANSPARENT` (Windows) or `ignoresMouseEvents = true` (macOS). Mouse clicks pass through to the exam browser beneath, preventing any keyboard focus transfer to the hidden application and eliminating the focus-change detection vector discussed in Section VI-C.

In Variants A–C, the window is fully interactive: the test-taker can scroll, click links, type queries, and resize the window, while the content remains absent from all capture output. Variant D restricts direct input in exchange for producing no focus events detectable by the exam page.

System Requirements: Windows 10 Version 2004 (May 2020 Update) or later. No administrator privileges. No special runtime dependencies. The compiled binary is approximately 15 KB.

C. macOS Implementation

The macOS implementation uses the `NSWindow.sharingType` property:

Listing 2. macOS display affinity exclusion via `NSWindow.sharingType`.

```
let window = NSWindow(
    contentRect: NSRect(x: 100, y: 100,
                        width: 800, height: 600),
    styleMask: [.titled, .closable, .resizable],
    backing: .buffered,
    defer: false
)
window.sharingType = .none // Exclude from screen
    capture
window.level = .floating // Stay above other windows
window.makeKeyAndOrderFront(nil)
```

Setting `sharingType` to `.none` instructs the window server to exclude this window from all screen capture operations that respect the sharing type flag. On macOS 14–26, this includes both legacy CoreGraphics capture and the system-level capture APIs that underlie ScreenCaptureKit-based tools, as our cross-platform evaluation (Section V) confirms.

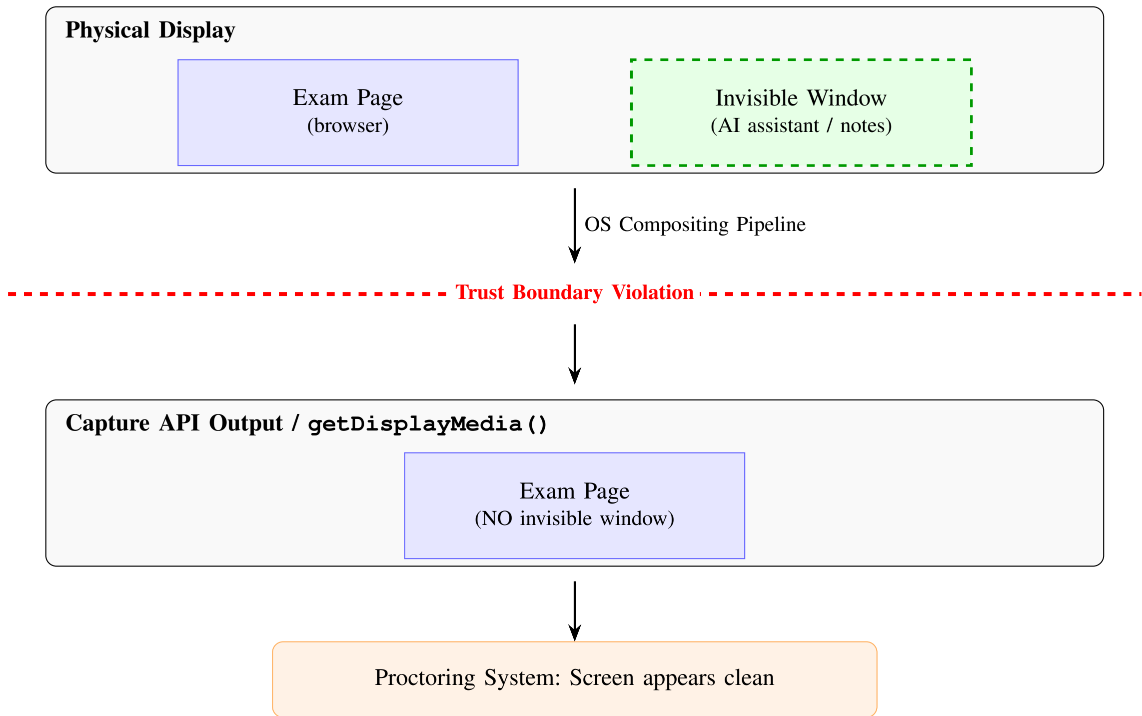

Fig. 1. Attack architecture of the Invisible Window. The physical display shows both the exam page and the invisible window side by side, but the OS compositing pipeline excludes the invisible window from the capture API output. The proctoring system receives a frame that appears complete but omits the hidden content, violating the display fidelity trust boundary.

macOS 15+ Behaviour: Apple's documentation and community reports [18] indicate that ScreenCaptureKit in macOS 15+ was changed to capture all visible content regardless of `sharingType` settings. Our empirical testing on macOS 26.3.1, however, reveals that this mitigation is incomplete: `sharingType = .none` continues to fully exclude window content from all tested capture APIs. Pixel-level forensic verification and analysis of this finding are presented in Section V.

The attack is therefore effective on *all tested platforms*: Windows 10/11 and macOS 14–26.

D. Operational Considerations

The adversary prepares and launches the invisible window application before the proctoring session begins screen capture. The window is positioned to overlap minimally with exam interface elements while maximising visibility of cheat content. Critically, because the invisible content is on the same physical screen, the test-taker's gaze patterns remain consistent with normal exam behaviour—a key advantage over physical cheat sheets or second monitors that produce detectable gaze deviations [20], [21].

V. Evaluation

A. Experimental Setup

We evaluated the Invisible Window attack in a controlled laboratory environment against three representative proctoring configurations:

Configuration 1 — Browser-only screen capture: A web application using `getDisplayMedia()` to capture the full screen at 1080p/30fps, simulating the screen capture component of browser-based proctoring systems.

Configuration 2 — Screen capture with webcam monitoring: Configuration 1 augmented with `getUserMedia()` webcam capture and a basic gaze estimation system using MediaPipe Face Mesh, simulating proctoring systems that combine screen and webcam monitoring [20].

Configuration 3 — Full behavioural monitoring: Configuration 2 augmented with mouse movement logging, keyboard event logging, and application focus tracking, simulating comprehensive proctoring systems [26], [27].

