Hunting Insecure UI Properties in Extended Reality

Bertram Liu
George Mason University
Fairfax, Virginia, USA
bliu22@gmu.edu

Vamsi Shankar Simhadri
George Mason University
Fairfax, Virginia, USA
vsimhadr@gmu.edu

Xiaokuan Zhang
George Mason University
Fairfax, Virginia, USA
xiaokuan@gmu.edu

Abstract

This paper addresses critical security vulnerabilities in Extended Reality (XR) user interfaces (UIs) by developing systematic detection mechanisms. Recent research has identified that XR applications are susceptible to UI-based attacks due to insecure properties like spatial overlap and invisible boundaries. These vulnerabilities arise from XR's unique characteristic of integrating digital content into the physical world, where multiple virtual elements from different sources must coexist in the same perceptual space. We present a detection framework that helps developers identify potentially malicious UI elements through continuous runtime analysis. Our framework implements two key detection mechanisms: a same-space detector that identifies overlapping UI elements that could enable clickjacking attacks, and an invisibility detector that discovers hidden boundary objects that could block legitimate interactions. We implement this framework as a Unity plugin and demonstrate its effectiveness through comprehensive evaluation across multiple attack scenarios. Our results show that the framework successfully detects both same-space and invisibility attacks while maintaining acceptable performance overhead. This work represents a significant step toward securing XR applications against UI-based attacks and provides developers with practical tools to identify and mitigate these vulnerabilities during development and runtime.

CCS Concepts

• **Security and privacy** → **Mobile platform security**; **Software security engineering**.

Keywords

Extended Reality, UI Security, Runtime Detection

ACM Reference Format:
Bertram Liu, Vamsi Shankar Simhadri, and Xiaokuan Zhang. 2025. Hunting Insecure UI Properties in Extended Reality. In *The Twenty-sixth International Symposium on Theory, Algorithmic Foundations, and Protocol Design for Mobile Networks and Mobile Computing (MobiHoc '25), October 27–30, 2025, Houston, TX, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3704413.3765301

1 Introduction

Augmented Reality (AR) and Virtual Reality (VR) are transforming how humans interact with digital information: AR overlays digital content onto the physical world, while VR creates fully immersive digital environments. AR and VR are collectively referred to as Extended Reality (XR). The global XR market is expected to reach a market worth of $472.39 billion by 2029 [1]. This significant market expansion demonstrates the widespread integration of XR technologies across diverse domains. These technologies have found critical applications in education for interactive learning environments [2], healthcare for surgical training and patient care [3], manufacturing for process optimization [4], retail for enhanced shopping experiences [5], real estate for virtual property tours [5], remote work for team collaboration [6], and industrial design for prototyping [4]. This broad adoption indicates a clear transition of XR from niche applications to essential tools in modern computing environments.

XR systems fundamentally alter user interaction with digital content through immersive environments. This immersive nature introduces unique security challenges that differ from traditional computing interfaces. While conventional displays limit digital content to a screen, XR user interfaces (UIs) integrate digital elements throughout the user's field of view. This integration creates novel attack vectors that exploit human perception and spatial awareness. Attackers can manipulate virtual objects in three-dimensional space to deceive users, potentially triggering unintended actions or exposing sensitive information. The security challenges are amplified by modern XR applications' modular architecture, where content from multiple sources, including third-party components, must coexist within the same perceptual space. This multi-source integration complicates UI integrity maintenance and trust boundary establishment.

Recent research by Cheng *et al.* [7] has systematically investigated security vulnerabilities specific to AR user interfaces. Their work examines a threat model involving interactions between multiple entities within AR UIs, particularly focusing on scenarios where third-party components (such as external libraries) embedded within AR applications may attempt to compromise application security, or vice versa. Through their analysis, they identified several critical UI vulnerabilities. Two notable examples demonstrate the severity of these threats: i) The **same-space attack** involves attackers taking advantage of spatial ambiguity by placing several virtual objects at the same 3D location, facilitating clickjacking attacks that mislead users into unintended actions. ii) The **invisible attack** entails attackers encapsulate AR objects within invisible 3D boundaries to perform denial-of-service attacks, thereby obstructing user access to legitimate interface components. The researchers validated these attack vectors across major AR platforms and devices, indicating the widespread susceptibility of these vulnerabilities.

In this paper, we found that the attacks are also applicable to VR scenarios. To address the lack of systematic defense mechanisms against these UI-based security threats, we propose an insecure UI scanner that helps developers identify potentially malicious UI elements. Our approach performs continuous runtime analysis of XR scenes through periodic scanning to detect suspicious UI properties.

