AutoVR: Automated UI Exploration for Detecting Sensitive Data Flow Exposures in Virtual Reality Apps

John Y. Kim, Chaoshun Zuo, Yanjie Zhao, and
Zhiqiang Lin, *The Ohio State University*

https://www.usenix.org/conference/usenixsecurity25/presentation/kim-john

This paper is included in the Proceedings of the
34th USENIX Security Symposium.

August 13–15, 2025 • Seattle, WA, USA

978-1-939133-52-6

AUTOVR: Automated UI Exploration for Detecting Sensitive Data Flow Exposures in Virtual Reality Apps

John Y. Kim Chaoshun Zuo Yanjie Zhao Zhiqiang Lin
The Ohio State University

Abstract

The rise of Virtual Reality (VR) has provided developers with an unprecedented platform for creating games and applications (apps) that require distinct inputs, different from those of conventional devices like smartphones. The Meta Quest VR platform, driven by Meta, has democratized VR app publishing and attracted millions of users worldwide. However, as the number of published apps grows, there is a notable lack of robust headless tools for user interface (UI) exploration and user event testing. To address this need, we present AUTOVR, an automatic framework for dynamic UI and user event interaction in VR apps built on the Unity Engine. Unlike conventional Android and GUI testers, AUTOVR analyzes the app's internal binary to reveal hidden events, resolves generative event dependencies, and utilizes them for comprehensive exploration of VR apps. Using sensitive data exposure as a performance metric, we compare AUTOVR with Android Monkey, a widely used headless Android GUI stress testing tool. Our empirical evaluation demonstrates AUTOVR's superior performance, triggering an order of magnitude of more sensitive data exposures and significantly enhancing the privacy of VR apps.

1 Introduction

Virtual Reality (VR) is rapidly emerging as a transformative force in the consumer device market. By 2025, the global VR market is projected to reach USD 32.40 billion, with expectations to soar to USD 187.40 billion by 2030 [13]. This platform offers users a diverse range of interactions, from physical gestures to haptic feedback, delivering immersive experiences that far surpass traditional platforms like smartphones. As the VR audience continues to expand, the need for rigorous testing and quality assurance of VR apps has become increasingly critical. Ensuring user privacy, safety, and security is paramount: not only to safeguard sensitive data but also to provide authentic and trustworthy user experiences.

Despite this urgency, automated testing for VR apps remains underdeveloped [37]. Conventional tools such as Android Monkey [15], while capable of UI testing, struggle with the complexities of 3D engine-based apps, particularly those built on Unity [12], which serves as the backbone for most VR games and applications [9]. Notably, Unity is the only 3D engine officially endorsed by Apple for its Vision Pro VR device [11]. However, the absence of accessible source code for VR apps creates a significant gap in understanding their internal workings, demanding a reverse engineering approach to the Unity VR app binaries [37].

While reverse engineering of Unity-centric apps is hardly new, with established success in static analysis [14, 57] as exemplified by tools like Il2CppDumper [10] in the Unity space, the arena of dynamic analysis tools specifically designed for Unity apps remains vacant. Unity-based apps, such as video games, not only present a complex matrix for security testing due to event dependencies on preceding user actions as well as in-app physical interactions but also contain a broader amount of execution entry points, meaning that there is more than one way to interact with the “main" execution of the app. Therefore, current security testing tools such as UI fuzzing, are often ineffective on Unity-based apps.

Upon detailed examination of Unity app binaries, four distinct observations surface: (1) the Unity app UI structure is embedded within the Unity app binary, instead of traditional markup or configuration files; (2) the Unity UI structure is generative: UI objects and in-app objects become enabled or disabled based on the state of the app execution; (3) the binary's operation on a virtual machine that offers internal APIs, thereby enabling the identification of class, method, and object symbols within the Unity app at run-time; (4) Unity VR apps are highly reliant on physical interactions (e.g., grabbing, hitting, and moving) with surrounding in-app objects, such interactions may contain critical functionality as users are constantly interacting with such objects.

In light of these observations, we present AUTOVR, the first automated UI exploration tool that leverages Unity's unique generative features and internal binary introspection APIs. AUTOVR ambitiously tackles three interrelated challenges from Unity VR apps: (1) the semantic recovery of the Unity

UI information, (2) the modeling of generative Unity UI elements to extract event handlers, and (3) the context-aware execution of these handlers by resolving event dependencies. As a practical demonstration, we utilize AutoVR to detect sensitive data exposures that developers may have embedded within their apps.

We have developed AutoVR on top of Frida [3], an open-source dynamic instrumentation toolkit, compatible with Android devices and, consequently, Meta Quest devices. To gauge the effectiveness of AutoVR, we have evaluated its performance against the widely used tool Android Monkey across the Unity VR apps, evaluating the sensitive data exposures that both tools could invoke. In this empirical assessment spanning 366 apps, including 103 paid apps, AutoVR demonstrated a remarkable capacity to activate 390 **unique** sensitive data exposures, dwarfing the 117 instances (a 2.2× coverage increase) induced by Monkey.

Contributions. We make the following contributions:

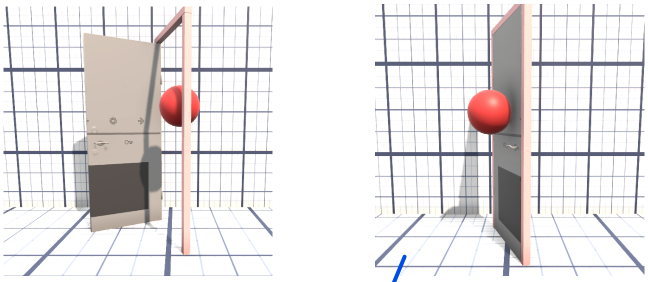
- **Novel Techniques (§4).** We introduce novel techniques including VR *UI semantic recovery*, *generative event modeling* to recover symbols and semantics of Unity VR apps, and *context-aware event execution* of UI elements.
- **Practical Framework (§5).** We have developed these techniques in AutoVR, an open-source[1] framework for automated UI exploration of VR apps, compatible with the ever-changing environment of VR applications.
- **Empirical Evaluation (§6).** Our evaluation with 366 VR apps shows a superior capacity of AutoVR to trigger 2.2x more **unique** sensitive data exposing functions in comparison to widely used tool Monkey.

2 Background

Developing VR apps is a complex task, often requiring specialized tools and support. 3D engines like Unity have made this process much easier by providing essential tools such as real-time rendering APIs, documentation, cross-platform support, and user-friendly IDEs. In fact, Unity is the most widely used game engine within the Meta Quest store [32]. Given its popularity and extensive support for VR development, we have chosen to focus on VR apps developed using the Unity engine.

Scripting Backend. Unity VR apps, much like mobile games, are developed using C#, with the final binary translated into assembled C++ code. Developers may choose from two translation scripting backends: (1) Mono, a just-in-time (JIT) compiler [8], or (2) IL2CPP, an ahead-of-time (AOT) compiler [5]. Unity predominantly emphasizes IL2CPP for its enhanced performance and security.

IL2CPP. The translation of C# code to C++ is a complex process, requiring compatibility with specific C# features such as garbage collection, reflection, and exception handling [22]. Both developer scripts and Unity libraries written in C# must be translated to C++. IL2CPP accomplishes this translation, augmenting each function with additional checks. The resultant assembled native C++ binary, `libil2cpp.so`, has symbols (e.g., class names, method names, and field names) stripped [52], but to support reflection and other features, the symbols are encrypted and stored separately in `global-metadata.dat`, bundled within the APK.



(a) A Trigger Event **(b) A Collision Event**

Figure 1: Illustration of Trigger and Collision Events

Scene. Within Unity, a scene acts as a container for game objects, each defined with specific attributes such as position, rotation, and scale. Managed by the `SceneManager` object [53], scenes are analogous to the UI in Android's `Activity`. Although an app may include multiple scenes, only one is rendered at a time, allowing Unity UI to switch between them to create varied app scenarios.

GameObject. `GameObjects` in Unity [41,51] can represent any object within the game environment, such as players, walls, or UI buttons. Serving as the building blocks of a Unity app, each `GameObject` can host multiple `Component` objects in a modular fashion [50]. These components govern the `GameObject`'s behavior, such as movement or sound, and can range in complexity. Unity provides numerous base components to control app logic, including `Collider` [49] components for collision detection and the `UnityEngine.UI` components for UI handling [17].

UI Events. UI events are typically user-driven and can be triggered through interaction with the VR controllers, such as pointing and clicking using the controller's trigger button. A detailed illustration of UI events can be found in Figure 2. This example presents a series of `GameObjects`—such as the `START` button, `OPTIONS`, `ABOUT`, and `QUIT`—each equipped with a `Button` component that manages their appearance and logic within the VR app. In this example, we assume a developer crafting this user interface in the Unity Editor by attaching a `Button` component [1,45] to each button-themed `GameObject`. While other UI elements also exist, buttons are universally understood and functionally unambiguous.

Physics Events. In addition to UI events, there are physics events that involve interactions with 3D objects within the VR

[1] https://github.com/OSUSecLab/AutoVR

play area and may be associated with critical functionalities, such as scene changes. There are two primary kinds of physics events, as illustrated in Figure 1:

(1) Triggers. Triggers are activated when one `GameObject` intersects with the bounds of another [40, 43]. Unlike collisionable objects, trigger objects are not solid, allowing other `GameObjects` to pass through rather than collide. For instance, a 3D ball object is thrown through an open door. The ball intersects the bounds of the door but is not physically affected by the door. Such an event is classified as a trigger event.

(2) Collisions. Collisions happen when a solid `GameObject` intersects with the bounds of another solid object that is **not** a trigger [40, 43]. For example, a solid enemy hand might interact with a rock, another solid object, by picking it up, touching it, or engaging in other forms of interaction. In contrast to the ball and open door example, suppose the door was closed and is now a solid, non-trigger object. Now the ball physically bounces off the door on intersection, and such an event is classified as a collision event.

3 Overview

3.1 Objective and Scope

Objective. The primary objective of this work is to develop AutoVR, an automated UI testing framework tailored specifically for 3D VR apps developed using Unity. Unlike traditional Android app UI exploration, this framework can effectively test Unity apps, which are packaged as Android Package files (APK). Traditional methods stumble because UI/physics events within Unity apps are concealed within the binary, rendering them neither parsable nor adaptable to standard UI exploration techniques. AutoVR seeks to overcome these barriers, with an extended ambition to employ the framework for enhancing security and privacy. We showcase the effectiveness of our framework by systematically analyzing privacy data exposure in third-party Unity VR apps.

Scope. While there are different 3D engine-based VR apps, this work focuses on VR apps based on the Unity Engine, specifically those utilizing IL2CPP. As mentioned in §2, the majority of VR apps for Quest devices are developed via the Unity Engine—a trend likely to escalate with the introduction of new VR devices like the Apple Vision Pro, which employs Unity for the app development [11]. For this work, we focus on Unity apps compiled using the IL2CPP compiler. Compared to the JIT compiler `Mono`, IL2CPP, an AOT compiler, ensures enhanced performance, a critical attribute for VR apps. Additionally, IL2CPP is anticipated to be used by future Unity-developed VR apps [26].