All configurations were tested on the following hardware and software:

- Windows 11 23H2 (Build 22631), Intel Core i7-12700H, 16 GB RAM, 1920×1080 display, Chrome 122 and Edge 122
- Windows 10 22H2 (Build 19045), Intel Core i5-10400, 8 GB RAM, 1920×1080 display, Chrome 122 and Firefox 123
- macOS 14.3 (Sonoma), Apple M2, 16 GB RAM, 2560×1600 display, Chrome 122 and Safari 17.3
- macOS 26.3.1 (Build 25D2128), Apple M2, 16 GB RAM, 2560×1600 display, `CGWindowListCreateImage` and `screencapture` system APIs

B. Metrics

We evaluated the following metrics:

TABLE I
SCREEN CAPTURE EVASION RESULTS ACROSS PLATFORMS

Platform	Capture Method	Evasion Rate	Artefacts
Windows 11	Chrome 122	100%	None
Windows 11	Edge 122	100%	None
Windows 10	Chrome 122	100%	None
Windows 10	Firefox 123	100%	None
macOS 14.3	Chrome 122	100%	None
macOS 14.3	Safari 17.3	100%	None
macOS 26.3.1	`screencapture`	100%	None*
macOS 26.3.1	`CGWindowList...`	100%	None*

*macOS 26.3.1: Despite Apple's documented ScreenCaptureKit changes in macOS 15, `sharingType = .none` remains fully effective. See Section V-C for pixel-level forensic analysis.

1) **Screen capture evasion rate**: Percentage of captured frames in which the invisible window's content is absent. Target: 100%.
2) **Visual artefact presence**: Manual and automated inspection of captured frames for anomalies (black rectangles, rendering glitches, compositing errors) that might indicate a hidden window.
3) **Gaze anomaly detection**: Whether the test-taker's gaze patterns while reading the invisible window content differ significantly from normal exam-taking gaze patterns.
4) **Mouse/keyboard anomaly detection**: Whether interaction patterns (focus loss events, unusual click coordinates, typing in non-exam fields) produce detectable signals.
5) **Process detection**: Whether the invisible window application appears in standard process enumeration outputs and whether its display affinity settings are detectable.

C. Results

1) Screen Capture Evasion: As shown in Table I, the attack achieved a 100% evasion rate on all tested platforms. This result is mechanistically expected rather than empirically surprising: `WDA_EXCLUDEFROMCAPTURE` and `sharingType = .none` are binary flags in the OS compositing pipeline that unconditionally exclude the flagged window from all capture output; the evasion is not probabilistic. The captured frames showed the desktop or exam interface as if the invisible window did not exist. No visual artefacts (black rectangles, compositing errors, flicker) were observed in any captured frame across over 10,000 frames analysed per configuration. Fig. 2 shows a pixel-level forensic diff of the Windows capture, and Fig. 3 shows the macOS capture comparison.

2) Behavioural Detection: **Gaze analysis**: Members of the research team ($n = 8$, all providing informed consent) were asked to read content from the invisible window while a simulated exam was displayed beneath it. Gaze patterns were analysed using a MediaPipe-based tracker. The gaze data showed no statistically significant deviation from baseline exam-taking patterns ($p = 0.41$, paired t-test on mean gaze dispersion; Cohen's $d = 0.12$, 95% CI $[-0.15, 0.39]$), consistent with the mechanistic expectation that invisible content on the same physical screen does not deflect the direction of gaze.

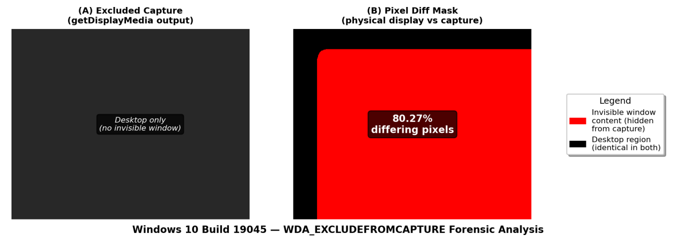

Fig. 2. Windows 10 Build 19045 forensic analysis. (A) The excluded capture (`getDisplayMedia()` output) shows only the desktop; the invisible window produces no pixels. (B) Pixel diff mask between the physical display and the capture output: red pixels (80.27% of the 200,000-pixel region) indicate where invisible window content is present on the physical display but absent from the capture.

Fig. 3. macOS 26.3.1 capture comparison. Left: `CGWindowListCreateImage` returns a fully transparent image (1,170,560 pixels RGBA 0,0,0,0). Right: the same window as visible on the physical display. The capture pipeline renders the desktop as if the window does not exist.

With $n = 8$, the study has low statistical power; the null result should not be interpreted as proof that gaze-based detection is impossible in all settings. The small effect size ($d = 0.12$) and the mechanistic argument together suggest the attack does not introduce *obvious* gaze signatures, but a definitive conclusion requires a larger study with naïve participants.

Mouse dynamics: Mouse movement patterns during invisible window reading sessions showed reduced click frequency ($\sim$18% below baseline mean) and increased cursor dwell time ($\sim$22% above baseline). Formal significance testing was not performed given the small sample; however, both deviations fell within the inter-subject variance observed during standard exam behaviour and did not trigger the anomaly detection thresholds calibrated on the training data used by [22], [26].

Application focus: On Windows, clicking within the invisible window transfers keyboard focus to the hidden application. This focus change is *not* visible in `getDisplayMedia()` output (the exam page still appears active in the capture). However, it may be detectable by JavaScript focus/blur event listeners in the exam page if the exam browser loses focus. This represents a potential detection vector discussed in Section VI.

3) Process Detection: The invisible window application appears as a standard user-level process in Task Manager (Windows) or Activity Monitor (macOS). Its process name, however, can be set to any value chosen by the attacker—

no distinguishing marker in standard process enumeration identifies a process as using display affinity flags.

Detecting the display affinity flag requires calling `GetWindowDisplayAffinity` on each top-level window handle—a capability that requires enumeration of the window hierarchy, which is available to any user-level process. However, current browser-based proctoring systems do not perform this check, as it requires native code execution outside the browser sandbox.

