[Usenix Security 25] AUTOVR: Automated UI Exploration for Detecting Sensitive Data Flow Exposures in Virtual Reality Apps

John Y. Kim, Chaoshun Zuo, Yanjie Zhao, and Zhiqiang Lin,

Ohio State University

Reporter: Zhengyang Zhu

November, 4, 2025

中山大學
SUN YAT-SEN UNIVERSITY

Introduction

快速省流总结

- 为了解决VR App 黑盒自动化UI探索和用户事件测试问题
- 通过拦截敏感数据流作为评估指标对比SOTA
- 目前AUTOVR**仅局限于拦截**敏感数据流
- 未来**可结合**识别数据流方法，判断App是否泄露用户隐私、敏感信息

Introduction

快速省流总结

- 为**获取元数据**，AUTOVR 首先通过对打包后的VR Apps 进行逆向 (UI语义恢复、函数回调地址提取)；
- 为更全面覆盖App的UI和用户事件，AUTOVR结合元数据 + Frida + Bypassing 机制，进行事件执行和回溯；
- 能够绕过控制器输入层，从更高层对场景中UI/物理事件进行全面执行、覆盖

Introduction

Motivation

- 虚拟现实市场规模持续增长（预计从 2025 年的 324 亿美元增长至 2030 年的 1874 亿美元）
- VR 应用的质量保证至关重要（涉及隐私安全、敏感数据保护以及可靠的用户体验）
- Unity 的动态分析工具仍然空缺（注：Unity 是最受欢迎的 VR 3D 引擎，例如，是 Apple Vision Pro 唯一官方支持的引擎）

Introduction

Challenges

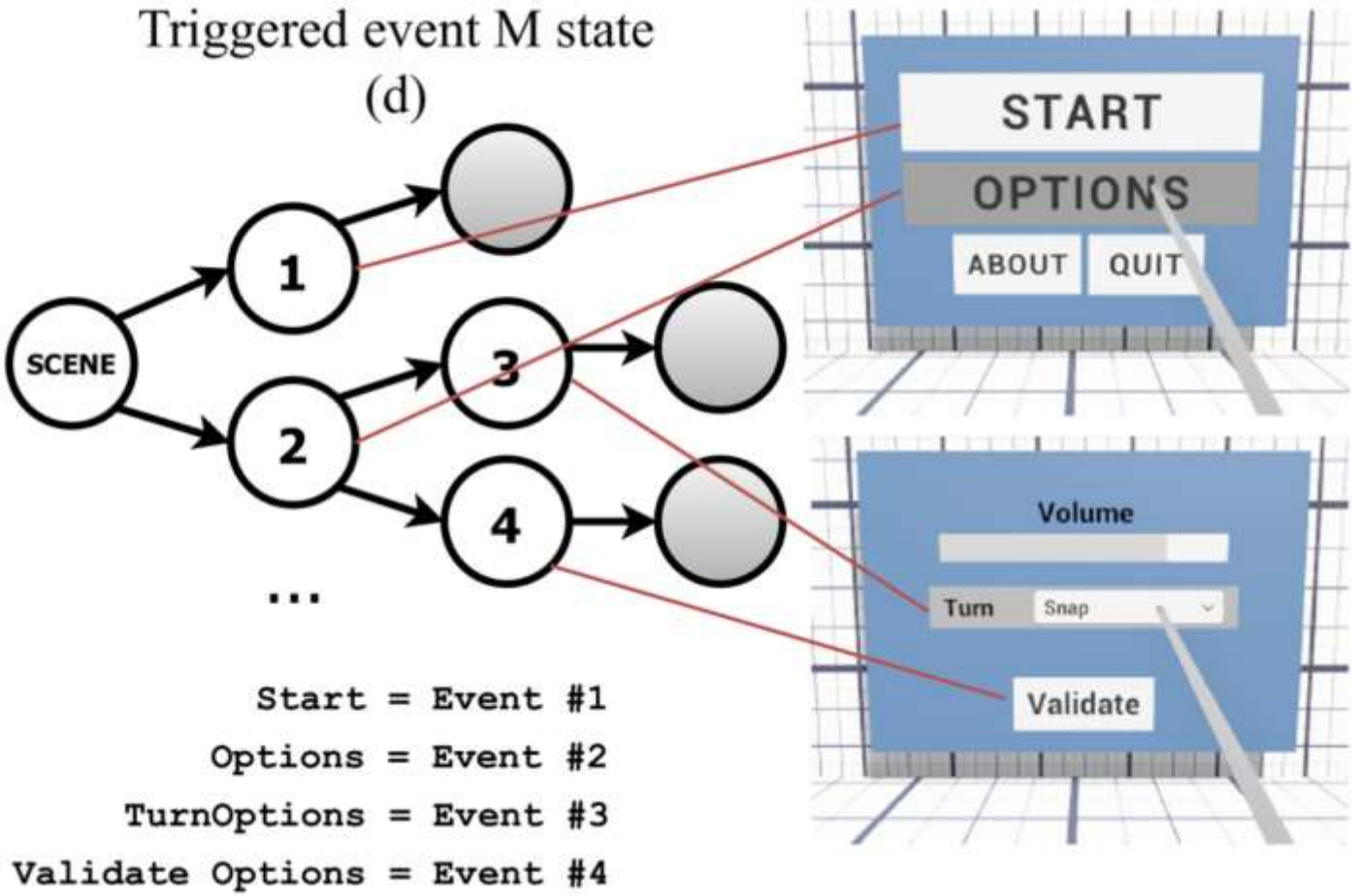
- Unity **Embedded UI 结构** (UI嵌入到打包后的二进制APK中，相比于安卓的.xml难以解析)
- **动态生成式 UI 物体** (UI物体的覆盖检测不像传统的树遍历问题那样，可以通过回溯解决；Unity的UI物体存在出现 disabled状态的可能，意味着无法直接回到上一级，即无法通过简单的事件回溯解决)