The scanner implements automated detection mechanisms that examine spatial relationships and visibility properties of XR objects, alerting users to potential security issues. We implement this scanner as a Unity plugin with two main detection goals: *same-space object detection* for identifying overlapping UI elements that could enable clickjacking attacks, and *invisible object detection* for discovering hidden boundary objects that could block legitimate interactions. While our implementation targets Unity due to its widespread use in XR development, our approach is adaptable to other XR frameworks.

We systematically evaluated our scanner's performance using a Meta Quest 2 headset connected to a Dell G15 desktop computer. Our evaluation methodology involved running the scanner across VR scenes while varying two key parameters: scanning frequency and scene complexity (number of objects). Results demonstrate that CPU utilization, GPU utilization, and memory consumption remain stable across test conditions. The primary performance impact manifests in frame rate reduction, which correlates with increased scanning frequency and scene complexity.

Contributions. This paper makes the following key contributions:

- We design a scanner that detects UI-based vulnerabilities in XR applications by identifying same-space and invisible attacks.
- We create a Unity plugin that implements our detection mechanisms and evaluate its effectiveness through controlled experiments in VR environments.
- We identify key research directions including performance optimization, broader application testing, and empirical user studies to advance XR security.

2 Background and Related Work

2.1 UI Security in 2D Displays

Traditional devices such as desktops or smartphones have a 2D flat surfaces for display, which suffer from clickjacking and UI redressing attacks that can manipulate users into triggering unintended actions through deceptive layering or hidden elements [8, 9]. Mobile platforms face additional risks such as tapjacking [10] and overlay attacks [11], where the limited size of touchscreens allows malicious components to intercept input or disguise interface elements [12, 13]. Defensive methods in 2D environments include framebusting [14] techniques designed to stop unauthorized framing of web content. Security tools have been created to address mobile-specific risks and reduce exposure to overlay-based manipulation [15, 16].

2.2 Security Issues in XR

There is an emerging awareness of security and privacy concerns within XR systems. Specifically, recent work has highlighted side-channel vulnerabilities in XR devices, demonstrating that motion sensors in head-mounted displays and controllers can inadvertently leak sensitive information. Such leakage enables various attacks like key stroke inference using motion data [17–20], acoustic signals [21], network traffic [22], Wi-Fi signals[23] and user movements [24]. Collectively, these works indicate that beyond improving functionality and user experience, XR platforms introduce new and often overlooked vectors for information leakage. To mitigate such threats, researchers have mainly adopted differential privacy noise injection for motion data [25–28] to reduce the attack accuracy.

UI Security in XR. Unlike traditional 2D interfaces, where users interact through external screens, XR immerses users inside interactive three-dimensional environments. Interaction occurs through natural inputs such as gaze, gesture, and voice rather than clicks or taps, introducing new security and privacy challenges. Virtual elements can blend seamlessly with the real or virtual environment, making it difficult to distinguish safe content from malicious interference. Attackers might inject fake objects, alter shared cues, or spoof entire scenes to mislead participants [29]. For example, a fake warning in AR might resemble a system alert floating in your space, or in VR, a pop-up could mimic a trusted app. Malicious buttons or menus might appear as legitimate parts of the system but perform harmful actions. Research works [7, 30] have focused on the security of the user interface in XR and how an attacker can exploit the security properties of the user interface. AdCube [31] was introduced to address ad frauds within WebXR environments and serves as a defense mechanism against malicious advertising activities. Cheng *et al.* [7] have identified specific UI properties that introduce vulnerabilities on modern AR platforms. A concurrent work by Xiu *et al.* [32] proposed a way for detecting obstruction attacks for augmented reality using vision language models. Their focus is to detect overlaps between virtual and physical objects; however, we focus on detecting same-space and invisible virtual objects to address the attacks proposed in [7].

3 Motivation

Recent research by Cheng *et al.* [7] has identified critical security vulnerabilities in AR user interfaces that stem from specific UI properties. Through systematic analysis across major AR platforms, they characterized three key security properties that can lead to user misperception and potential attacks: 1) Same-Space Scenario: A condition where multiple virtual elements share identical spatial coordinates in the AR environment; 2) Invisibility Scenario: A situation where AR objects are rendered with full or partial transparency while maintaining interaction capabilities; and 3) Synthetic Input Scenario: The ability for programmatic systems to generate artificial inputs that mimic genuine user interactions. Their research demonstrated that malicious actors can exploit these properties to execute various attacks including clickjacking, denial-of-service, and input forgery. Through proof-of-concept implementations on widely-used AR platforms such as ARKit, ARCore, HoloLens, Oculus, and WebXR, they established that these vulnerabilities represent concrete security risks rather than theoretical concerns.