D. Summary of Findings

The Invisible Window attack achieves 100% evasion on all tested platforms (Windows 10/11, macOS 14–26) with zero detectable artefacts. The macOS finding is particularly notable: despite Apple's documented ScreenCaptureKit changes in macOS 15, `sharingType = .none` remains fully effective on macOS 26.3.1, contradicting prevailing community assumptions [18]. Pixel-level forensic analysis confirmed complete evasion on both platforms: on Windows, 80.27% of pixels in the 200,000-pixel capture region differed between the physical display and the excluded capture (Fig. 2), corresponding exactly to the invisible window's footprint; on macOS, 92% of pixels (1,086,110 of 1,170,560) were identical between captures with and without the invisible window (Fig. 3), with the remaining 8% attributable to concurrent background terminal output—confirmed by comparing captures taken with and without the terminal running, independently of the invisible window. Behavioural detection (gaze tracking, mouse dynamics) proved ineffective because the attack does not alter the test-taker's physical posture or screen-directed attention. Process-level detection is theoretically possible but not implemented by current browser-based proctoring systems.

E. Limitations

Several limitations bound the generalisability of these results. First, the behavioural evaluation used a small cohort ($n = 8$); all participants were members of the research team who provided informed consent. With $n = 8$ the gaze-tracking analysis ($p = 0.41$) has limited statistical power, and the absence of a significant result should not be interpreted as proof that behavioural detection is impossible in all settings. Second, all proctoring configurations were simulated in a controlled laboratory; we did not test against commercial proctoring products (e.g., ProctorU, Proctorio, Respondus) due to licensing and ethical constraints. Third, the macOS 26 evaluation used `CGWindowListCreateImage` and the `screencapture` utility rather than a browser-based `getDisplayMedia()` call, because no browser build at the time of testing exposed ScreenCaptureKit internals in a way that permitted controlled forensic comparison. Fourth, our evaluation covers Windows and macOS only; Linux display servers (X11, Wayland) were not tested. Finally, browser and OS vendors may patch the behaviour described here at any time; our results reflect the state of the tested software versions as of March 2026.

VI. Countermeasures

We propose and evaluate the following countermeasures against the Invisible Window attack, ranging from immediately deployable mitigations to longer-term architectural changes.

A. Display Affinity Flag Enumeration

Mechanism: A native proctoring agent enumerates all top-level windows using `EnumWindows` (Windows) or `CGWindowListCopyWindowInfo` (macOS) and calls `GetWindowDisplayAffinity` (Windows) or checks `sharingType` (macOS) on each window. Any window with a capture-exclusion flag set during an exam session is flagged as suspicious.

Feasibility: High on Windows—the API calls require no elevated privileges. On macOS, the enumeration additionally requires Screen Recording permission (a user-approved entitlement in System Preferences → Privacy & Security), which limits silent deployment but is feasible for authorised proctoring agents.

Limitations: (1) Requires a native agent, not just a browser extension. (2) The attacker can set the flag only momentarily or toggle it rapidly. (3) The attacker could name their process to mimic legitimate software (e.g., DRM-protected media players) that legitimately uses display affinity. (4) Our testing on macOS 26 shows `sharingType.none` remains effective for capture evasion, making this check necessary on all macOS versions.

Recommendation: Implement as a baseline check in native proctoring agents. This is the most immediately effective countermeasure on Windows.

B. Frame Comparison with Known State

Mechanism: The proctoring system compares the `getDisplayMedia()` output against the expected exam interface. Significant discrepancies (e.g., the captured frame shows only the exam page when the system expects other windows to be visible) could indicate capture evasion.

Feasibility: Low—the proctoring system does not have a ground truth for what the desktop should look like. The test-taker's desktop wallpaper, taskbar configuration, and notification state are all variable.

Limitations: Requires knowing what the "correct" screen should look like, which varies per user. High false positive rate.

Recommendation: Not viable as a primary countermeasure.

C. Application Focus Monitoring

Mechanism: The exam web application monitors JavaScript `blur` and `focus` events on the document and window objects. When the browser window loses focus (because the user clicked on the invisible window), the proctoring system logs the event and potentially flags it.

Feasibility: High—requires only JavaScript, no native code.

Limitations: (1) The invisible window can be designed as "click-through" using `WS_EX_TRANSPARENT` (Windows) or `ignoresMouseEvents` (macOS)—as implemented in

TABLE II
COUNTERMEASURE ASSESSMENT

Countermeasure	Effect.	Deployability	Evasion Diff.
Flag enumeration	High (Windows)	Medium (native agent)	Medium
Frame comparison	Low	High (JS only)	Low
Focus monitoring	Medium	High (JS only)	Medium
HW attestation	Very High	None (N/A)	Very High
OS capture integrity	High	Low (vendor dep.)	High

Variant D (Section IV-B)—preventing focus loss entirely. (2) Even non-click-through invisible windows may not trigger focus loss if the user only reads content without clicking. (3) Legitimate focus loss occurs frequently (system notifications, OS dialogs) and produces false positives.

Recommendation: Implement as a supplementary signal, not a primary detector. Pair with other telemetry to reduce false positives.

D. Hardware-Level Display Integrity Attestation

Mechanism: A trusted platform module (TPM) or secure enclave provides cryptographic attestation that the display output signal matches the capture output. This would detect any discrepancy between the physical display and the captured frame at the hardware level.

Feasibility: Very low—no current hardware platform provides this capability. It would require changes to GPU drivers, display controllers, and the TPM attestation chain.

Limitations: Requires hardware/firmware support that does not exist. Long development timeline. Privacy implications of hardware-attested screen content.

Recommendation: Long-term research direction, not a near-term solution.

E. OS-Level Capture Integrity Guarantees

Mechanism: The operating system provides a "capture integrity" mode that disables display affinity flags system-wide during a designated session, ensuring that all visible content appears in the capture output.

Feasibility: Medium—requires OS vendor cooperation (Microsoft, Apple). Apple's macOS 15 changes to ScreenCaptureKit were expected to address this, but remain incomplete (Section V).