3.2 Running Example

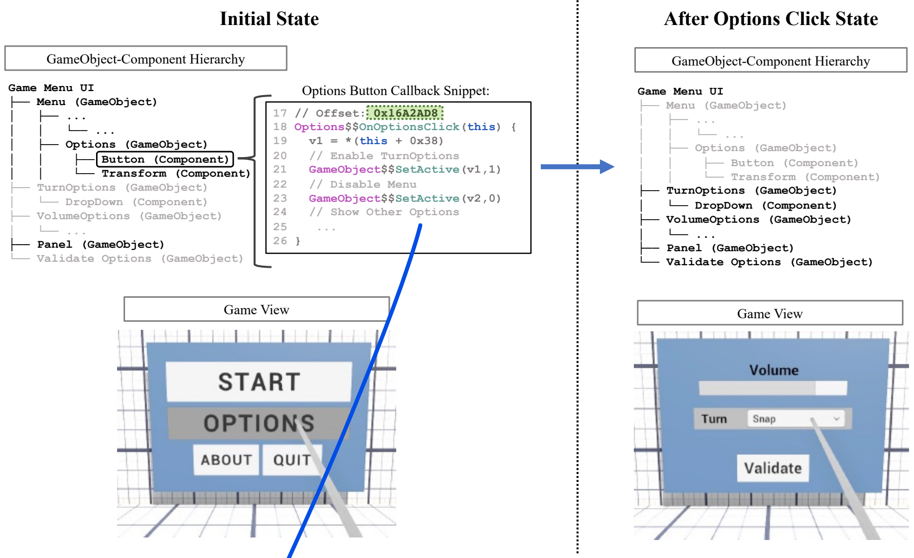
To better understand the Unity UI structure, we depict a VR UI example formulated using the Unity Editor; we have shown this example in Figure 2. The left side of the figure shows the initial state of the UI app before the user clicks the `OPTIONS` button. The right-hand side shows the state of the app after the `OPTIONS` button is clicked. We also illustrate the hierarchical interconnection of `GameObjects` and their corresponding components within the app. This hierarchy may encompass child `GameObjects`. Grayed-out text represents currently deactivated `GameObjects`, whereas bold text signifies enabled ones. The *Game View* segment reflects what a user would perceive within his/her VR device upon executing this app at their respective state. The highlighted portion (i.e., `0x16A2AD8`) indicates the function offsets or virtual addresses of the UI element's function callback.

We also present the code snippet for the `OPTIONS` button's event function callback (EFC). The `OPTIONS` button's EFC (denoted by `Options$$OnOptionsClick`) activates the `TurnOptions GameObject`, along with its associated components, thereby making them visible to the user within the UI. Simultaneously, this action leads to the disabling of the previous `Menu GameObject`, and their subsequent child `GameObjects`, as indicated in line 23 of the "Options Button Callback Snippet".

3.3 Challenges and Insights

(C1) How to Recover UI Semantics. Parsing Unity UI elements differs from typical Android apps as these elements are not directly accessible through the Android app layer nor from a static configuration file such as the `AndroidManifest.xml` file. For instance, when using the Android UI Inspector tool in Android Studio, it shows the UI hierarchy of all Android UI elements. However, when debugging a Unity app, this inspector will not display any UI elements created within the Unity app engine. This is because Unity apps do not embed UI elements through Android activities, instead they are embedded within the IL2CPP game binary.

To solve this challenge, we employ Frida to dynamically instrument the IL2CPP app binary and extract UI elements at runtime. IL2CPP provides a runtime introspection API utilized by the Unity Engine to access detailed class metadata, including class fields and methods, and to create, collect, and resolve object types at any state of the Unity app. While these APIs resemble standard C# libraries, they have been modified by the IL2CPP compiler for optimizations and C++ compatibility. Unlike previous approaches [14, 57] that statically accessed class metadata from the `libil2cpp.so` binary using `global-metadata.dat`, our solution directly invokes IL2CPP API functions from the `libil2cpp.so` binary using Frida to access class metadata. However, while direct introspection can contain accurate symbols, critical symbols are sometimes lost as some functions are called only


Initial State
After Options Click State
GameObject-Component Hierarchy
Options Button Callback Snippet:
Game View
START
OPTIONS
ABOUT
QUIT
Volume
Turn
Snap
Validate

```
Game Menu UI
├── Menu (GameObject)
│   ├── ...
│   │   └── ...
│   ├── Options (GameObject)
│   │   ├── Button (Component)
│   │   └── Transform (Component)
├── TurnOptions (GameObject)
│   └── DropDown (Component)
├── VolumeOptions (GameObject)
│   └── ...
├── Panel (GameObject)
└── Validate Options (GameObject)
```

```
// Offset: 0x16A2AD8
Options$$OnOptionsClick(this) {
  v1 = *(this + 0x38)
  // Enable TurnOptions
  GameObject$$SetActive(v1,1)
  // Disable Menu
  GameObject$$SetActive(v2,0)
  // Show Other Options
   ...
}
```

Figure 2: Running example of a Unity Scene, illustrating Unity UI logic internals.

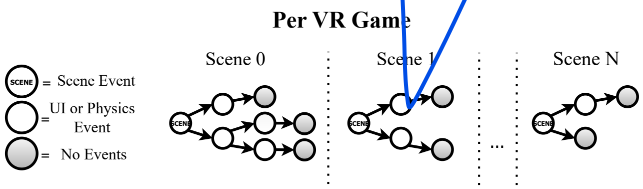


Figure 3: Outline of a scene model per VR game.

through reflection rather than direct invocation. As such, we combine both the static `global-metadata.dat` file along with dynamic introspection, to get a comprehensive list of IL2CPP functions to call.

(C2.1) How to Parse and Model UI Events. To extract and model the Unity UI structure, parsing the Unity UI is necessary. Unlike typical Android apps, where the UI is separate from the app logic, Unity UI elements are integrated directly into the logic and perform like generic `GameObjects`. This relationship is illustrated by the hierarchy in Figure 2, where GUI components like the `Button` are attached to their respective `GameObjects`, much like other objects in an app. Therefore, we face the challenge of collecting UI elements and their callbacks from a generative UI at runtime, rather than relying on static configurations. Furthermore, the extraction of the function callbacks from the UI elements poses an additional challenge: the Unity SDK provides two different ways of assigning EFCs to UI event components: dynamically by using the `UnityEvent.AddListener` API, or static assignment in the Unity Editor.

We identify UI elements and event handling interfaces in Unity by identifying the `IEventSystemHandler` interface in Component classes using the IL2CPP class introspection API. We bypass the time-consuming process of class inspection by directly accessing loaded objects within the app. This allows to only extract the UI elements in the current scene. We then analyze each object at the start of a Scene to identify event system handler objects as UI elements, extracting EFCs. Simply using API hooks for the `UnityEvent` is insufficient, as function callbacks assigned in the editor will not use the `UnityEvent.AddListener` API, nor any API during initialization. As such, to extract the hidden function callbacks, we use IL2CPP's field introspection API to extract the developer's function callbacks, enabling the extraction of UI objects at runtime. We call this process *Generative UI Modeling*.

(C2.2) How to Parse and Model Physics Events. Similarly, we realize physics events are also generative, especially in a VR environment. For example, we notice a popular game *VRChat* utilizes *physics* events to perform scene changes (e.g., when players enter portals to different online rooms). In a VR 3D environment, it is crucial to account for physics, as a significant portion of VR interactions involve the movement and intersection of objects.

Fortunately, physics events are comparatively easier to extract than UI events. As outlined in §2, Unity has two main physics event types: *collision* and *trigger*. Physics events occur when two `GameObjects` (with an attached Collider component) intersect. To trigger these events, `GameObjects` must follow their respective physics type's rules, in order to trigger the correct event. As such, to invoke `Collision` type events, each respective GameObject must contain both a `Collider` and `RigidBody` component. To invoke *Trigger* type events, each component must have an attached `Collider` component, and the `isTrigger` property set to be **true**. Using IL2CPP and Frida, we can use semantic information resolved from **C1** to find `GameObjects` that match their respective physics criteria and set the positions of each `GameObject` to intersect one another. We iteratively perform these intersections on dynamic `GameObjects` within the scene. As such, we similarly label this process *Generative Physics Modeling*.

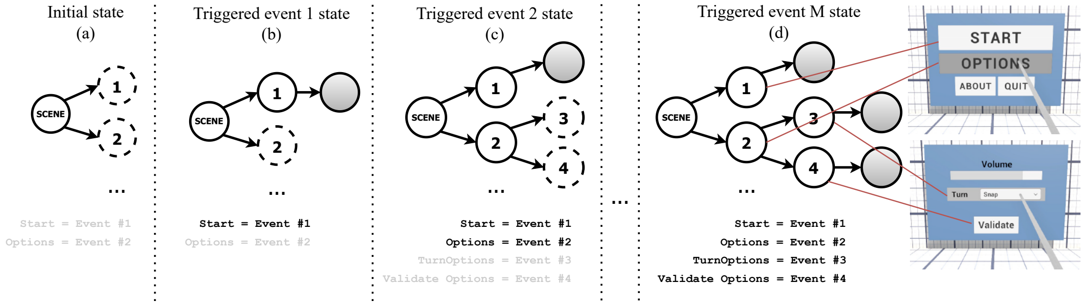


Figure 4: Generative UI-driven model of the running example. This is analogous to Scene 0 in Figure 3, where the modeling process per scene is repeated for every scene in a VR game. The lighter colored text indicates the UI event has been found but not yet triggered, whereas the darkened colored text indicates a the UI event has been triggered.

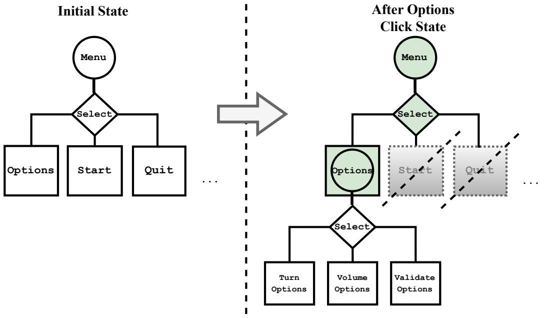


Figure 5: High level event state before and after `Options$$OnOptionsClick` was clicked in Figure 2. Highlighted in green is the path taken, and highlighted gray with dotted outlines are disabled events.

(C3) How to Execute Events and Resolve Event Dependencies. Executing the EFCs from the generative Unity UI also introduces another challenge: dynamic dependencies. Because the UI is generative, it is also possible that triggering EFCs may cause other UI `GameObjects` to be deactivated, or spawn/enable new UI `GameObjects`. For instance, in Figure 2, we notice that the `Options$$OnOptionsClick` will disable/deactivate the Menu UI and all UI elements underneath. However, neither `About` nor `Quit` buttons have been explored. We identify these issues as dynamic dependencies where the spawned or deactivated UI `GameObjects` are dependent on the initial UI `GameObject` executed. This also applies for `GameObjects` that integrate physics events. This dependency model occurs for every scene within a VR game, as such, each scene may contain an entirely different set of events to model. Figure 3 illustrates example event models for each scene per VR game, representing the complete event exploration state of the game. To be complete, execution of as-many-as-possible modeled event function callbacks is necessary and is a challenge we solve for this work.