3.1 Scope

Building on their findings, our work focuses on developing detection mechanisms against the Same-Space and Invisibility scenarios (explained below), and we leave the investigation of Synthetic Input scenarios for future work. While the presence of same-space or invisibility properties alone does not conclusively indicate malicious intent, these conditions create opportunities for attackers to exploit user perception and interaction patterns.

- **Same-Space Scenario**: This vulnerability occurs when multiple virtual objects occupy identical spatial coordinates in an XR environment. The objects may share identical geometric properties, resulting in complete visual overlap. An attacker can exploit this spatial ambiguity by positioning a malicious interactive object at

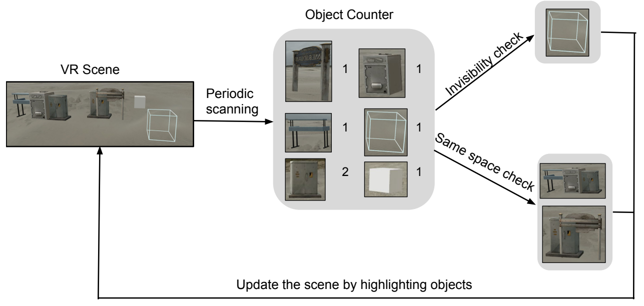


Figure 1: Workflow

the exact coordinates of a legitimate UI element, causing users to unknowingly interact with the malicious object rather than their intended target.

- **Invisibility Scenario**: This vulnerability stems from XR system's handling of transparent virtual objects. While these objects may be partially or fully invisible to users, XR platforms typically maintain their full interactive capabilities, including input event handling and collision detection. This creates a security risk where attackers can deploy transparent interactive layers over legitimate UI elements. These invisible layers can intercept user interactions intended for underlying objects, effectively hijacking the input without any visual indication to the user.

3.2 Threat Model

Our threat model focuses on XR applications that incorporate third-party software components, such as external libraries and plugins, which share the same UI space as the main application. This model aligns with prior work by Cheng *et al.* [7], where third-party components embedded within a legitimate XR application may exhibit malicious behavior. We aim to build a defense mechanism for the developers to use, so that such insecure UI properties can be captured and mitigated at runtime.

4 Insecure UI Property Scanner

To address the two insecure UI scenarios, we develop runtime detection mechanisms that enable developers to identify potentially dangerous UI properties during application execution. We chose runtime dynamic detection instead of static analysis, as many UI properties depend on runtime context, such as dynamic layouts, user inputs, or framework-driven rendering, which cannot be fully inferred from code alone through static analysis. Our insecure UI property scanner continuously monitors the application's UI state and, when suspicious patterns are detected, provides immediate alerts and mitigation options to protect users from potential exploitation.

The high-level idea of our scanner is to periodically checking the objects rendered in the scene, and alert the user when insecure UI properties related to the two scenarios (§ 3.1) are detected. The design of our scanner is shown in Figure 1. When the VR application is running, our scanner will first identify all the object types and their properties, such as coordinates and visibility attributes (❶). Then, it will run the two detectors designed for detecting the two scenarios (❷). When issues are detected, the objects that contain insecure UI properties will be highlighted, and information will be sent to the user (❸).

The scanner will be run at a fixed interval determined by the application developer. The scanner can be embedded in any XR applications to perform the detection. Our scanner was implemented within the Unity engine using C# scripts compatible with the Universal Render Pipeline. Scene scanning is performed at regular intervals, during which all `Objects` including those marked as inactive are evaluated against the invisibility and overlap criteria. Note that while our implementation is based on Unity, our design can be easily extended to other XR platforms. In the following, we explain how we implemented our scanner.

4.1 Object Identification

The first step of our scanner is to identify all objects. To do so, The scanner gathers every Game Object in the scene, both active and inactive. For each object, it accesses the GameObject reference, its spatial information, its Renderer and materials, its MeshFilter, and its Collider. The spatial information and Collider data are used to determine the object's coordinates and whether multiple objects occupy the same space (for the same-space scenario). The Renderer, MeshFilter, and material properties are used to determine whether an object is invisible (for the invisibility scenario).