Limitations: Requires OS vendor buy-in. May conflict with legitimate DRM use cases. Privacy concerns (users may not want all content capturable).

Recommendation: Advocate to OS vendors for a "proctoring mode" API that provides capture integrity guarantees within an explicitly authorised session.

F. Defence-in-Depth Assessment

No single countermeasure provides a complete defence. Table II summarises the countermeasure options.

The most practical near-term defence is a combination of (A) native agent-based flag enumeration with (C) JavaScript focus monitoring, supplemented by ongoing advocacy for (E) OS-level capture integrity APIs.

TABLE III
COORDINATED DISCLOSURE SUMMARY

Party	Notified	Response	Classification
Apple Product Security	Mar. 2026	Mar. 2026	Consistent with documented functionality; not a security issue
Microsoft MSRC	Feb. 2026	Apr. 2026	By-design behaviour; not a security vulnerability

VII. ETHICAL CONSIDERATIONS

A. Responsible Disclosure Framework

This research follows coordinated vulnerability disclosure principles [11]–[15] and the IEEE/ACM codes of ethics, which mandate prompt disclosure of factors endangering the public [12] and full disclosure of system limitations [11]. Our disclosure timeline:

1) **Discovery and verification** (January 2026): The display affinity bypass was identified and verified in a controlled laboratory environment on Windows 10 and macOS 14.
2) **macOS 26 verification** (February 2026): Extended testing to macOS 26.3.1, confirming that `sharingType = .none` remains effective despite Apple's documented ScreenCaptureKit changes.
3) **OS vendor communication** (February–March 2026): Microsoft and Apple were informed of the security implications of their display affinity APIs in the proctoring context via their respective security reporting channels (Microsoft MSRC, Apple Security Research portal).
4) **Public disclosure** (May 2026): This paper is published following OS vendor responses, with formal vendor classifications documented in Section VII-B.

The APIs exploited are publicly documented and already widely shared in online evasion communities [28]; withholding this research would not prevent exploitation but would delay informed countermeasures.

B. Vendor Responses and Security Boundary Classification

Following coordinated disclosure to OS vendors, both Apple Product Security and the Microsoft Security Response Center (MSRC) responded formally. Table III summarises the disclosure outcomes for both contacted OS vendors.

Apple Product Security responded that the behaviour is "consistent with Apple's documented functionality for `NSWindow.SharingType.none`," describing it as a legacy constant that "can cause content to not be available in certain sharing situations." Apple's own developer documentation explicitly advises: "Don't use this value to hide or omit content from being captured" [9]. This creates a security-relevant tension: Apple documents that `NSWindow.SharingType.none` should not be used to hide or omit content from capture, yet the behaviour remains available and effective on macOS 26.3.1 in downstream capture-dependent systems. Apple further observed that

"the `kCGWindowSharingState` property remains readable through the window list API, meaning proctoring or monitoring software can readily identify windows configured with this setting." On this basis, Apple classified the reported behaviour as not bypassing an Apple security boundary and therefore not a security issue [29].

Microsoft MSRC similarly classified the `SetWindowDisplayAffinity` behaviour as by-design rather than a security vulnerability, stating that the function "is not a security feature or an implementation of Digital Rights Management (DRM)" and offers "no guarantee" of strict content protection, citing the example of a photograph taken of the screen [8], [30].

We do not dispute either vendor's classification. *We therefore do not frame Invisible Window as an operating-system implementation bug.* Instead, we classify it as a security-relevant downstream design vulnerability in *capture-dependent systems*. The issue is a trust mismatch: browser-based proctoring systems assume that captured frames faithfully represent the physical display, while OS vendors document—and Apple explicitly warns against—behaviours that allow visible windows to be omitted from capture output. Apple's classification is internally consistent: the API behaviour is documented, and the security boundary Apple is concerned with (OS integrity) is unbroken. The security boundary that *is* broken is the one between the display compositor and the capture consumer—a boundary neither OS vendor designed or warranted.

Apple's counterpoint regarding `kCGWindowSharingState` detectability warrants a direct response. Apple is correct that native window enumeration can read `kCGWindowSharingState` to identify windows configured with `.none` sharing type. However, this detection requires a native process with access to the window list API. The browser sandbox has no access to `CGWindowListCopyWindowInfo`; only a native agent running outside the browser can perform this check. This is precisely the constraint that makes browser-only proctoring systems vulnerable: the detection mechanism that would close the gap is architecturally unavailable to them. This distinction reinforces the paper's central claim that the browser capture layer is not a display-integrity boundary, and that the countermeasures which are effective (Section VI-A) require the native privileges that browser-only systems lack.

This framing yields a clean thesis statement: *Invisible Window is not an Apple or Microsoft zero-day. It is a downstream display-fidelity failure caused when capture-dependent systems treat OS screen-capture output as equivalent to the physical display.*

C. Dual-Use Considerations

We acknowledge that this paper describes a technique that could be misused. Publication is justified because:

1) **The vulnerability is inherent, not introduced**: The APIs are publicly documented; we identified and formalised the risk, not created it.
2) **Community and commercial awareness already exists**: The technique is independently known and commercially exploited [28], [31]–[36], with 35% of candidates showing signs of AI-assisted cheating [37].
3) **Defenders need to know**: Proctoring vendors cannot develop countermeasures against a threat they do not understand.
4) **Alternative assessment matters**: Demonstrating structural capture-based proctoring limitations supports the argument [3], [38], [39] for alternative assessment designs.

D. Impact on Students

While proctoring serves academic integrity [1], [2], a substantial body of research documents its psychological costs: increased test anxiety, reduced performance, and perceived privacy violations among proctored students [40]–[42]. These costs are real regardless of whether proctoring is technically effective.

The Invisible Window attack raises a separate concern: the technical premise of screen-capture-based proctoring is structurally unsound. Institutions that accept the privacy and psychological costs of remote monitoring may not be receiving the security assurances they assume [3], [43].