We solve this challenge by creating a UI-driven generative state model, where the triggering of one such event may cause more unknown events to be spawned or enabled, thus adding to the overall known event state space of a scene. We showcase this process in Figure 4. When the initial scene is loaded (a), the game uncovers two events that have not been triggered. In a depth-first search manner, each event is triggered, and, as a side effect, uncover new unknown (or disabled) events. Additionally, it is possible that no additional events may be uncovered, leading to an empty set of next-events (b). However, unlike a typical tree-traversal problem, the state of the game cannot be easily back-tracked to its previous state, as the invocation of one event may lose the existence of another event. We illustrate this problem in Figure 5. When the `OPTIONS` button is selected, the `START` and `QUIT` buttons become disabled and cannot be further exercised upon, losing valuable events. As such, to solve this, we utilize a synthetic Scene Event that acts as the state reset in the game. This scene event essentially loads/reloads the current scene by leveraging `Frida`'s unique capabilities of function invocation on the IL2CPP binary. Specifically, we notice that Unity's core scene invocation relies on the `LoadSceneAsyncNameIndexInternal` function to load new scenes. Upon the invocation of this function, it is possible to reload the scene back to its initial state, effectively "back-tracking" the scene state to explore a different event path. Using the information gathered by (a) and (b) in Figure 4, we can derive the next sequence of unexplored events to trigger, without losing the information of other events, leading to state (c). Eventually, this process is repeated until every event path leads to an empty set of new events, ensuring the maximum number events are exercised per scene.

4 Detailed Design

In this section, we present the detailed design of AutoVR. We illustrate AutoVR's overall design in Figure 6. There are three key components inside AutoVR: (1) UI Semantic Recovery (§4.1), (2) Generative Event Modeling (§4.2), and (3) Context-aware Event Execution (§4.3). The final output is generated from the plugin application. In this work, our

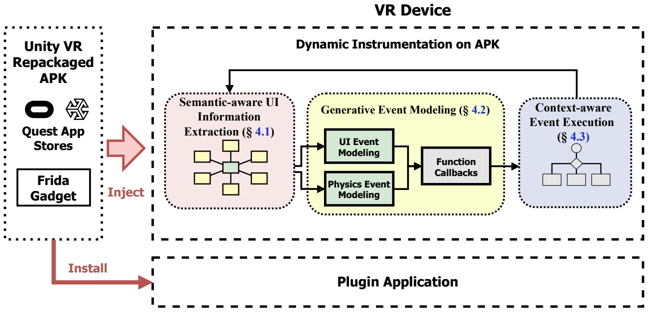

VR Device
Unity VR Repackaged APK
Quest App Stores
Frida Gadget
Inject
Install
Dynamic Instrumentation on APK
Semantic-aware UI Information Extraction (§ 4.1)
Generative Event Modeling (§ 4.2)
UI Event Modeling
Physics Event Modeling
Function Callbacks
Context-aware Event Execution (§ 4.3)
Plugin Application

Figure 6: Overview of AUTOVR

plugin application, *AntMonitor* [28], will produce information on sensitive data flows for each VR app tested.

4.1 UI Semantics Recovery

Identifying events requires knowledge of the UI semantics to eventually retrieve the EFCs within a scene. As such, to collect UI semantics, AUTOVR will first identify the UI components by extracting the class metadata (§4.1.1). Next, to identify the UI function callbacks, and subsequently all objects containing such callbacks, AUTOVR will then extract all available function metadata (§4.1.2). Lastly, to identify which objects contain UI elements, AUTOVR will collect all objects within the loaded scene (§4.1.3). With the class and function metadata, along with object information, AUTOVR eventually recovers the UI semantics of the loaded scene (§4.1.4).

4.1.1 Extracting Class Metadata

The IL2CPP runtime (i.e., libil2cpp.so) extracts and decrypts data from global-metadata.dat. More specifically, class name, function address, return type, and argument information can be extracted from the IL2CPP introspection APIs. In particular, we use APIs from the 9 functions prefixed with il2cpp_class invoked by Frida to collect such metadata. These introspection APIs are detailed in Table 6 (see Appendix) for readers of interest. We notice that class metadata is stored as pointers in memory and dereferencing such pointers can provide rich information from the class metadata, as well as the invocation of its own constructor, child methods, and fields. We also notice these class functions allow to invoke any child methods of the class, create a new instance, or access its field members. This functionality is especially important for UI information extraction, as class metadata is key to identifying UI elements within a VR app.

4.1.2 Extracting Function Metadata

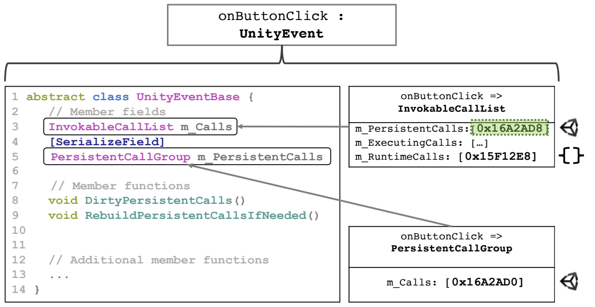
Similarly to the class metadata, the function metadata is also stored in the global-metadata.dat. These function names are denoted using the combination of the function's residing class name along with the function name itself (e.g., in Figure 2: Options$$OnOptionsClick). We notice that the IL2CPP runtime initializes this metadata information once the app is started, storing it in memory. In order for the engine to properly invoke translated functions, and support reflection in C++, there must be generic C++ code to handle function signature extraction and invocation. This extraction is achieved by invoking C++ IL2CPP functions prefixed with il2cpp_method. There are APIs for 7 such functions detailed in Table 6. These functions return or handle references to every method offset from both C++ translation functions (i.e., IL2CPP functions) and translated C# code (e.g., developer functions, and engine packages). AUTOVR essentially leverages these IL2CPP functions to extract function metadata, which will be used for EFC identification. By invoking such IL2CPP functions during the initial startup of the game, we create a global function table (GFT) with the entry address (i.e., the function offset) as the key, and the function's reference handle as the value.

4.1.3 Extracting Objects

To identify which objects are UI components and physics components, AUTOVR uses the IL2CPP runtime library to collect objects currently loaded in the scene using memory snapshots generated by IL2CPP. More specifically, we notice that the IL2CPP runtime can use memory snapshots to identify garbage collection handles that point to objects. This means we can invoke objects and the fields of such objects directly using Frida. Furthermore, we consider accessing field values and types essential for identifying UI events. Event objects are often stored within the fields of the residing object. For example, in Figure 2, we notice that Options$$OnOptionsClick, as well as nearly every UI GameObject, are attached with a Button component. We use two functions prefixed with il2cpp_object in Table 6 to collect values of event objects.

4.1.4 Recovering Unity UI elements

Recall in §2 that the UI elements are embedded in the IL2CPP app binary. As such, it is necessary for AUTOVR to access the libil2cpp.so binary and extract runtime UI objects which are in the form of GameObjects and Components. Using Frida with class metadata extraction as described, it is possible to extract the embedded Unity UI element GameObjects by filtering for UI identifying classes. Unity treats UI elements similarly to every other GameObject within the app. More specifically, the core developer scripts are built and managed using GameObjects, as such, Unity treats UI elements as GameObjects that have attached UI components. For instance, in Figure 2, the OPTIONS button GameObject is a UI element because of the UI component attached to it. However, while it is possible to identify every UI component from the base Unity SDK, developers may also implement their own UI components not derived from Unity UI components.



```
abstract class UnityEventBase {
    // Member fields
    InvokableCallList m_Calls
    [SerializeField]
    PersistentCallGroup m_PersistentCalls

    // Member functions
    void DirtyPersistentCalls()
    void RebuildPersistentCallsIfNeeded()


    // Additional member functions
    ...
}
```

onButtonClick : UnityEvent
onButtonClick => InvokableCallList
m_PersistentCalls: [0x16A2AD8]
m_ExecutingCalls: […]
m_RuntimeCalls: [0x15F12E8]
onButtonClick => PersistentCallGroup
m_Calls: [0x16A2AD0]

Figure 7: Class structure of the UI-based `GameController` component from the running example. The (Unity) 3D box logo indicates that the function callback was attached via Unity Editor, while the bracket symbol indicates the function callback was attached via code.

We notice that the Unity Engine describes all UI events under an event system interface. This interface must be implemented by every UI component, custom or not. We identify the core interface: `IEventSystemHandler` [47]. In Figure 2, every UI button `GameObject` contains an attached Button component which its base class inherits the `IEventSystemHandler` interface. Using the class metadata extracted, AUTOVR identifies all used UI components by extracting the class inheritance hierarchy, and searching for the `IEventSystemHandler` interface.

4.2 Generative Event Modeling

To identify the event function callbacks (EFCs) in a VR app, AUTOVR must identify which objects such events reside. As described in §2 there are two types of `Unity` events: UI, and physics. Each event type must be extracted from the corresponding objects within an app scene. Unity apps are dynamic. UI and physics events are resultant of the initialization of the scene and its `GameObjects`, unlike traditional event models which are hardcoded and static (e.g., Android UI XML model). Unity UI however, will be generated as `GameObjects` are initialized either through scene initialization or event invocation. As such, AUTOVR will first identify all UI event objects from the loaded scene, and collect all EFCs from the UI event objects (§4.2.1). AUTOVR will then identify all available physics events and recover available triggerable and collisionable objects (§4.2.2).

4.2.1 Identifying UI Events

After extracting and filtering the components described, AUTOVR then performs UI event identification. As mentioned in §2, `Unity`'s UI SDK provides base components for developers to modify or inherit. These UI components store callbacks in the form of `UnityEvent` objects. `UnityEvent` objects are often stored within the fields of the residing component, or in the form of a derived class. From Figure 2, the `OPTIONS` Button component contains a `UnityEvent` called `onButtonClick` within the fields of the Button class.

Unfortunately, `UnityEvents` themselves are not the function callbacks that contain developer logic. `UnityEvents` are structured so that multiple function callbacks can be attached to one `UnityEvent` in a many-to-one representation. Furthermore, `UnityEvents` distinguish their function callbacks between runtime callbacks and persistent callbacks. Runtime callbacks are temporary callbacks that are added during the execution of the app through developer code. For example, in Figure 2, the `Start$$Press` EFC is assigned to the `START` Button via Unity SDK API call: `UnityEvent.AddListener`. Persistent callbacks however, are serialized callbacks added prior to the execution of the app, typically within the Unity Editor (e.g., `Options$$OnOptionsClick` in Figure 2 and Figure 7). We notice that persistent callbacks are not added through an API unless the player of the app invokes the corresponding event (e.g., through a button click). As such, API hooks alone will not be able to recover the persistent callbacks of the `UnityEvent`.

Nevertheless, we notice that the base `UnityEvent` class, `UnityEventBase`, contains two fields: `m_Calls`, and `m_PersistentCalls`. These fields are of type `InvokableCallList` and `PersistentCallGroup`, respectively. Both of these fields store function callbacks into containers assigned to their respective `UnityEvent` as shown in Figure 7. `InvokableCallList m_Calls` stores the runtime callbacks and some persistent callbacks, while `PersistentCallGroup m_PersistentCalls` stores the remaining persistent callbacks assigned by the developer at compile time. These callback containers contain the callback function metadata. This metadata provides the callback function's name, virtual address, argument cache, and the object target (i.e., the `UnityEvent` object). Using Frida, AUTOVR is able to extract the values of these callback containers, and thus, the callback functions themselves, using the functions from Table 6.