4.2 Object Detection

We detect two types of objects: objects occupying the same space (*i.e.*, overlapping), and objects that are invisible.

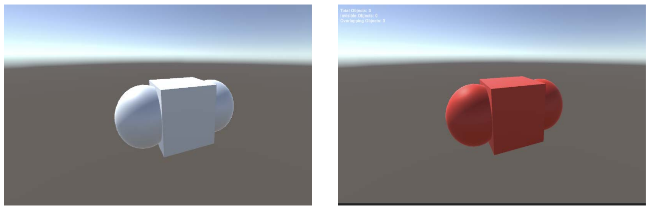
Same-space object detection. After we have identified all objects and their properties, we detect same-space objects based on pairwise comparisons of identified objects in the scene. The steps of this procedure are detailed in algorithm 1. If both objects contain valid Collider components, we use Unity's `Bounds.Intersects` function to check for spatial intersection between their bounding boxes. In the absence of colliders, we implemented a fallback mechanism that checks whether the objects' positions lie within a small threshold distance (e.g., 0.05 units), which indicates a likely overlap. This method ensures coverage across objects with and without colliders, increasing the robustness of detection in varied scene configurations.

Invisible object detection. Invisibility detection is implemented using a rule-based approach that evaluates various object rendering properties to identify elements that are fully or partially hidden from the user's view. The steps used for Invisibility detection is detailed in algorithm 2. We developed these rules by consulting Unity documentation [33] and performing extensive empirical testing of parameters, such as material alpha values, shader settings, and renderer states. These rules are designed to cover a wide range of techniques, both intentional and unintentional, used to hide objects in XR environments. Based on these rules, an object is classified as invisible if it meets at least one of the following conditions:

- a disabled `Renderer` component;
- alpha values below a visibility threshold of 0.05 in properties such as `_Color.a`, `_Alpha`, or `_Opacity`;
- a material explicitly listed by the developer as invisible;
- shader names containing keywords such as `transparent`, `fade`, or `invis`;

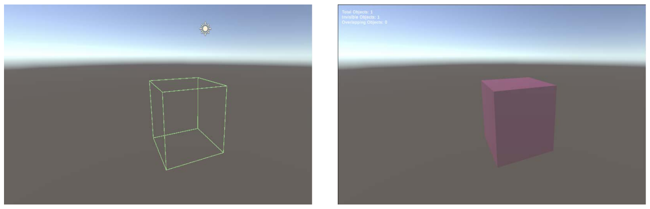
Algorithm 1: Same-space Object Detection

```
Input: GameObjects $a$, $b$
Output: true if $a$ and $b$ overlap, false otherwise
$aCol \leftarrow a$.GetComponentCollider;
$bCol \leftarrow b$.GetComponentCollider;
if $aCol \neq null$ and $bCol \neq null$ then
    if $aCol.bounds.Intersects(bCol.bounds)$ then
        return true
else
    $d \leftarrow$ Vector3.Distance(a.position, b.position);
    if $d < 0.05$ then
        return true
return false
```



(a) Before Detection (b) After Detection

Figure 2: An example of same-space object detection.



(a) Before Detection (b) After Detection

Figure 3: An example of invisible object detection. In (a), the object outline was plotted in green for demonstration purposes.

- a render queue value above a certain threshold[1];
- use of the Unity Standard Shader in transparent mode;
- a `MeshFilter` component without an assigned mesh.

4.3 Object highlighting and Information Sharing

After the detection script runs, we highlight the objects which are invisible and overlapping so that the user can distinguish and be aware. Figure 2 and Figure 3 visually illustrate scenes before and after applying detection. We have also developed a floating UI that is rendered within the XR scene and it updates periodically to reflect current detection metrics. It shows the total number of scene objects, the count of invisible objects, and the number of objects that occupy the same space (top left corner in Figure 2b and Figure 3b). The interface is positioned to follow the user's head movement while remaining in a fixed location within the field of view.

[1]Based on our empirical analysis, a value >=`2500` will make an object invisible.