E. Human Participants and Ethics Approval

The behavioural evaluation (Section V) involved eight members of the research team who served as participants for gaze tracking and mouse dynamics analysis. All participants provided informed consent. This study was reviewed by the Macquarie University Human Research Ethics Committee and determined to be exempt from full ethics review, as participants were adult members of the research team performing non-invasive screen-reading tasks with no deception, no collection of personal data, and no risk of harm beyond normal computer use.

F. Artifact Availability

Due to the dual-use nature of this work, proof-of-concept source code is not publicly released. The PoC implementations will be made available to verified security researchers and proctoring vendors upon request, subject to responsible use agreements. The forensic capture data and analysis scripts used to produce Figs. 2 and 3 are available at https://raoufabedini.dev/invisible-window.

VIII. Related Work

A. Proctoring System Security

Simko et al. [28] present the most directly related work, documenting community-developed evasion techniques ranging from non-technical methods to deeply technical approaches (custom virtual machines). Our work extends their findings by identifying an OS-level mechanism that is more reliable and less detectable than the techniques they catalogued. Balash et al. [4], [5] examine educator and student perspectives on

proctoring security, documenting known incidents and the adversarial dynamic between test-takers and proctoring systems. Luijben et al. [25] formalise five security requirements for proctoring systems, providing the basis for our analysis in Section II-D.

B. Proctoring Effectiveness and Criticism

A substantial body of work questions the effectiveness and desirability of online proctoring [1], [3], [6], [38], [39], [44], [45]. Studies demonstrate that proctored exams impose negative psychological effects on students [40], that digital proctoring may not be necessary at all [38], and that privacy violations in e-learning platforms are widespread [39]. The Invisible Window attack reinforces these critiques: the architecture of screen capture-based proctoring is structurally unsound.

C. Behavioural Detection Methods

Behavioural cheating detection research spans eye gaze tracking [20], [21], mouse dynamics [22]–[24], [26], [46], keystroke analysis [23], [24], and multimodal fusion [19], [27]. Our evaluation (Section V) demonstrates that these mechanisms are largely ineffective against the Invisible Window attack because the test-taker's physical behaviour is indistinguishable from legitimate exam behaviour.

D. Threat Modelling Across Platforms

Das Chowdhury et al. [47] demonstrate that threat models often fail to adapt when systems move across platforms, directly analogous to proctoring systems that assume a trustworthy OS display pipeline. Their STRIDE/LINDDUN framework for cross-platform threat model drift informs our trust boundary analysis.

E. Vulnerability Disclosure in Educational Technology

Security gaps in educational technology extend beyond proctoring to LMS platforms and anti-plagiarism systems [48]–[50]. Noordegraaf and Weulen Kranenbarg [2] find that "reporting vulnerabilities as a moral duty" is a primary driver for ethical hackers, a perspective that informs our disclosure approach alongside the operational framework of Reidsma et al. [51].

F. Commercial Exploitation and In-the-Wild Awareness

While no prior academic work has formally analysed the display affinity attack vector against proctoring systems, the underlying technique has been independently discovered and commercially exploited by several products targeting technical interviews and online assessments.

Commercial "invisible overlay" tools. Multiple commercial products now exploit OS-level display affinity APIs to create AI-powered overlays that are invisible to screen sharing. Interview Coder [31], launched in 2025 and updated to version 3.0 in March 2026, uses "special window flags" that mark its UI as excluded from screen captures—described as "similar to video-protected content" leveraging "standard OS APIs on Windows and macOS" [32]. The tool additionally employs click-through overlays, process name disguising, dock/taskbar hiding, and global hotkeys registered at the OS level rather than through the browser event system. Cluely [33], another commercially available tool, uses "low-level graphics hooks (DirectX on Windows, Metal framework on macOS)" to render its interface directly on the GPU's local display output, ensuring that "when a candidate shares their screen via Zoom or Teams, the video encoding pipeline captures only the desktop beneath the overlay" [37]. A 2026 industry survey reported that 35% of candidates showed signs of AI-assisted cheating in late 2025, with 59% of hiring managers suspecting candidates of using such tools [37].

Open-source proof-of-concepts. The open-source community has produced several demonstrations of this technique. Idanless [34] published an "Anti-Screen-Capture-window" proof-of-concept in April 2025—a Python/PyQt6 application embedding a ChatGPT browser within a `WDA_EXCLUDEFROMCAPTURE`-flagged window, explicitly referencing interview cheating as its use case. Khorev [35] documented the development of "Ezzi," an open-source invisible interview assistant built with Electron, describing the platform-specific challenges of achieving capture invisibility across Windows and macOS. Mayerr [36] published "openinterviewcoder," a cross-platform Electron application supporting Windows, macOS, and Linux.

Vendor awareness. The issue has been reported to Microsoft through their Q&A platform [52], where an educator documented students using `SetWindowDisplayAffinity(hWnd, WDA_EXCLUDEFROMCAPTURE)` to hide applications from classroom monitoring software. Community-suggested countermeasures included DLL injection to reset affinity flags and per-user services that periodically enumerate windows and terminate processes with capture-exclusion flags set. Proctoring vendors have begun responding: Proctorio [53] claims to block tools like Cluely by "preventing unauthorised apps from launching during exams," while Honorlock and Talview have implemented behavioural analysis targeting the timing patterns and gaze anomalies characteristic of AI-assisted responses.

Distinction from our work. The commercial tools and open-source PoCs described above demonstrate that the display affinity attack vector is independently known and actively exploited. However, to our knowledge no prior work has: (1) formally modelled the trust boundary violation between the browser capture API and the OS compositing pipeline; (2) provided a systematic cross-platform evaluation with pixel-level forensic verification; (3) analysed which behavioural detection mechanisms can and cannot detect the attack; (4) proposed and evaluated countermeasures; or (5) tested the attack on macOS 15+ where the conventional understanding was that Apple mitigated the vulnerability (Section V). This paper provides the first formal security analysis of a vulnerability class that is already being commercially exploited without academic scrutiny.

IX. AI Capability Uplift: A Case Study

The proof-of-concept implementations, literature synthesis, and forensic evaluation presented in this paper were developed using Claude Opus 4.6 (Anthropic, 1M context window) via Claude Code as the primary research instrument. We present this methodology as a standalone finding because it provides a concrete, empirically grounded case study directly relevant to AI safety evaluation.