Introduction

Challenges

- **动态生成式 UI 物体**

  Unity的UI物体存在出现 disabled状态的可能，意味着无法直接回到上一级，即无法通过简单的事件回溯解决)



Introduction

Challenges

- 动态生成式 UI 物体

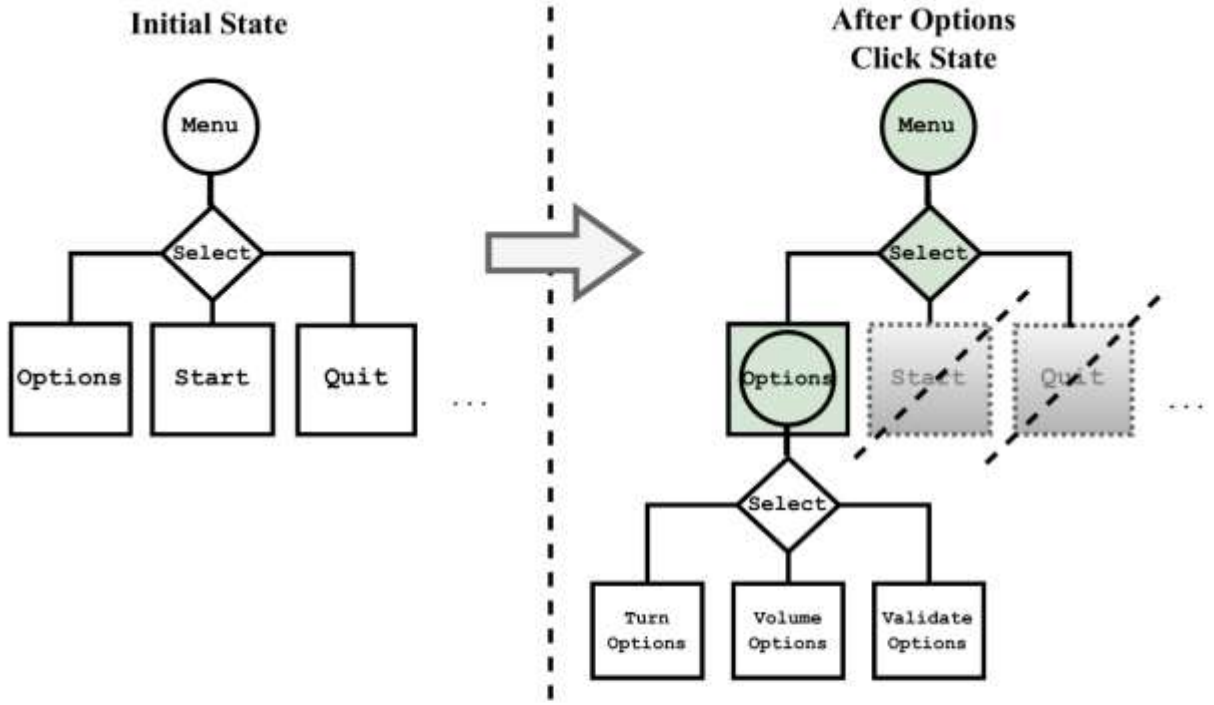


Figure 5: High level event state before and after `Options$$OnOptionsClick` was clicked in Figure 2. Highlighted in green is the path taken, and highlighted gray with dotted outlines are disabled events.