4.2.2 Identifying Physics Events

As described in §2, there are two types of physics events: **collisions**, and **triggers**. Fortunately, physics events are slightly easier to identify than UI events, as the EFCs are not hidden within field objects. However, physics events must follow specific rules dictated by the game engine, these rules are shown in Table 1. There are three important Collider properties to help identify physics interactions, Rigidbody attached, Static property, and Kinematic property [42]. Rigidbodies can be attached to a Collider component to apply physics motions, static property indicates the collider is a non moving object but allows physics events to take place, while the kinematic property indicates the collider behaves static but also allow the movement of the collider object. In the following, we describe how AUTOVR identifies and extracts such collider objects to execute physics events.

	Trigger Colliders			Collision Colliders		
	RB	Static	Kinematic	RB	Static	Kinematic
RB	✗	✗	✗	✓	✓	✓
Static	✗	✗	✓	✓	✗	✗
Kinematic	✗	✓	✓	✓	✗	✗

Table 1: Rule matrix of two colliders executing physics events where ✓ indicates event is executable, and ✗ not executable. RB = colliders with Rigidbody attached, Static = colliders without a Rigidbody, Kinematic = RB colliders with the kinematic property set to true.

Trigger Events. A trigger event occurs when a **non-solid** game object (i.e., a game object that does not contain a `Rigidbody` component) intersects with other non-solid game objects. We call these non-solid game objects **triggerables**. There are three properties that determine if a game object is an invokable triggerable:

(1) A collider is attached to the game object.
(2) The collider component must have `isTrigger` set to **true**.
(3) At least one of the game object's components implements at least one of the following trigger functions: `OnTriggerEnter`, `OnTriggerStay`, and `OnTriggerExit`.

To identify other collider objects to invoke these triggers with, we present the Trigger Collider matrix from the 2nd to the 4th column in Table 1. Collider components may contain a Rigidbody to allow physics movements to be applied to its host GameObject. This is also how we identify 'solid' objects as mentioned in §2. However, trigger events are simply intersections between Collider components that do not contain any physics movement properties. As such, only colliders without a Rigidbody attached may be used with trigger events.

To identify all invokable triggerable game objects, AUTOVR uses the components described in §4.1.3 to filter out components that do not have the three properties, and extract the virtual addresses of the trigger functions. These virtual addresses are then sent to our *Dependence Resolution* (§4.3.2) to resolve dependencies. Furthermore, to identify which Collider components may interact with such triggerable events, we filter out components using the Trigger Collider rule matrix from Table 1.

Collision Events. A collision event occurs when a **solid** game object (i.e., game object contains a `Rigidbody` component) collides with other solid game objects. We call these solid game objects **collisionables**. There are four properties that determine if a game object is an invokable collisionable:

(1) A collider is attached to the game object.
(2) A `Rigidbody` component is attached to the game object.
(3) The collider component must have `isTrigger` set to **false**.
(4) At least one of the game object's components implement at least one of the following collision functions: `OnCollisionEnter`, `OnCollisionStay`, and `OnCollisionExit`.

To identify other collider objects to invoke these collision events with, we present the Collision Collider matrix from the 5th to the 7th column in Table 1. However, unlike trigger colliders, at least one of the two intersecting colliders must have a Rigidbody attached to it. As a result, AUTOVR will filter for collider objects that follow the Collision Collider matrix when invoking the current invokable collisionable function, and extract the virtual addresses of these collision functions for the follow-up analysis.

4.3 Context-aware Event Execution

Once AUTOVR collects the initial events, the execution can be done using the extracted function offset and invocation by Frida. However, as outlined in §3.3, the execution of events may cause more events to be enabled and/or other events to be disabled. These dependencies depicted in Figure 4 demonstrate that when the `OPTIONS` button is clicked, the `START` and `QUIT` buttons become disabled, while the `TurnOptions`, `VolumeOptions`, and `ValidateOptions` buttons are enabled. To resolve such dependencies from event execution, AUTOVR will first execute the first found initial event from the loaded scene, then once executed, AUTOVR will attempt to find dependencies from the new state (§4.3.1). Subsequently, AUTOVR will then resolve the identified dependencies if there are any (§4.3.2).

4.3.1 Event Execution and Dependency Identification

To cover and execute EFCs, AUTOVR performs context-aware event execution by analyzing dynamic dependencies of all identified EFCs collected. More specifically, once all the EFCs are extracted, execution of events can be performed through direct invocation using Frida along with the GFT to extract the function offset and parameters. To find object parameters, we retrieve all currently available objects (collected upon scene initialization) to find any active objects to test the EFC with. At this stage, possible plugin applications such as network traffic collection applications (e.g., AntMonitor [28]) can be used to observe the effects of executing the identified EFCs. Initially, the first EFC is chosen at random; however, as mentioned in §3.3 and illustrated in Figure 4, it is possible that executing the EFC causes other EFCs to be enabled or disabled. In such a case, we identify these newly changed or added EFCs as dependencies, and map a tree of events to model their dependencies.

4.3.2 Dependency Resolution

To cover the entire tree from Figure 4, AUTOVR will maintain a state machine, identifying every new event linked with its parent. An example is the recovery of the initial state to recover the disabled events in Figure 4. There are two cases of recovering state:

- The parent EFC is a root event of the current scene. In this case, recovering the initial state is necessary to recover disabled events. As such, AUTOVR will reload the current scene to recover the disabled callbacks and execute on a different event path.
- The parent EFC is a child event of the current scene. In this case, to recover the events generated by this parent EFC, AUTOVR invokes the parent EFC again without reloading the scene. Therefore, the disabled events in this level will be recovered and callbacks will execute on a different path.

Using these recovery methodologies, the event dependencies will be resolved. This way, the sequence of events can be covered holistically, and all dependencies can be resolved. As many VR apps, such as video games where the state of the game is important to covering every execution path, we must be sure to cover dependencies and the sequence of events to resolve such dependencies.

5 Implementation

Dynamic Instrumentation. We have implemented a prototype of AUTOVR atop Frida as our main dynamic instrumentation engine. Frida contains multiple modules with both JavaScript and Python bindings. In particular, we use Frida's `Interceptor` [4] module to perform API hooks on function addresses, `NativeFunction` to perform invocation of such function addresses, and `Instruction` module to reassemble ARM64 instructions from such function addresses. We have developed AUTOVR with over 3,000 lines of TypeScript code and over 1,000 lines of Python code.

To perform dynamic instrumentation to collect and invoke events, AUTOVR uses Frida as the dynamic instrumentation toolkit and interfaces with the functions shown in Table 6. Frida is the only dynamic instrumentation toolkit and supports non-rooted Android devices, which is crucial as Quest 2 devices are typically root locked. Therefore, for non-rooted Quest devices, we must inject the Frida server binary into every tested app. We use `frida-gadget` [16], a Frida server binary that can be injected directly into the game source files. We then use `objection` [38] to repackage the APK with the injected Frida server binary, and install the modified APK into the Quest 2 device using `adb`.

Application: Sensitive Data Detection. AUTOVR supports various security applications, including crash detection and malware analysis, but this work focuses on detecting sensitive data exposure. As an event exploration and dynamic analysis framework, AUTOVR is application-agnostic. For sensitive data detection, AUTOVR employs an adapted version of *AntMonitor* [28] to intercept outgoing TLS traffic from VR apps. However, SSL pinning in third-party Unity apps presents challenges, as it encrypts network traffic and obscures potential exposures.

To address this, AUTOVR integrates *AntMonitor* for TLS traffic capture and employs decryption techniques to bypass SSL pinning in both Android and Unity layers. While bypassing SSL pinning in Android apps is well-documented [20], Unity apps introduce additional challenges. Older Unity versions use the `mbedtls_x509_crt_verify_with_profile` function for verification [48], but apps released after 2021 often utilize `x509_crt_verify_restartable_ca_cb`. To handle this, AUTOVR supports bypassing the new verification function using Frida for API hooking, nullifying flags and return values.

During experiments, *AntMonitor* collected network traffic while AUTOVR triggered sensitive data flows through automated event execution, unlike prior works like OVRseen [48], which relied on human-triggered events. While AUTOVR enables the discovery of otherwise hidden data flows, not all such flows constitute privacy leaks, as developers may disclose them in privacy policies.

6 Evaluation

In this section, we present the evaluation results of AUTOVR. First, we outline our experiment setup (§6.1), followed by its effectiveness including the capability for detecting privacy data exposure to current dynamic analysis tool Android Monkey (§6.2). Finally, we present the efficiency overhead of AUTOVR (§6.3).

6.1 Experiment Setup

A Custom VR Unity App. To show the effectiveness of AUTOVR's event exploration and execution capabilities, we have developed a custom Unity VR app that contains both UI and physics events and their dependencies. To accurately test event exercising, we develop the app by ourselves so that we know exactly the number of scenes, events, and UIs and we can also easily instrument the app to log the event exercise behaviors. In particular, within the app, we developed three scenes: (1) only UI events, (2) only physics events, (3) a combination of UI and physics events. To show the dependency structure of such scenes, we present Figure 8, where a screenshot of each scene is shown with the events enabled to clearly illustrate the structure. Additionally, to test whether AUTOVR can execute an event beyond the screen, we placed event 4 from scene 1, and events 4 and 6 from scene 3, outside the default field of view.

Unity game acquisition. To test AUTOVR across the VR app ecosystem, we collected 263 free games in total from both the Meta Quest app store and the SideQuest app store. SideQuest is a third-party endorsed app store, typically where developers publish apps or games for facilitating distribution or for early access releases. Specifically, we have scraped

Scene 0												
Event ID	**1**	**1a**	**2**	**2a**	**2b**	**2c**	**2d**	**3**	**3a**	**3b**	**3c**	**4**
AUTOVR	41	40	41	40	39	40	39	41	40	39	39	41
Monkey	0	1	0	0	0	0	0	0	0	0	0	0
Scene 1												
Event ID	**CCube#1**	**CCube#2**	**CCube#3**	**CCube#4**	**CCube#5**	**CCube#6**	**CCube#7**	**CCube#8**	**CCube#9**	**TCube#1**	-	-
AUTOVR	47	212	355	249	243	200	108	619	503	9	-	-
Monkey	0	0	0	0	0	0	0	0	0	0	-	-
Scene 2												
Event ID	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**CCube#1**	**CCube#2**	**TCube#1**	**TCube#2**	-
AUTOVR	19	19	19	4	18	3	17	133	133	10	10	-
Monkey	2	0	1	0	0	0	0	0	0	0	0	-

Table 2: Total number of events triggered by AUTOVR and `Monkey`, grouped by scene number from Figure 8. Physics events (e.g., CCube#1, TCube#1) sum up the total events triggered from all three callbacks (e.g., `On(Trigger/Collision)Enter`, `On(Trigger/Collision)Stay`, `On(Trigger/Collision)Exit`).