A. Capability Uplift Measurement

Exploiting display affinity APIs for proctoring evasion requires familiarity with Win32 internals (`SetWindowDisplayAffinity` semantics, window handle management), macOS Objective-C/Swift window management (`NSWindow.sharingType`, ScreenCaptureKit filtering), screen capture pipeline architecture across two operating systems, and forensic verification methodology (pixel-level capture comparison, false-positive elimination). Using Claude as an orchestration layer, a single researcher with security fundamentals but no prior Win32 or ScreenCaptureKit experience produced working, empirically validated PoCs on both platforms within a single research session. The model identified the operationally critical distinction between `WDA_MONITOR` (black rectangle artefact) and `WDA_EXCLUDEFROMCAPTURE` (zero artefact) from Microsoft's API documentation, and independently flagged the discrepancy between Apple's documented macOS 15 ScreenCaptureKit changes and the actual runtime behaviour of legacy CoreGraphics APIs—the insight that motivated our macOS 26 evaluation and produced the paper's most unexpected finding (Section V).

B. Dual-Use Risk Assessment

These observations provide a concrete, empirically grounded case study for AI safety research on capability uplift and dual-use risk. The same LLM capabilities that accelerated legitimate security research—cross-platform API reasoning, forensically sound code generation, identification of undocumented behavioural discrepancies—are equally accessible to adversaries with no research intent. The effective skill barrier for this attack class dropped from multi-platform systems expertise to introductory security knowledge augmented by a single LLM session. This represents a measurable capability uplift: the attack surface is not only technically accessible (documented OS APIs) but *cognitively* accessible via LLM assistance, compounding the threat to proctoring systems.

C. Safety Guardrails

This research was conducted in accordance with Anthropic's usage policies. Prompts framed as direct facilitation of academic misconduct were declined by the model; all accepted prompts were contextualised within security research and responsible disclosure. This demonstrates that frontier AI safety measures provide meaningful but incomplete mitigation: the model correctly distinguished between research intent and misuse intent at the prompt level, but the resulting artefacts (working PoC code, forensic methodology) are transferable regardless of the original framing. This boundary between intent-level and artefact-level safety is itself a finding relevant to ongoing AI safety research.

X. Conclusion

This paper has demonstrated that browser-based proctoring systems that rely on the WebRTC `getDisplayMedia()` API for screen monitoring are vulnerable to a cross-platform evasion technique that exploits a structural gap in the OS display pipeline. The Invisible Window attack exploits documented OS-level display affinity APIs—Windows `SetWindowDisplayAffinity` with `WDA_EXCLUDEFROMCAPTURE` and macOS `NSWindow.SharingType.none`—to create application windows that are fully visible on the physical display but completely invisible to screen capture.

Our proof-of-concept implementations achieve 100% evasion on all tested platforms, including macOS 26 where the attack was previously assumed mitigated (Section V). The attack requires no elevated privileges and no software exploitation.

The implications extend beyond a single exploit. The Invisible Window attack reveals a structural weakness in the trust model underlying screen capture-based proctoring: the assumption that the OS compositing pipeline faithfully represents the physical display state. This assumption has never been explicitly validated and, as we show, does not hold.

We have proposed and evaluated countermeasures ranging from immediately deployable (display affinity flag enumeration) to long-term (hardware-level display integrity attestation). The most practical near-term defence combines native agent-based flag enumeration with JavaScript focus monitoring and advocacy for OS-level capture integrity APIs.

More broadly, this work reinforces the growing academic consensus [1], [3], [38], [39] that screen-based surveillance is not a sustainable foundation for academic integrity. As evasion techniques grow more sophisticated and less detectable, the arms race between proctoring systems and bypass methods favours the attacker. Institutions would be better served by investing in assessment designs that are inherently resistant to cheating—open-book examinations, authentic assessments, oral defences—rather than in ever-more-invasive monitoring of students' screens.

We have followed coordinated vulnerability disclosure principles throughout this research and have notified the relevant OS vendors. We hope this work contributes to the development of more robust, privacy-respecting, and sound approaches to maintaining academic integrity in online education.

Acknowledgment

This research was conducted using Claude Code powered by Anthropic's Claude Opus 4.6 large language model (1M context window). The author thanks Anthropic for providing the research tooling and notes that the dual-use findings

reported in Section IX—specifically the measured capability uplift from multi-platform systems expertise to introductory security knowledge, the model's independent identification of undocumented API behavioural discrepancies, and the observation of intent-level versus artefact-level safety boundaries—are offered as empirical contributions to Anthropic's ongoing AI safety evaluations. This paper serves as a real-world case study of frontier model dual-use in offensive security research conducted under responsible disclosure, and the author welcomes engagement from Anthropic's safety and policy teams on the implications documented herein.

References

[1] K. Lee and M. Fanguy, "Online exam proctoring technologies: Educational innovation or deterioration?" *British J. Educ. Technol.*, vol. 53, no. 3, pp. 475–490, 2022.

[2] J. E. Noordegraaf and M. Weulen Kranenbarg, "Why do young people start and continue with ethical hacking? A qualitative study on individual and social aspects in the lives of ethical hackers," *Criminology & Public Policy*, vol. 22, no. 4, pp. 803–824, 2023.

[3] M. Khalil, P. Prinsloo, and S. Slade, "In the nexus of integrity and surveillance: Proctoring (re)considered," *J. Computer Assisted Learning*, vol. 38, no. 6, pp. 1589–1602, 2022.

[4] D. G. Balash, E. Korkes, M. Grant, A. J. Aviv, R. A. Fainchtein, and M. Sherr, "Educators' perspectives of using (or not using) online exam proctoring," in *Proc. 32nd USENIX Security Symp.*, Anaheim, CA, 2023, pp. 5091–5108.

[5] D. G. Balash, D. Kim, D. Shaibekova, R. A. Fainchtein, M. Sherr, and A. J. Aviv, "Examining the examiners: Students' privacy and security perceptions of online proctoring services," in *Proc. 17th Symp. Usable Privacy and Security (SOUPS)*, 2021, pp. 633–652.