Algorithm 2: Invisibility Detection

```
Input: A scene object $o$ with components: Renderer,
       Materials $M = \{m_1, m_2, \ldots, m_n\}$, Shader, MeshFilter
Output: Boolean value indicating whether $o$ is invisible
if Renderer component of $o$ is disabled then
    return true
foreach material $m \in M$ do
    if $Alpha(m) < 0.05$ and $RenderQueue(m) > 2500$ then
        return true
    if $m \in InvisibleMaterialList$ then
        return true
    if ShaderName(m) contains "transparent", "fade", or
     "invis" (case-insensitive) then
        return true
    if ShaderName(m) = "Standard" and
     RenderingMode(m) = "Transparent" then
        return true
if $o$ has a MeshFilter component and Mesh(o) is null then
    return true
return false
```

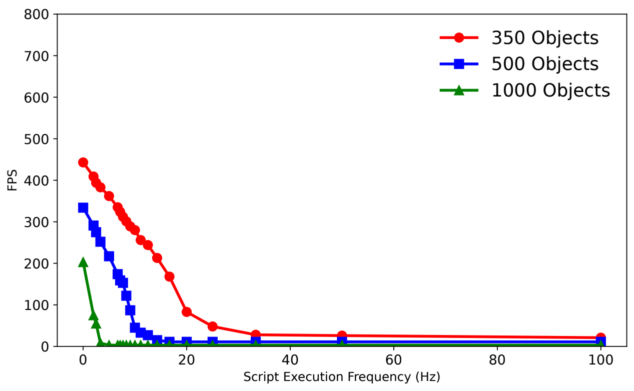
5 Evaluation

We evaluated our scanner (C# script) on a set of VR scenes designed to simulate typical VR environments. We use an Meta Quest 2 device connected to a Dell G15 desktop equipped with an AMD Ryzen 5 5600H processor with Radeon Graphics (6 cores), 16 GB of RAM, and an NVIDIA GeForce RTX 3050 GPU. For evaluating our scanner, we ran our script on VR scenes while varying both the execution frequency and the number of objects in the scene. The script scanned the entire scene at regular intervals to detect invisible and overlapping objects. We monitored CPU usage, GPU usage, memory consumption, and frame rate throughout the experiments using Unity's built-in profiling tools.

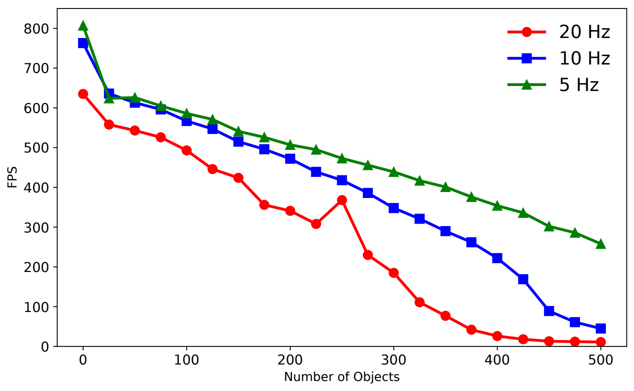
5.1 Same-space Object Detection

Execution frequency. We measured frame rates with object counts of 350, 500, and 1000 while running the scanner with frequency between 0 to 100 Hz. When the scanner was first enabled at 0.01s, FPS dropped sharply for all object counts, as illustrated in Figure 4a. Specifically, with 350 objects, FPS decreased from about 450 FPS to around 20 FPS; with 500 objects, it fell from approximately 330 FPS to around 10 FPS; and with 1000 objects, FPS plummeted from about 200 FPS to near 5 FPS. As the interval between executions increased, the FPS gradually recovered. At 2Hz frequency, FPS increased to about 410 FPS with 350 objects, about 290 FPS with 500 objects, and about 75 FPS with 1000 objects. Across all frequencies, larger object counts consistently produced lower FPS. GPU, CPU, and RAM did not show significant changes during these tests.

Objects count. Figure 4b shows the FPS trends across increasing object counts with updated data. At 20 Hz frequency (0.05s interval), FPS begins at 684 and remains consistently high across all object counts. Even at 500 objects, FPS stays at 461, with only small fluctuations throughout. This indicates stable performance and efficient handling of frequent updates. At 10 Hz frequency (0.10s interval), FPS starts higher at 739, with moderate variation across the object range. It dips slightly in some places but remains above 487 FPS at



(a) Execution Frequency



(b) Object Count

Figure 4: Analysis of same-space object detection.

500 objects, showing reliable performance under medium-frequency updates. The 5Hz frequency (0.20s interval) also maintains strong performance, starting at 762 and ending at 499 FPS at 500 objects. FPS remains steady across all counts, with minor dips and recoveries. Overall, all three intervals maintain high and stable frame rates, with minimal degradation as object count increases, indicating that the system handles invisibility detection efficiently even under load.