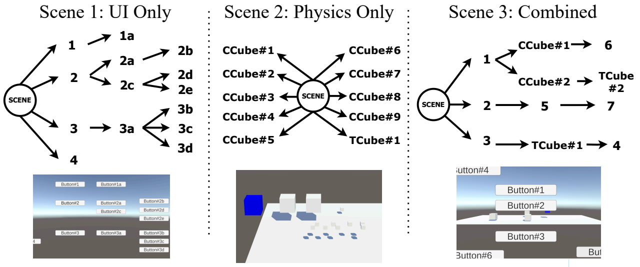


Figure 8: Custom VR app describing the dependency structure for each scene, where each alphanumerical value (e.g., 1, 2d, 3a, etc.) indicates a UI event, each "Cube" prefixed with "C" indicates a collisionable event and "T" indicates a triggerable event.

84 games from the free Meta Quest store and 179 games from the SideQuest store. Additionally, we collected 103 paid games from the Meta Quest app store, totaling 366 games. Scraping Meta Quest and SideQuest resulted in metadata acquisition from each individual game collected. Such metadata includes app rating, number of downloads, category keywords, and privacy policy links.

Experiment Environment. AUTOVR was deployed in an Ubuntu 21.10 LTS desktop environment, running on a machine equipped with an Intel i7-8750H processor, 4 CPU cores, and 16 GB of RAM. While more processing power could be allocated for AUTOVR, the primary bottleneck in our tests was the event modeling and execution process, which runs on the Quest 2 devices. The 4 used Quest 2 devices runs on 8 CPU cores from a Qualcomm Snapdragon XR2 processor and includes 6 GB of RAM [33].

6.2 Effectiveness

Custom VR App Verification. We first verified that AUTOVR effectively executes and accurately identifies scenes, `GameObjects`, as well as UI, collision, and trigger events within the custom VR app. The goal state being, AUTOVR, traverses all events of each scene.

We present the results of AUTOVR running on the custom VR app in Table 2. We first notice a sizable number of collision events triggered. Specifically, for each collision event, the `OnCollisionStay` EFC is vastly higher than the `OnCollisionEnter/Exit` EFC. Due to the necessity of moving collider components to different positions to trigger collision events, a short delay (300 ms) is introduced between collisions to ensure collision event invocation. `OnCollisionStay` follows the `FixedUpdate` time rate (i.e., 20 ms per call [44, 46]), as such, each collision event will call `OnCollisionStay` at least 15 times (300ms / 20ms). Additionally, for each collision event, AUTOVR will attempt to collide all collisionable `GameObjects` with the acting collider within the scene, further bloating the occurrences.

Futhermore, as shown in Table 2, AUTOVR was able to comprehensively invoke all given UI and physics events while also traversing the internal scenes. AUTOVR was able to invoke events within a nested state (i.e., triggered all events with dependencies), as well as explore the dependencies of interleaving physics and UI events. Additionally, AUTOVR was able to invoke events 4 (scene 1), 5 and 6 (scene 2), which are **outside** the field-of-view of the default headset position. Note that all invocations are recorded by the custom app. Because AUTOVR attempts to trigger an event on all available threads, it is highly likely that AUTOVR invokes the same event multiple times per execution. Moreover, we notice fewer invocations the deeper the UI event lies within the dependency tree (e.g., events 4 and 6 from scene 2). The same initial events will be invoked more times than the nested ones due to the backtracking behavior described in §4.3.2.

VR Apps from App Stores. Next, we ran AUTOVR on the corpus of 366 VR apps collected. The aggregated results for this experiment are presented in Table 3. Specifically, we aggregated the runtime data results based on the app's rating listed on the MetaQuest and SideQuest stores. The metadata from the SideQuest and Meta Quest stores contain largely inconsistent data keys, with some intersection, specifically the user ratings, as such, we use ratings as the primary key to describe the experiment results. We observe that the category comprising apps with ratings ranging from 4.75 to 5

	Free Apps							Paid Apps						
Rating	**# Apps**	**# Scenes**	**# GameObject**	**# UI Events**	**# Collisions**	**# Triggers**	**Time (s)**	**# Apps**	**# Scenes**	**# GameObject**	**# UI Events**	**# Collisions**	**# Triggers**	**Time (s)**
[2.5, 2.75)	23	57	61,357	913	494	775	3353.20	3	4	1,536	0	0	13	205.21
[2.75, 3.0)	2	14	9,778	28	0	147	175.89	1	28	7,722	0	0	54	472.36
[3.0, 3.25)	6	18	35,914	82	780	281	1956.54	1	11	6,826	5	0	2	243.72
[3.25, 3.5)	9	17	29,162	2,896	0	7	1176.12	3	48	42,789	0	0	2	807.12
[3.5, 3.75)	20	99	65,731	11,320	170	541	3696.56	5	17	12,016	99	2	877	861.57
[4.0, 4.25)	39	158	158,823	10,879	191	3116	17360.33	14	45	66,106	130	230	266	3126.51
[4.25, 4.5)	39	158	222,434	6,474	1139	2774	10238.70	18	30	49,873	1,483	0	107	3111.51
[4.5, 4.75)	41	187	327,299	1,999	1114	4082	5811.47	29	194	210,346	4,396	451	520	6590.65
[4.75, 5)	84	426	439,990	22,118	4967	5466	16773.521	29	217	115,270	1,338	341	632	5437.94

Table 3: Aggregated data for Meta apps & SideQuest apps based on ratings, separating paid and free games.

(denoted as [4.75, 5)) represents the largest subset within our dataset and, consequently, contains the highest number of `GameObjects`. The table also reveals a direct correlation between the number of `GameObjects` and the number of both UI and physics events (collisions and triggers). This correlation is expected, as events are inherently tied to `GameObjects` within a scene. Therefore, apps with a greater number of `GameObjects` are likely to feature more event-associated `GameObjects`. While a noticeable correlation exists between the number of scenes and the number of `GameObjects`, the latter is primarily influenced by the developer's design choices within each scene. For example, the paid apps under [4.5, 4.75) range contains more `GameObjects` than paid apps under [4.75, 5), however, the scene count for [4.75, 5) is larger than the [4.5, 4.75) range. The same can be said for [4.0, 4.25) and [4.25, 4.5) paid apps.

From both tables, we also notice a direct correlation between the number of collision and trigger events to the amount of time taken for execution. Due to the necessity of moving collider components to different positions to trigger collision events, a short delay (300 ms) is introduced between collisions to ensure collision event invocation. Otherwise, the game engine will not be able to handle too many collisions in a short period of time, causing the game to crash. As collider components are highly prevalent in VR apps, which will produce more collisions and triggers, the time taken to execute AutoVR on VR apps is significant.

Sensitive Data Exposure Detection. Next, to quantitatively evaluate the effectiveness of applying AutoVR for sensitive data exposure detection, we showcase Figure 10. The X-axis represents the types of sensitive data exposures, while the Y-axis indicates the number of occurrences found in our total corpus of VR apps. Unsurprisingly, the Unity version is the most popular data exposed by all the apps, which is typically used for game metrics sent to the developer. The next three top data types (i.e., `APP_INFO`, `PLATFORM_INFO`, `SESSION_DATA`) are potentially used for digital fingerprinting [48]. Additionally, data types such as `SCREEN_INFO`, `GPU_INFO`, `CPU_INFO`, and `DEVICE_INFO` are collected to increase fingerprinting entropy, commonly used to accurately track user behavior and identifiablility [21, 48]. Throughout our experiments however, we noticed that `USER_ID` and `DEVICE_ID` are the most stable identifier amongst the VR apps, and is one of the common data points we see in Figure 9.

We notice from Figure 10 that there are large discrepancies in data flows for free and paid apps (e.g., USER_ID and DEVICE_ID). Many of the paid apps were generally not exposing sensitive data, as we collected exposures from only 56 apps. Additionally, we noticed application-side encryption in many of the paid app's network packets. As these could not be automatically decrypted beyond just SSL/TLS decryption, much of the data could not be interpreted. We understand that paid apps are generally well built, as most of the paid apps are within the 3.5 rating+ range (see Table 3), as such, application-side encryption would likely be integrated into the apps. Free apps, conversely, generally rely on the free in-house tools provided by Unity to send network traffic, and many apps have integrated Unity's in-house analytics to send data regarding user behavior (e.g., in Figure 14, we notice such analytical traffic where "userid" and "deviceid" data pairs are being sent). The two top outgoing hostnames (*cdp.cloud.unity3d.com* and *perf-events.cloud.unity3d.com*), are specifically used for analytics and performance. Additionally, there are more free apps in our corpus than paid apps, further inflating the number of data exposures found.

Furthermore, to quantify the correlation of the total unique sensitive data flows to the destination hostname, we present Figure 9. We notice that majority of the data flows reach a *unity3d.com* domain, Facebook domain (i.e., *fb.com* and *facebook.com*). As such, we grouped the third-party domains into one entity. We denote 3rd party domains as any domain not associated with Meta, Facebook, or Unity. We identify that significant traffic of analytic and device data outgoing to *cdp.cloud.unity3d.com* and *perf-events.cloud.unity3d.com*. *perf-events.cloud.unity3d.com* is typically used for social features, where analytical data is needed to support such features [48]. Additionally, other network traffic outgoing to `*.cloud.unity3d.com` could be used by the developer to obtain analytical or device information. We show an example network packet in Figure 14 in the Appendix.

Comparison with Monkey. To compare the results of AutoVR with Monkey, we performed the same data collection process used for AutoVR to Monkey, using *AntMonitor* to collect network data and Frida for event trigger data collection.

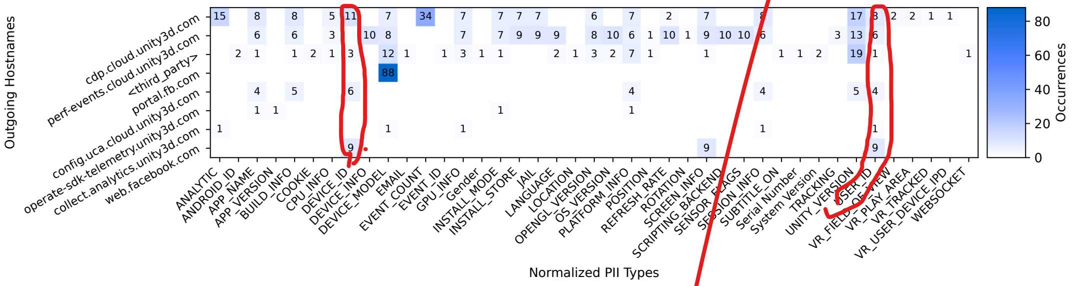


Figure 9: A heatmap showing unique occurrences of sensitive data flow types per app rating range of both free and paid Meta Quest apps and SideQuest apps.