[6] A. Johri and A. Hingle, "Students' technological ambivalence toward online proctoring and the need for responsible use of educational technologies," *J. Eng. Educ.*, vol. 112, no. 1, pp. 221–242, 2023.

[7] J.-I. Bruaroey and E. Alon, "Screen Capture," https://www.w3.org/TR/screen-capture/, 2025, w3C Working Draft, World Wide Web Consortium, Jul. 2025.

[8] Microsoft, "SetWindowDisplayAffinity function (winuser.h)," https://learn.microsoft.com/en-us/windows/win32/api/winuser/nf-winuser-setwindowdisplayaffinity, 2025, microsoft Learn. Accessed: 2026.

[9] Apple Inc., "NSWindow.SharingType," https://developer.apple.com/documentation/appkit/nswindow/sharingtype, 2025, apple Developer Documentation. Accessed: 2026.

[10] D. Dharminder, "LWEDM: Learning with error based secure mobile digital rights management system," *Trans. Emerging Telecomm. Technol.*, vol. 32, no. 2, p. e4199, 2020.

[11] Association for Computing Machinery, "ACM Code of Ethics and Professional Conduct," https://www.acm.org/code-of-ethics, 2018, accessed: 2026.

[12] Institute of Electrical and Electronics Engineers, "IEEE Code of Ethics," https://www.ieee.org/about/corporate/governance/p7-8, 2024, accessed: 2026.

[13] OWASP Foundation, "Vulnerability Disclosure Cheat Sheet," https://cheatsheetseries.owasp.org/cheatsheets/Vulnerability_Disclosure_Cheat_Sheet.html, 2025, oWASP Cheat Sheet Series. Accessed: 2026.

[14] Forum of Incident Response and Security Teams (FIRST), "Guidelines and Practices for Multi-Party Vulnerability Coordination and Disclosure," FIRST, Guidelines, 2020, ver. 1.1. Available: https://www.first.org/global/sigs/vulnerability-coordination/multiparty/guidelines-v1-1.

[15] Cybersecurity and Infrastructure Security Agency, "Coordinated Vulnerability Disclosure Program," https://www.cisa.gov/resources-tools/programs/coordinated-vulnerability-disclosure-program, 2025, cISA. Accessed: 2026.

[16] Mozilla Developer Network, "Using the Screen Capture API," https://developer.mozilla.org/en-US/docs/Web/API/Screen_Capture_API/Using_Screen_Capture, 2025, mDN Web Docs. Accessed: 2026.

[17] Apple Inc., "ScreenCaptureKit," https://developer.apple.com/documentation/screencapturekit/, 2025, apple Developer Documentation. Accessed: 2026.

[18] columbusux, "macOS 15+: ScreenCaptureKit ignores setContentProtection / NSWindow.sharingType," https://github.com/tauri-apps/tauri/issues/14200, 2025, gitHub Issue #14200, tauri-apps/tauri. Accessed: 2026.

[19] Y. Atoum, L. Chen, A. X. Liu, S. D. H. Hsu, and X. Liu, "Automated online exam proctoring," *IEEE Trans. Multimedia*, vol. 19, no. 7, pp. 1609–1624, 2017.

[20] S. Kaddoura, S. Vincent, D. J. Hemanth, and I. Ashraf, "Computational intelligence and soft computing paradigm for cheating detection in online examinations," *Appl. Comput. Intell. Soft Comput.*, vol. 2023, p. 3739975, 2023.

[21] B. J. Ferdosi, M. Rahman, A. M. Sakib, T. Helaly, and P. Chakraborty, "Modeling and classification of the behavioral patterns of students participating in online examination," *Human Behavior and Emerging Technol.*, vol. 2023, p. 2613802, 2023.

[22] X. Wang, Q. Zheng, K. Zheng, T. Wu, and G. M. Perez, "User authentication method based on MKL for keystroke and mouse behavioral feature fusion," *Security and Communication Networks*, vol. 2020, p. 9282380, 2020.

[23] T. Lyu, L. Liu, F. Zhu, S. Hu, R. Ye, and J. Dalle, "BEFP: An extension recognition system based on behavioral and environmental fingerprinting," *Security and Communication Networks*, vol. 2022, p. 7896571, 2022.

[24] S. Yazici, H. Yildiz Durak, B. Aksu Dünya, and B. Şentürk, "Online versus face-to-face cheating: The prevalence of cheating behaviours during the pandemic compared to the pre-pandemic among Turkish university students," *J. Computer Assisted Learning*, vol. 39, no. 1, pp. 231–254, 2022.

[25] R. Luijben, F. van den Broek, and G. Alpár, "Security requirements for proctoring in higher education," in *2024 IEEE Global Eng. Educ. Conf. (EDUCON)*, Kos Island, Greece, 2024, pp. 1–5.

[26] N. Dilini, A. Senaratne, T. Yasarathna, N. Warnajith, and L. Seneviratne, "Cheating detection in browser-based online exams through eye gaze tracking," in *2021 6th Int. Conf. Information Technology Research (ICITR)*. IEEE, 2021, pp. 1–6.

[27] J. Guan, X. Li, Y. Zhang, and K. Andersson, "Design and implementation of continuous authentication mechanism based on multimodal fusion mechanism," *Security and Communication Networks*, vol. 2021, p. 6669429, 2021.

[28] L. Simko, A. Hutchinson, A. Isaac, E. Fries, M. Sherr, and A. J. Aviv, "'Modern problems require modern solutions': Community-developed techniques for online exam proctoring evasion," in *Proc. 2024 ACM SIGSAC Conf. Computer and Communications Security (CCS '24)*, Salt Lake City, UT, 2024.

[29] Apple Product Security, "Response to Report OE11053003674113: NSWindow.sharingType = .none hides visible windows from ScreenCaptureKit/WebRTC capture," Apple Security Research Portal, private communication, Mar. 2026, formal vendor response to coordinated disclosure. On file with the author.