5.2 Invisible Object Detection

Execution frequency. As illustrated in Figure 5a, when the scanner was disabled, all configurations consistently achieved approximately 625 FPS. However, enabling the scanner caused a drop in FPS, which then remained stable across all tested frequencies from 0 to 100Hz. The impact of the scanner on FPS was directly related to the number of objects in the scene: with 350 objects, the FPS dropped to roughly 530–540; with 500 objects, the FPS stabilized around 455–465; and with 1000 objects, the FPS remained at approximately 265–275. These results indicate that, while the frequency itself did not cause further FPS variation, the scanner's performance degrades considerably in more complex scenes. GPU, CPU, and RAM usage remained roughly the same in all cases.

Objects count. Figure 5b illustrates how FPS changes with increasing object counts across different frequency. At 20 Hz frequency (0.05s interval), FPS starts high at 677 but steadily decreases, dropping significantly from around 540 FPS at 75 objects to just 11 FPS at 500 objects, indicating a severe performance drop. At a 10 Hz frequency (0.10s interval), FPS declines more gradually at first, maintaining above 350 FPS up to 325 objects, before sharply falling to 42 FPS at 500 objects. The 5 Hz (0.20s interval) frequency interval shows the most stable performance, with a gentle FPS decline from 677 down to 243 FPS at 500 objects, suggesting that less frequent scanner execution effectively maintains higher FPS under heavy object loads.

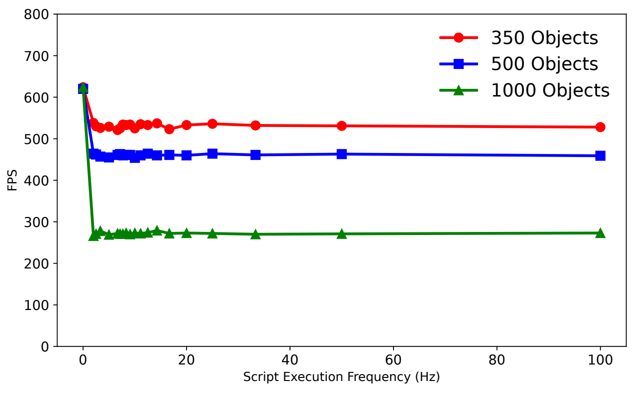
6 Conclusion and Future work

In this work, we presented the first systematic approach for detecting insecure UI properties in XR platforms. Our key contribution is an insecure UI property scanner implemented as a Unity plugin that continuously monitors XR applications at runtime. The scanner focuses on detecting two insecure UI properties: spatial overlap between objects that could enable clickjacking attacks, and invisible boundary objects that could block legitimate user interactions. Through systematic evaluation varying both detection frequency and scene complexity, we demonstrated that our scanner can effectively identify these vulnerabilities while quantifying its performance overhead.

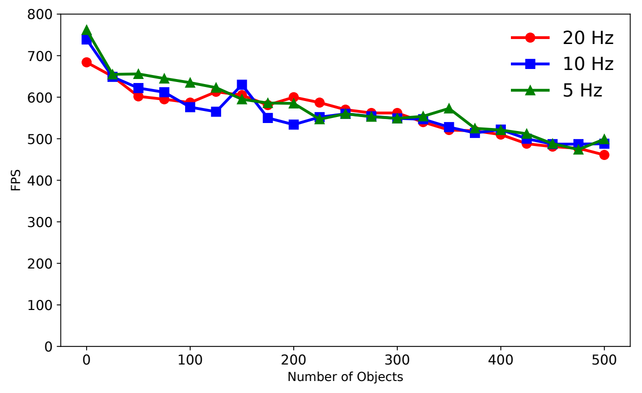
Our research opens several promising directions for future investigation. First, the current scanning approach of periodically checking the entire scene at fixed intervals introduces unnecessary computational overhead. A more efficient solution would be to implement incremental scanning that only processes objects that have changed since the last scan. Second, our evaluation was limited to a single test scene, which may not capture the full range of real-world XR applications. Future work should evaluate the scanner across diverse XR applications with varying levels of scene complexity and interaction patterns. Third, a formal user study would provide valuable insights into both the practical utility of our detection system and help determine optimal scanner configurations that balance detection effectiveness with user experience. Such a study could inform guidelines for detection frequency and visualization approaches that minimize disruption while maintaining security.

Acknowledgement

Bertram Liu was supported in part by 1) an undergrad research award from the Undergraduate Research Scholars Program (URSP) at Office of Student Creative Activities and Research (OSCAR), George Mason University, and 2) a CCI NoVa Node Cybersecurity Undergraduate Research Assistantship. This research was funded in part by 4-VA, a collaborative partnership for advancing the Commonwealth of Virginia.