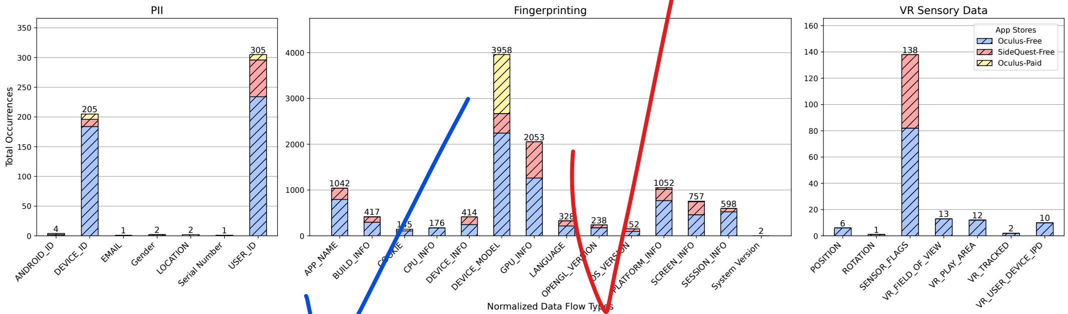


Figure 10: Total number of sensitive data flow occurrences grouped by app store, categorized by PII (Personal Identifiable Information), Fingerprinting, VR Sensory Data.

We allotted Monkey 20 minutes of running time and set AUTOVR to a 20 minute timeout to fairly assess the comparison.

We present a comparative analysis of the detected sensitive data exposure by Monkey and AUTOVR in Figure 11. Both tools were ran against the 366 apps. AUTOVR has collected 390 **unique** data flow exposures, whereas Monkey has gathered only 117. This represents a 2.2x improvement in finding **unique** sensitive data flow exposures.

Uniquely, AUTOVR is capable of capturing more unique PII data flows, such as `EMAIL`, `Gender`, `LOCATION`, and `Serial Number`. Uninspiringly, AUTOVR was able to trigger more sensitive data flow exposures, as AUTOVR's context-aware execution allow the game to enter more states as more events are triggered. Furthermore, because AUTOVR also traverses through the app scenes, which allows AUTOVR to load new events in different levels that `Monkey` cannot access. Therefore, AUTOVR will visit more `GameObjects`, and trigger both 3D collision/trigger and UI events, leading to more outgoing data flows.

Furthermore, AUTOVR is context-aware, taking into account various game states such as scene iterations and the components of the scene (e.g., `GameObjects`). This enables it to trigger a greater number of events, thereby leading to the identification of more privacy exposures compared to Monkey. The comparative performance of Monkey and AUTOVR is additionally detailed in Table 2. We notice that AUTOVR executes all events (i.e., UI events and physics events) far more frequently than Monkey. As such, a correlation can be determined by the number of privacy exposures detected by each tool versus the number of events executed, highlighting the advancements that AUTOVR has made over Monkey.

6.3 Efficiency

For the 366 VR apps, AUTOVR took a total of 81,398.92 seconds with an average running time of 222.40 seconds each. It is obvious that executing a random pre-computed set of events (Monkey) would have significantly shorter runtimes than a generative parsing and execution event model would (AUTOVR).

From Figure 12, we notice a positive correlation between the number of `GameObjects` and runtime and an event stronger positive correlation between the number of scenes to the total runtime. While there is a clear correlation to the number of scenes and total objects to the running time, the event execution strategy also plays a factor. AUTOVR executes events in a DFS manner, and for each newly found object, another DFS of the object's class fields is also performed for event identification. We show our calculation for the worst case running time in Equation 1, where E = number of events, P_c = collisionable objects, P_t = triggerable objects, C_t = class types, C_f = fields per class type, D = event dependencies. Because we are not memoizing $(C_t + C_f)$, after each event, AUTOVR must recalculate $(C_t + C_f)$ after each event invokes. As such, the running time scales higher depending on the combination of the three events.

$$O(E * (P_c + P_t + (C_t + C_f)) + D) \tag{1}$$

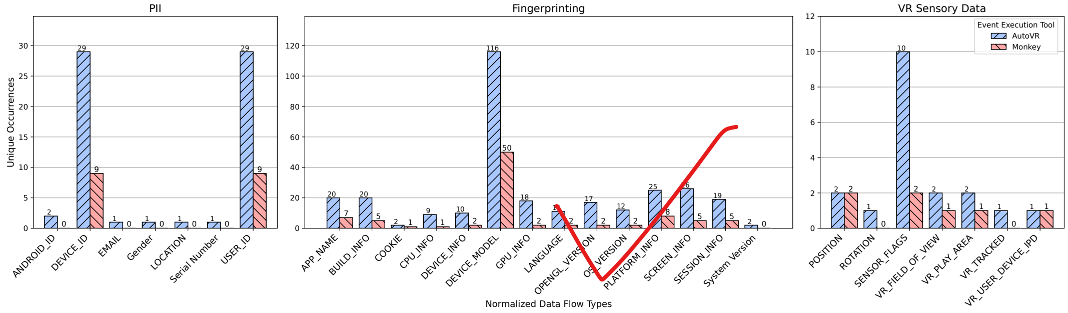
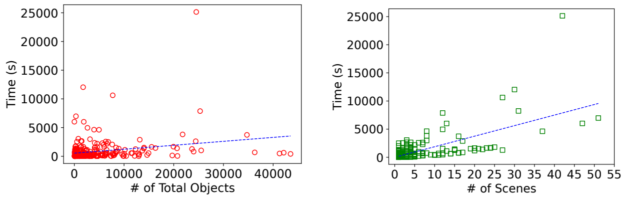


Figure 11: Comparison of the effectiveness with respect to the number of unique sensitive data flow occurrences found using AUTOVR vs Monkey.



(a) Objects vs Time **(b) Scenes vs Time**

Figure 12: Relationship of (a) total detected objects and (b) total scenes, to the total running time of AUTOVR for each VR app in our corpus, where (a) has a positive correlation coefficient of r = 0.24 and, (b) a stronger positive correlation coefficient r = 0.6721.

To quantify the efficiency in relation to the scene traversal events, EFC extraction, and event execution, we present Table 4. We initially notice scene loading significantly contributes to the runtime, as AUTOVR waits 5 seconds for each time a scene loads, as such, the number scenes will scale with running time. Physics-based events are a significant contributor to the total running time of (2). For each physics event, all combinations of the available collisionable/triggerable `GameObjects` are executed with the EFC. In this case, $P_c + P_t$ is larger than $(C_t + C_f)$, which is reflected in the stage's runtime. When bloating the number of UI events, the UI execution becomes the primary contributor to the total running time of (3). Notably, the UI execution from (3) starkly increases from (1)-(2). Because $(C_t + C_f)$ is recalculated after every UI event, the total $(C_t + C_f)$ will be significantly larger than both P_c and P_t, resulting in the largest contributor to running time in (3).

7 Discussion, Limitations, and Future Work

Privacy Implications of Sensitive Data Flows. While AUTOVR was successfully able to extract sensitive data flows from VR apps, tracking the privacy implications requires knowledge of the context (e.g., purpose, notice and consent, law, etc.) and type (e.g., PII, fingerprinting, cookies, etc.) of data flows. This is because the data flows themselves do not determine a privacy concern, rather, a labeling of the outgoing data. For instance, as shown in Figure 13, the outgoing network packet finds an email and password, however, this data relates to the developer's PII data and not the user's. As the data flow collected is PII from the developer, this is not necessarily a privacy concern for the user. Additionally, the outgoing hostname is also relevant since first-party and third-party contexts are also needed. In Figure 14, we notice the sensitive data flows are outgoing to `Unity` servers, a first-party context, not necessarily a privacy concern without the purpose of the data being collected. As such, additional analysis must be performed with context to identify the privacy implications, which is outside of the scope of AUTOVR.

	Running Time (ms)		
Stage Name	*(1)*	*(2)*	*(3)*
Scene Loading	32,497	32,536	32,516
EFC Identification	859	1,280	1,196
UI Execution	9,074	9,034	46,833
Triggers Execution	21,064	86,220	21,093
Collisions Execution	22,129	91,072	22,151
Miscellaneous	11,286	12,826	34,989
Total	96,909	232,968	158,778

Table 4: Relationship of time consuming AUTOVR stages to run time in milliseconds across three versions of the custom test app: (1) no changes, (2) with bloated physics events, (3) with bloated UI events. For (2) and (3), the bloated events are 3x from (1), including dependencies.

Nevertheless, to show that AUTOVR is still useful to help privacy concern analyses, we have collected the privacy policies from our corpus of VR apps and cross-compared them with the collected sensitive data flows using OVRSeen's improved PoliCheck [19, 48] tool. The results of which are shown in Table 5 in the Appendix for interested readers. Only 158 of our corpus of 366 VR apps contained privacy policies, and only 44 VR apps were able to be analyzed. We notice a significant number of results of *vague* and *omitted* network-to-consistency values from the privacy policies versus the traffic being collected. *Omitted* (i.e., data type not found in the privacy policy) and *vague* results (i.e., data collector is vague in the privacy policy), are an indication of leaking data flow, however, without the purpose of the data being collected, one cannot conclude that it is indeed a privacy concern.

While the VR ecosystem continues to develop, there is still a lack of readily available privacy policies within both the Meta Quest and Side Quest app stores. In 2024, Guo et al. [25] has collected 900 VR apps from the SideQuest app store, and have found only 44.8% contain privacy policies. Similarly, amongst our corpus of 366 VR apps, only 43.2% contain privacy policies. Not only is there a considerable lack of privacy policies within the VR ecosystem, but also a significant number of inconsistent data collection policies found by OVRSeen [48]. Yet, without these privacy policies, privacy analysis becomes significantly more challenging as this critical context is missing from the overall analysis.

Comparison with state-of-the-art. AUTOVR effectively exercises events within a VR game. Because these VR games are essentially APKs, it is only possible to exercise events using Android-supported event tools created before AUTOVR. Monkey, for example, is one such a tool. Similarly, existing state-aware UI exercising tools such as DroidBot [29] and AutoDroid [55] also utilize the same triggering functionality as Monkey by using MonkeyRunner [7] as the engine for triggering events. As MonkeyRunner triggers UI events by simulating screen-level human interactions (e.g., touch, swipes, double-taps), which are inconsistent event types within a VR environment, event invocation becomes much more difficult. Inherently, any Android-based event exercising tool are limited to 2D space, unlike VR, where events are expressed in a 3D space (e.g, event occurs **behind** the user's viewpoint will be unknown to traditional tools). Therefore, state-of-the-art tools such as DroidBot and AutoDroid consequently fail for VR apps. AUTOVR, however, does not run into described problems. AUTOVR's automation relies on instructing events on the IL2CPP binary level, overcoming the screen image and VR controller dependency that current state-of-the-art tools run into. As such, AUTOVR significantly advances UI automation and dynamic analysis for VR games/apps.

Integration of symbolic execution. AUTOVR provides the foundation of event identification and execution. This lays the groundwork for symbolic execution. Consequently, because the Unity Engine highly utilizes custom object types as the basis of the SDK, `UnityEvent`s also support object types as inputs to execute events. However, identifying such inputs requires a deeper look into solving event constraints and is beyond the scope of this work. We foresee AUTOVR to expand and support solving. An SMT solver (e.g., the Z3 solver [56]) will need to be integrated with EFCs that contain parameters that can be abstracted and taken as input for the SMT solver. Encouragingly, AUTOVR event execution and state-modeling provides a first step towards understanding the dynamic paths of a game, opening up an opportunity for further investigation.