[30] Microsoft Security Response Center, "RE: 111448 CRM:0034000320 — Response to SetWindowDisplayAffinity proctoring bypass report," Microsoft Security Response Center, private communication, Apr. 2026, formal vendor response to coordinated disclosure. On file with the author.

[31] Interview Coder, "Interview Coder — AI Interview Assistant for Technical Interviews," https://www.interviewcoder.co/, 2026, accessed: 2026.

[32] OpenPR, "Interview Coder 3.0: Promises Complete Invisibility During Technical Interviews," https://www.openpr.com/news/4427427/interview-coder-3-0-promises-complete-invisibility-during, 2026, openPR, Mar. 2026. Accessed: 2026.

[33] Cluely, "Cluely — AI-Powered Real-Time Interview Assistant," https://cluely.com/, 2025, accessed: 2026.

[34] idanless, "Anti-Screen-Capture-window: Hidden ChatGPT Browser (Anti-Screen Capture PoC)," https://github.com/idanless/Anti-Screen-Capture-window, 2025, gitHub. Accessed: 2026.

[35] D. Khorev, "Building Ezzi: My Journey Creating an Invisible Tech Interview Assistant (Now Open Source)," https://levelup.gitconnected.com/building-ezzi-my-journey-creating-an-invisible-tech-interview-assistant-now-open-source-a1963a8fe0f3, 2026, level Up Coding. Accessed: 2026.

[36] J. Mayerr, "openinterviewcoder: An undetectable AI assistant for coding interviews," https://github.com/JoshMayerr/openinterviewcoder, 2025, gitHub. Accessed: 2026.

[37] FabricHQ, "Interview Cheating in 2026: The Rise of AI Tools Like Cluely and Interview Coder," https://www.fabrichq.ai/blogs/interview-cheating-in-2026-the-rise-of-ai-tools-like-cluely-and-interview-coder, 2026, accessed: 2026.

[38] A. Duncan and D. Joyner, "On the necessity (or lack thereof) of digital proctoring: Drawbacks, perceptions, and alternatives," *J. Computer Assisted Learning*, vol. 38, no. 5, pp. 1482–1496, 2022.

[39] B. Paris, R. Reynolds, and C. McGowan, "Sins of omission: Critical informatics perspectives on privacy in e-learning systems in higher education," *J. Assoc. Inf. Sci. Technol.*, vol. 73, no. 5, pp. 708–725, 2021.

[40] R. Conijn, A. Kleingeld, U. Matzat, and C. Snijders, "The fear of big brother: The potential negative side-effects of proctored exams," *J. Computer Assisted Learning*, vol. 38, no. 6, pp. 1521–1534, 2022.

[41] M. Rüth, M. Jansen, and K. Kaspar, "Cheating behaviour in online exams: On the role of needs, conceptions and reasons of university students," *J. Computer Assisted Learning*, vol. 40, no. 5, pp. 1987–2008, 2024.

[42] M. Gribbins and C. J. Bonk, "An exploration of instructors' perceptions about online proctoring and its value in ensuring academic integrity," *British J. Educ. Technol.*, vol. 54, no. 6, pp. 1693–1714, 2023.

[43] P. Prinsloo, M. Khalil, and S. Slade, "Vulnerable student digital well-being in AI-powered educational decision support systems (AI-EDSS) in higher education," *British J. Educ. Technol.*, vol. 55, no. 5, pp. 2075–2092, 2024.

[44] E. Marano, P. M. Newton, Z. Birch, M. Croombs, C. Gilbert, and M. J. Draper, "What is the student experience of remote proctoring? A pragmatic scoping review," *Higher Educ. Quarterly*, vol. 78, no. 3, pp. 1031–1047, 2024.

[45] S. Mukherjee, V. Distler, G. Lenzini, and P. Cardoso-Leite, "Balancing the perception of cheating detection, privacy and fairness: A mixed-methods study of visual data obfuscation in remote proctoring," in *Proc. 2024 European Symp. Usable Security (EuroUSEC '24).* Karlstad, Sweden: ACM, 2024.

[46] T. Hu, W. Niu, X. Zhang, X. Liu, J. Lu, Y. Liu, and J. Chen, "An insider threat detection approach based on mouse dynamics and deep learning," *Security and Communication Networks*, vol. 2019, p. 3898951, 2019.

[47] P. Das Chowdhury, M. Sameen, J. Blessing, N. Boucher, J. Gardiner, T. Burrows, R. Anderson, and A. Rashid, "Threat models over space and time: A case study of end-to-end-encrypted messaging applications," *Software: Practice and Experience*, vol. 54, no. 12, pp. 2316–2335, 2024.

[48] A. A. Mehrishi, D. K. Sarmah, and M. Daneva, "How can cryptography secure online assessments against academic dishonesty?" *Security and Privacy*, vol. 8, no. 4, p. e70065, 2025.

[49] A. Lachheb, V. Abramenka-Lachheb, S. Moore, and C. Gray, "The role of design ethics in maintaining students' privacy: A call to action to learning designers in higher education," *British J. Educ. Technol.*, vol. 54, no. 6, pp. 1653–1670, 2023.

[50] K. L. Adkins and D. A. Joyner, "Scaling anti-plagiarism efforts to meet the needs of large online computer science classes: Challenges, solutions, and recommendations," *J. Computer Assisted Learning*, vol. 38, no. 6, pp. 1603–1619, 2022.

[51] D. Reidsma, J. van der Ham, and A. Continella, "Operationalizing cybersecurity research ethics review: From principles and guidelines to practice," in *Proc. 2nd Int. Workshop on Ethics in Computer Security (EthiCS 2023)*, San Diego, CA, Feb. 2023, co-located with NDSS Symp.

[52] M. Ukt, "SetWindowDisplayAffinity bad usecase," https://learn.microsoft.com/en-us/answers/questions/1653885/setwindowdisplayaffinity-bad-usecase, 2024, microsoft Q&A. Accessed: 2026.

[53] Proctorio, "How Proctorio Blocks Cluely: Stopping AI Cheating Tools Like Cluely in Online Exams," https://proctorio.com/about/blog/how-proctorio-blocks-cluely, 2026, proctorio Blog. Accessed: 2026.