(a) Execution Frequency



(b) Object Count

Figure 5: Analysis of invisible object detection.

References

[1] Mordor Intelligence, "Extended Reality Market Size & Share Analysis - Growth Trends & Forecasts (2024 - 2029)." https://www.mordorintelligence.com/industry-reports/extended-reality-xr-market, 2024.

[2] J. C.-Y. Sun, S.-L. Ye, S.-J. Yu, and T. K. Chiu, "Effects of wearable hybrid ar/vr learning material on high school students' situational interest, engagement, and learning performance: The case of a physics laboratory learning environment," *Journal of Science Education and Technology*, vol. 32, no. 1, pp. 1–12, 2023.

[3] D. Jones, S. Fealy, D. Evans, and R. Galvez, "The use of extended realities providing better patient outcomes in healthcare," *Frontiers in medicine*, vol. 11, p. 1380046, 2024.

[4] S. Doolani, C. Wessels, V. Kanal, C. Sevastopoulos, A. Jaiswal, H. Nambiappan, and F. Makedon, "A review of extended reality (xr) technologies for manufacturing training," *Technologies*, vol. 8, no. 4, p. 77, 2020.

[5] S. Zhang and W. Li, "Applying extended reality (xr) technology in commerce, management, and business applications: A survey," in *2024 4th International Conference on Computer, Control and Robotics (ICCCR)*, pp. 108–113, IEEE, 2024.

[6] Y. Lee and B. Yoo, "Xr collaboration beyond virtual reality: work in the real world," *Journal of Computational Design and Engineering*, vol. 8, no. 2, pp. 756–772, 2021.

[7] K. Cheng, A. Bhattacharya, M. Lin, J. Lee, A. Kumar, J. F. Tian, T. Kohno, and F. Roesner, "When the user is inside the user interface: An empirical study of {UI} security properties in augmented reality," in *33rd USENIX Security Symposium (USENIX Security 24)*, pp. 2707–2723, 2024.

[8] L.-S. Huang, A. Moshchuk, H. J. Wang, S. Schecter, and C. Jackson, "Clickjacking: Attacks and defenses," in *21st USENIX Security Symposium (USENIX Security 12)*, pp. 413–428, 2012.

[9] C.-A. Staicu and M. Pradel, "Leaky images: Targeted privacy attacks in the web," in *28th USENIX Security Symposium (USENIX Security 19)*, pp. 923–939, 2019.

[10] S. Aonzo, A. Merlo, G. Tavella, and Y. Fratantonio, "Phishing attacks on modern android," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pp. 1788–1801, 2018.

[11] A. Bianchi, J. Corbetta, L. Invernizzi, Y. Fratantonio, C. Kruegel, and G. Vigna, "What the app is that? deception and countermeasures in the android user interface," in *2015 IEEE Symposium on Security and Privacy*, pp. 931–948, IEEE, 2015.

[12] Y. Lee, X. Wang, K. Lee, X. Liao, X. Wang, T. Li, and X. Mi, "Understanding {iOS-based} crowdturfing through hidden {UI} analysis," in *28th USENIX Security Symposium (USENIX Security 19)*, pp. 765–781, 2019.

[13] Q. A. Chen, Z. Qian, and Z. M. Mao, "Peeking into your app without actually seeing it:{UI} state inference and novel android attacks," in *23rd USENIX Security Symposium (USENIX Security 14)*, pp. 1037–1052, 2014.

[14] G. Rydstedt, E. Bursztein, D. Boneh, and C. Jackson, "Busting frame busting: a study of clickjacking vulnerabilities at popular sites," *IEEE Oakland Web*, vol. 2, no. 6, p. 24, 2010.

[15] Y. Yan, Z. Li, Q. A. Chen, C. Wilson, T. Xu, E. Zhai, Y. Li, and Y. Liu, "Understanding and detecting overlay-based android malware at market scales," in *Proceedings of the 17th Annual International Conference on Mobile Systems, Applications, and Services*, pp. 168–179, 2019.

[16] H. Zhou, S. Wu, C. Qian, X. Luo, H. Cai, and C. Zhang, "Beyond the surface: uncovering the unprotected components of android against overlay attack," in *The 31st Network and Distributed System Security Symposium*, pp. 1–17, 2024.