8 Related Work

Privacy and Security in VR. Privacy risks in VR and head-mounted displays (HMDs) have been studied since the technology's inception [18, 23, 54]. Trimananda et al. [48] audited network traffic to identify sensitive data risks, revealing that 70% of Meta VR apps' data flows were not disclosed to users. O'Brolcháin et al. [35] explored ethical concerns, including privacy threats, social manipulation, and the blurred lines between real and virtual worlds. George et al. [23] analyzed the usability and security of traditional authentication methods in VR, while Lou et al. [31] demonstrated real-time facial reconstruction through sensory input. AUTOVR complements these works by providing an automated framework to explore VR apps and expose privacy risks.

Android Testing. As Meta Quest runs on an Android OS variant, related works in Android UI testing are relevant. Gu et al. [24] abstracted the Android GUI model to generate test cases, and Li et al. [29] introduced a lightweight tool for dynamic, UI-guided test inputs. Su et al. [39] employed functional fuzz testing to uncover logic bugs, while Huang et al. [27] emphasized fuzzing Android apps in the context of data handling. More recently, Liu et al. [30] leveraged Large Language Models (LLMs) for context-aware test input generation, and Ran et al. [36] proposed prioritizing UI events to improve code coverage and detect unique crashes.

State-of-the-art tools like DroidBot [29] and AutoDroid [30] rely on computer vision and tools such as Minicap [34] for state identification, but these methods are incompatible with Android 12, which powers Quest 2 devices. Additionally, they depend on MonkeyRunner for event triggering, which struggles with VR-specific applications. Despite advancements in Android testing, a critical gap remains in developing specialized dynamic tools for 3D engine-based apps and VR games on platforms like Meta Quest.

9 Conclusion

We have presented AUTOVR, a novel framework tailored for dynamically executing and exploring VR apps. By introspecting into the app's internal binary, AUTOVR overcomes the limitations of existing tools, efficiently identifying otherwise inaccessible events. Our empirical evaluation, compared against Monkey, illustrates AUTOVR's significant advancement in event exploration of VR environments by triggering events that lead to sensitive data exposures, thereby enhancing the security and privacy of VR apps.

Acknowledgments

We would like to thank the anonymous reviewers and our shepherd for providing their feedback that improved this paper. Additionally, we would like to give special thanks to Jerry Liang and Edward Liu for providing support for this work. This work was supported by a research grant from *Meta*, in addition to providing a test account and an unlocked Meta Quest 2 device.

Ethics Considerations

We ensured that ethics were considered early in the experimentation process with AutoVR. To collect user data and instrumentation, three locked Meta Quest 2 devices were purchased and used with test accounts. Additionally, a fourth unlocked (i.e., rootable) Quest 2 device was provided by *Meta*, along with a test account that had entitlements to the tested 366 VR apps used in the evaluation (§6). *Meta* has given explicit permission to instrument on VR apps from the Meta Quest store, usage, and instrumentation of AutoVR on the rooted device. Therefore, AutoVR addresses the following concerns: (a) experimented VR apps were entitled to the instrumenting Meta Quest account, and (b) instrumented on devices with owner permission. However, we must note that any misuse of AutoVR may occur if (a) and (b) are violated. Additionally, the data collected from the VR devices and subsequently the test accounts will be stripped of potentially identifiable information (PII) leaked by the developer of each VR app, prior to the release of the network data.

Open Science

The source code of AutoVR is available on GitHub under the MIT license. Additionally, raw data collected from the AutoVR experiments depicted in the evaluation (§6), along with the stripped network data collected by *AntMonitor*. These artifacts can be found at https://doi.org/10.5281/zenodo.15636793.

References

[1] Button | Unity UI | 1.0.0 — 1.0. https://docs.unity3d.com/Packages/com.unity.ugui@1.0/manual/script-Button.html. [Accessed 03-06-2024].

[2] Driving UI Updates with Events in Unreal Engine | Unreal Engine 5.5 Documentation | Epic Developer Community — dev.epicgames.com. https://dev.epicgames.com/documentation/en-us/unreal-engine/driving-ui-updates-with-events-in-unreal-engine. [Accessed 27-05-2025].

[3] frida - a world-class dynamic instrumentation toolkit. https://frida.re/.

[4] Frida interceptor. https://frida.re/docs/javascript-api/#interceptor, journal=Frida, year=2023, month=Mar.

[5] Il2cpp overview. https://docs.unity3d.com/Manual/IL2CPP.html.

[6] Levels in Unreal Engine | Unreal Engine 5.5 Documentation | Epic Developer Community — dev.epicgames.com. https://dev.epicgames.com/documentation/en-us/unreal-engine/levels-in-unreal-engine. [Accessed 27-05-2025].

[7] monkeyrunner | android studio | android developers. https://developer.android.com/studio/test/monkeyrunner.

[8] Mono overview. https://docs.unity3d.com/Manual/Mono.html, journal=Unity, author=Technologies, Unity.

[9] Oculus quest 2 game development options. https://gamefromscratch.com/oculus-quest-2-game-development-options/, journal=GameFromScratch.com, author=Mike, year=2020, month=Oct.

[10] Perfare/il2cppdumper: Unity il2cpp reverse engineer. https://github.com/Perfare/Il2CppDumper, journal=GitHub, author=Perfare.

[11] Unity launches beta program for visionos — enabling unity developers to create games and apps for apple vision pro | business wire. https://www.businesswire.com/news/home/20230719202814/en/Unity-Launches-Beta-Program-for-visionOS-%E2%80%94-Enabling-Unity-Developers-to-Create-Games-and-Apps-for-Apple-Vision-Pro. (Accessed on 07/31/2023).

[12] Unity real-time development platform. https://unity.com/.

[13] Virtual reality (vr) market size, share, growth & trends.

[14] Il2cpp reverse engineering part 1: Hello world and the il2cpp toolchain. https://katyscode.wordpress.com/2020/06/24/il2cpp-part-1/, Dec 2020.

[15] UI/Application Exerciser Monkey. https://developer.android.com/studio/test/other-testing-tools/monkey, Jan 2022. [Online; accessed 1. Mar. 2023].

[16] Gadget. https://frida.re/docs/gadget/, Mar 2023.

[17] The interaction components in the UI system handle interaction, such as mouse or touch events and interaction using a keyboard or controller. https://docs.unity3d.com/Packages/com.unity.ugui@1.0/manual/comp-UIInteraction.html, Jun 2023. [Online; accessed 15. Jul. 2023].

[18] Devon Adams, Alseny Bah, Catherine Barwulor, Nureli Musaby, Kadeem Pitkin, and Elissa M Redmiles. Ethics emerging: the story of privacy and security perceptions in virtual reality. In *Fourteenth Symposium on Usable Privacy and Security (SOUPS 2018)*, pages 427–442, 2018.

[19] Benjamin Andow, Samin Yaseer Mahmud, Justin Whitaker, William Enck, Bradley Reaves, Kapil Singh, and Serge Egelman. Actions speak louder than words: Entity-Sensitive privacy policy and data flow analysis with PoliCheck. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 985–1002. USENIX Association, August 2020.

[20] Mykhailo Antonishyn. Four ways to bypass android SSL. verification and certificate pinning. *Transfer of Innovative Technologies*, 3(1):96–99, September 2020.

[21] Enrico Bacis, Igor Bilogrevic, Robert Busa-Fekete, Asanka Herath, Antonio Sartori, and Umar Syed. Assessing web fingerprinting risk. In *Companion Proceedings of the ACM on Web Conference 2024*, pages 245–254, 2024.

[22] BillWagner. Attributes and reflection. https://learn.microsoft.com/en-us/dotnet/csharp/advanced-topics/reflection-and-attributes/.

[23] Ceenu George, Mohamed Khamis, Emanuel von Zezschwitz, Marinus Burger, Henri Schmidt, Florian Alt, and Heinrich Hussmann. Seamless and secure vr: Adapting and evaluating established authentication systems for virtual reality. NDSS, 2017.

[24] Tianxiao Gu, Chengnian Sun, Xiaoxing Ma, Chun Cao, Chang Xu, Yuan Yao, Qirun Zhang, Jian Lu, and Zhendong Su. Practical gui testing of android applications via model abstraction and refinement. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 269–280. IEEE, 2019.

[25] Hanyang Guo, Hong-Ning Dai, Xiapu Luo, Gengyang Xu, Fengliang He, and Zibin Zheng. An empirical study on meta virtual reality applications: Security and privacy perspectives. *IEEE Transactions on Software Engineering*, 2025.

[26] Ralph Hauwert. The future of scripting in unity, May 2014.

[27] Xinyue Huang, Anmin Zhou, Peng Jia, Luping Liu, and Liang Liu. Fuzzing the android applications with http/https network data. *IEEE Access*, 7:59951–59962, 2019.

[28] Anh Le, Janus Varmarken, Simon Langhoff, Anastasia Shuba, Minas Gjoka, and Athina Markopoulou. Antmonitor: A system for monitoring from mobile devices. In *Proceedings of the 2015 ACM SIGCOMM Workshop on Crowdsourcing and Crowdsharing of Big (Internet) Data*, pages 15–20, 2015.

[29] Yuanchun Li, Ziyue Yang, Yao Guo, and Xiangqun Chen. Droidbot: a lightweight ui-guided test input generator for android. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, pages 23–26, 2017.

[30] Zhe Liu, Chunyang Chen, Junjie Wang, Xing Che, Yuekai Huang, Jun Hu, and Qing Wang. Fill in the blank: Context-aware automated text input generation for mobile gui testing. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 1355–1367. IEEE, 2023.

[31] Jianwen Lou, Yiming Wang, Charles Nduka, Mahyar Hamedi, Ifigeneia Mavridou, Fei-Yue Wang, and Hui Yu. Realistic facial expression reconstruction for vr hmd users. *IEEE Transactions on Multimedia*, 22(3):730–743, 2019.

[32] Meta. Oculus quest store: Vr games, apps, amp; more. https://www.oculus.com/experiences/quest/.

[33] Meaghan Moody. Oculus quest 2 specifications. https://studiox.lib.rochester.edu/oculus-quest-2-specifications/, Sep 2021.

[34] OpenSTF. Openstf/minicap: Stream real-time screen capture data out of android devices.

[35] Fiachra O'Brolcháin, Tim Jacquemard, David Monaghan, Noel O'Connor, Peter Novitzky, and Bert Gordijn. The convergence of virtual reality and social networks: threats to privacy and autonomy. *Science and engineering ethics*, 22:1–29, 2016.

[36] Dezhi Ran, Hao Wang, Wenyu Wang, and Tao Xie. Badge: Prioritizing ui events with hierarchical multi-armed bandits for automated ui testing. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 894–905. IEEE, 2023.

[37] Dhia Elhaq Rzig, Nafees Iqbal, Isabella Attisano, Xue Qin, and Foyzul Hassan. Virtual reality (vr) automated testing in the wild: A case study on unity-based vr applications. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 1269–1281, 2023.

[38] Sensepost. objection - runtime mobile exploration. https://github.com/sensepost/objection.

[39] Ting Su, Yichen Yan, Jue Wang, Jingling Sun, Yiheng Xiong, Geguang Pu, Ke Wang, and Zhendong Su. Fully automated functional fuzzing of android apps for detecting non-crashing logic bugs. *Proceedings of the ACM on Programming Languages*, 5(OOPSLA):1–31, 2021.