[17] Y. Wu, C. Shi, T. Zhang, P. Walker, J. Liu, N. Saxena, and Y. Chen, "Privacy leakage via unrestricted motion-position sensors in the age of virtual reality: A study of snooping typed input on virtual keyboards," in *2023 IEEE Symposium on Security and Privacy (SP)*, pp. 3382–3398, IEEE, 2023.

[18] C. Slocum, Y. Zhang, N. Abu-Ghazaleh, and J. Chen, "Going through the motions:{AR/VR} keylogging from user head motions," in *32nd USENIX Security Symposium (USENIX Security 23)*, pp. 159–174, 2023.

[19] Ü. Meteriz-Yıldıran, N. F. Yıldıran, A. Awad, and D. Mohaisen, "A keylogging inference attack on air-tapping keyboards in virtual environments," in *2022 IEEE Conference on Virtual Reality and 3D User Interfaces (VR)*, pp. 765–774, IEEE, 2022.

[20] S. Luo, X. Hu, and Z. Yan, "Hololoqger: Keystroke inference on mixed reality head mounted displays," in *2022 IEEE Conference on Virtual Reality and 3D User Interfaces (VR)*, pp. 445–454, IEEE, 2022.

[21] S. Luo, A. Nguyen, H. Farooq, K. Sun, and Z. Yan, "Eavesdropping on controller acoustic emanation for keystroke inference attack in virtual reality," in *The Network and Distributed System Security Symposium (NDSS)*, 2024.

[22] Z. Su, K. Cai, R. Beeler, L. Dresel, A. Garcia, I. Grishchenko, Y. Tian, C. Kruegel, and G. Vigna, "Remote keylogging attacks in multi-user {VR} applications," in *33rd USENIX Security Symposium (USENIX Security 24)*, pp. 2743–2760, 2024.

[23] A. Al Arafat, Z. Guo, and A. Awad, "Vr-spy: A side-channel attack on virtual key-logging in vr headsets," in *2021 IEEE Virtual Reality and 3D User Interfaces (VR)*, pp. 564–572, IEEE, 2021.

[24] A. Nguyen, X. Zhang, and Z. Yan, "Penetration Vision through Virtual Reality Headsets: Identifying 360-degree Videos from Head Movements," in *33rd USENIX Security Symposium (USENIX Security 24)*, 2024.

[25] V. Nair, W. Guo, J. F. O'Brien, L. Rosenberg, and D. Song, "Deep Motion Masking for Secure, Usable, and Scalable Real-Time Anonymization of Virtual Reality Motion Data," Nov. 2023. arXiv:2311.05090 [cs].

[26] V. C. Nair, G. Munilla-Garrido, and D. Song, "Going Incognito in the Metaverse: Achieving Theoretically Optimal Privacy-Usability Tradeoffs in VR," in *Proceedings of the 36th Annual ACM Symposium on User Interface Software and Technology*, (San Francisco CA USA), pp. 1–16, ACM, Oct. 2023.

[27] J. Li, A. R. Chowdhury, K. Fawaz, and Y. Kim, "{Kaleido}:{Real-Time} privacy control for {Eye-Tracking} systems," in *30th USENIX security symposium (USENIX security 21)*, pp. 1793–1810, 2021.

[28] B. David-John, K. Butler, and E. Jain, "Privacy-preserving datasets of eye-tracking samples with applications in xr," *IEEE Transactions on Visualization and Computer Graphics*, vol. 29, no. 5, pp. 2774–2784, 2023.

[29] W.-J. Tseng, E. Bonnail, M. McGill, M. Khamis, E. Lecolinet, S. Huron, and J. Gugenheimer, "The dark side of perceptual manipulations in virtual reality," in *Proceedings of the 2022 CHI Conference on Human Factors in Computing Systems*, pp. 1–15, 2022.

[30] C. Mukherjee, R. Mohamed, A. Arunasalam, H. Farrukh, and Z. B. Celik, "Shadowed realities: An investigation of ui attacks in webxr," in *USENIX Security Symposium*, 2025.

[31] H. Lee, J. Lee, D. Kim, S. Jana, I. Shin, and S. Son, "{AdCube}:{WebVR} ad fraud and practical confinement of {Third-Party} ads," in *30th USENIX Security Symposium (USENIX Security 21)*, pp. 2543–2560, 2021.

[32] Y. Xiu, T. Scargill, and M. Gorlatova, "Viddar: Vision language model-based task-detrimental content detection for augmented reality," *IEEE transactions on visualization and computer graphics*, 2025.

[33] Unity Technologies, *Unity Documentation*, 2025.