[40] Unity Technologies. Unity - Manual: Event Functions — docs.unity3d.com. https://docs.unity3d.com/Manual/EventFunctions.html. [Accessed 03-06-2024].

[41] Unity Technologies. Unity - Manual: GameObject — docs.unity3d.com. https://docs.unity3d.com/Manual/class-GameObject.html. [Accessed 03-06-2024].

[42] Unity Technologies. Unity - Manual: Interaction between collider types — docs.unity3d.com. https://docs.unity3d.com/Manual/collider-types-interaction.html. [Accessed 02-06-2024].

[43] Unity Technologies. Unity - Manual: Introduction to collider types — docs.unity3d.com. https://docs.unity3d.com/Manual/collider-types-introduction.html. [Accessed 03-06-2024].

[44] Unity Technologies. Unity - Manual: Order of execution for event functions — docs.unity3d.com. https://docs.unity3d.com/2021.3/Documentation/Manual/ExecutionOrder.html. [Accessed 27-05-2025].

[45] Unity Technologies. Unity - Scripting API: Button — docs.unity3d.com. https://docs.unity3d.com/2018.2/Documentation/ScriptReference/UI.Button.html. [Accessed 03-06-2024].

[46] Unity Technologies. Unity - Scripting API: MonoBehaviour.FixedUpdate() — docs.unity3d.com. https://docs.unity3d.com/2022.3/Documentation/ScriptReference/MonoBehaviour.FixedUpdate.html. [Accessed 27-05-2025].

[47] Unity Technologies. Unity - Scripting API: IEventSystemHandler — docs.unity3d.com. https://docs.unity3d.com/2019.1/Documentation/ScriptReference/EventSystems.IEventSystemHandler.html, August 2019.

[48] Rahmadi Trimananda, Hieu Le, Hao Cui, Janice Tran Ho, Anastasia Shuba, and Athina Markopoulou. OVRseen: Auditing network traffic and privacy policies in oculus VR. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 3789–3806, Boston, MA, 2022.

[49] Unity. Collider. https://docs.unity3d.com/ScriptReference/Collider.html.

[50] Unity. Component. https://docs.unity3d.com/ScriptReference/Component.html.

[51] Unity. Gameobject. https://docs.unity3d.com/ScriptReference/GameObject.html.

[52] Unity. Managed code stripping. https://docs.unity3d.com/Manual/ManagedCodeStripping.html.

[53] Unity. Scenemanager. https://docs.unity3d.com/ScriptReference/SceneManagement.SceneManager.html.

[54] John Vilk, David Molnar, Benjamin Livshits, Eyal Ofek, Chris Rossbach, Alexander Moshchuk, Helen J Wang, and Ran Gal. Surroundweb: Mitigating privacy concerns in a 3d web browser. In *2015 IEEE Symposium on Security and Privacy*, pages 431–446. IEEE, 2015.

[55] Hao Wen, Yuanchun Li, Guohong Liu, Shanhui Zhao, Tao Yu, Toby Jia-Jun Li, Shiqi Jiang, Yunhao Liu, Yaqin Zhang, and Yunxin Liu. Autodroid: Llm-powered task automation in android, 2024.

[56] Z3Prover. Z3prover/z3: The z3 theorem prover. https://github.com/Z3Prover/z3.

[57] Chaoshun Zuo and Zhiqiang Lin. Playing without paying: Detecting vulnerable payment verification in native binaries of unity mobile games. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 3093–3110, Boston, MA, August 2022. USENIX Association.

	Consistency Result		
Data Flow Types	*Ambiguous*	*Omitted*	*Vague*
Android ID	0	0	1
App Name	0	13	0
Build Version	0	12	0
Cookie	0	1	0
Device ID	1	5	6
Flags	0	12	0
Hardware Information	0	50	1
Language	0	3	5
Sdk Version	0	22	1
Session Information	0	14	0
System Version	0	10	0
Usage Time	0	11	0
User ID	1	5	8
VR Field of View	0	1	0
VR Movement	0	8	0
VR Play Area	0	2	0
VR Pupillary Distance	0	1	0
Total	2	170	22

Table 5: *PoliCheck* [19] results for the total data flow type for each consistency result using OVRSeen's ontology. The set contains 44 VR apps, with privacy policies, where AUTOVR was able to trigger outgoing network data flows.

```
ob.opencampus.mobile#0.5P???
??email=********%40protonmail.com&password=******
%21ob.opencampus.mobile#0.5?P??
?POST /api/login HTTP/1.1
Host: gcsvrapi.worldbank.org
Accept-Encoding: gzip, identity
Connection: Keep-Alive, TE
TE: identity
User-Agent: BestHTTP 1.12.1
Content-Type: application/x-www-form-urlencoded
Content-Length: 54

ob.opencampus.mobile#0.5P????ebbEb??????P??IB
GET /api/featured?unity=true&wm=false HTTP/1.1
Authorization: Bearer eyJ0eXAiOiJKV1QiLCJhImp0a<truncated>
```

Figure 13: Outgoing network traffic from `ob.opencampus.mobile` intercepted while performing AUTOVR. The plaintext email and password for this network request is redacted, and the authorization token is truncated.

A Case Study

While perusing the collected data exposures for AUTOVR's traffic, we noticed incoming and outgoing network requests between the VR device and external servers. Notably, the one app that exposes `EMAIL` was identified traffic coming from `ob.opencampus.mobile`, which contained alarming HTTP packets that include the plaintext string of an individual's email and password. Specifically, as shown in Figure 13, we notice that the endpoint to the incoming and outgoing traffic is linked to a `gcsvrapi.worldbank.org` domain. However, when attempting to invoke the same exposure behavior with Monkey, none of this traffic was identified. This is also supported by Figure 11. This highlights the significance of AUTOVR's ability to uncover events at the binary level, invoking events accurately and triggering potentially damaging privacy exposures.

B Inclusion of Unreal Engine

AUTOVR 's internals largely depend on the structure of Unity IL2CPP games, however, the technique of scene loading, event extraction, and execution applies to engines such as Unreal. Unity Scenes are synonymous with Unreal Levels [6]. UI events are similarly connected using function hooks [2]. Using scenes/levels to reset the state, identifying UI-events within GameObjects/Blueprints, and extracting the function callbacks are synonymous to both engines. To cover more games within the app stores, Unreal binaries will be considered for future work beyond AUTOVR.

C Preliminary crash detection

Because AUTOVR is agnostic of the privacy/security application, AUTOVR can be used for additional applications beyond sensitive data exposure, such as crash detection. As such, we separately collected the number of crashes invoked by AUTOVR and compared it with Monkey as shown in Table 7. We notice that a significant number of crashes are `SEGV_-MAPERR`, which could either be mapping issues between *Frida*

```
Host: cdp.cloud.unity3d.com
User-Agent: UnityPlayer/2019.2.19f1
Accept-Encoding: deflate, gzip
Accept: */*
Content-Type: application/json
event_count: 1
data_block_id: 64351db8b4d511c2e16a1d97a1907e7b
expired_session_dropped: 0
data_retry_count: 5
continuous_request: 1
request_ts: 1713146923986
X-Unity-Version: 2019.2.19f1
Content-Length: 496

com.aura.Arrows#0.5.5X?????"k??????P??-|{"common":
{"appid":"local.1457b9e91549e2a40a8d759ee2972f52",
"userid":"f213a7998a70e2340aa2f60d6137682c",
"sessionid":5687031950375056550,"platform":"AndroidPlayer",
"platformid":11,"sdk_ver":"u2019.2.19f1","session_count":3,
"localprojectid":"1457b9e91549e2a40a8d759ee2972f52",
"build_guid":"ff76281b9a0c28c46914e0c261de62e1",
"device_id":"b8ba3da0d4458fde63a51772ce84547b"}}
{"type":"analytics.appStart.v1","msg":
{"previous_sessionid":1848577819610040954,
"ts":1713146819046,"t_since_start":11062992}}
com.aura.Arrows#0.5.5Xp??..E.OM:??"oq(?Z?P??yPOST / HTTP/1.1
```

Figure 14: Outgoing network traffic from `com.aura.Arrows` intercepted while performing AUTOVR.

API Name	Return Type	Argument Types	Description
Classes			
il2cpp_class_get_field_from_name	field pointer	class pointer, string	Get class field handle from name.
il2cpp_class_get_fields	fields array pointer	class pointer, class type	Get field handles from class.
il2cpp_class_get_method_from_name	method pointer	class pointer, string, int	Get class method handle from method name and parameter count.
il2cpp_class_get_methods	methods array pointer	class pointer, pointer, class type	Get all method handles from class.
il2cpp_class_get_name	string	class pointer	Get name from class handle.
il2cpp_class_get_namespace	string	class pointer	Get class namespace name from class handle.
il2cpp_class_get_parent	parent class pointer	class pointer	Get class' parent handle.
il2cpp_class_get_type	class type pointer	class pointer	Get class type from class handle.
il2cpp_class_is_assignable_from	bool	class pointer, other class pointer	Test if class is assignable from another class.
Fields			
il2cpp_field_get_parent	parent class pointer	class pointer	Get parent class of field handle.
il2cpp_field_get_name	field pointer	string	Get field name from field handle.
il2cpp_field_get_offset	int32	field pointer	Get class field offset from field handle.
il2cpp_field_static_get_value	void	field pointer, value pointer	Get field handle's static value into value pointer.
il2cpp_field_get_type	class type pointer	field pointer	Get type of field from field handle.
il2cpp_field_is_static	bool	field pointer	Test if field is static from field handle.
Methods			
il2cpp_method_get_class	class pointer	method pointer	Get residing class handle from method handle.
il2cpp_method_get_name	string	method pointer	Get method name from method handle.
il2cpp_method_get_object	object pointer	method pointer	Get object handle from method handle.
il2cpp_method_get_param_count	uint8	method pointer	Get number of parameters from method.
il2cpp_method_get_parameter_name	parameters pointer	string, uint32	Get parameter handle from name and parameter index.
il2cpp_method_get_pointer	virtual address pointer	method pointer	Get virtual address from method handle.
il2cpp_method_get_return_type	class type pointer	method pointer	Get return type of method.
Objects			
il2cpp_object_get_class	object pointer	class pointer	Get class handle from object.
il2cpp_object_new	object pointer	class pointer	Create a new object from class handle.
il2cpp_capture_memory_snapshot	objects array pointer	void	Creates a memory snapshot of all GC handles.

Table 6: IL2CPP Runtime Library API

and IL2CPP, or critical software crashes. Interestingly, we notice that Monkey was also able to trigger `SEGV_MAPERR` crashes, which is unrelated to *Frida* and IL2CPP's mapping, potentially indicating software issues with the game/app itself. We acknowledge that crash detection can be a separate research direction for AutoVR, exemplifying AutoVR's significance in the VR ecosystem.

	Free Apps		Paid Apps	
	SEGV Code			
Tool	MAPERR	ACCERR	MAPERR	ACCERR
Monkey	3	0	0	0
AutoVR	69	2	5	0

Table 7: Aggregated crash count with signal (SIGSEGV) between Monkey and AutoVR for both free and paid games.