Overview of Proposed Approach

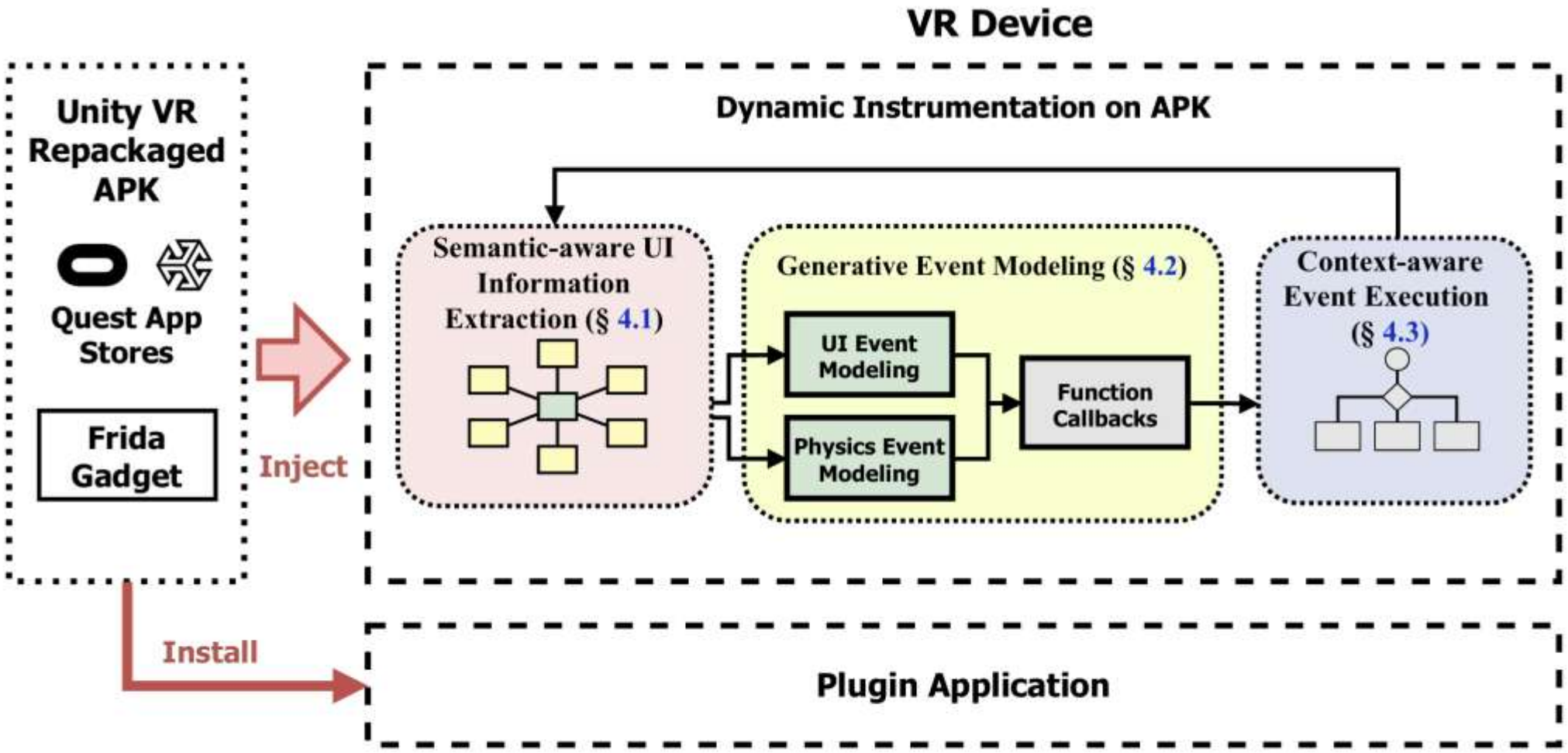


Figure 6: Overview of AUTOVR

Overview of Proposed Approach

UI语义恢复 (Recover UI Semantics)

- 由于Unity **Embedded UI 结构**，UI嵌入到打包后的二进制APK中，相比于安卓可以直接通过AndroidManifest.xml要更难获取UI语义。
- Unity 支持两种脚本后端（Scripting Backend）：
  - 一是 **Mono**，采用即时编译（JIT，Just-In-Time）机制；
  - 二是 **IL2CPP**，采用预编译/提前编译（AOT，Ahead-Of-Time）机制。IL2CPP 会将 C# 源码/中间语言转换为 C++ 再编译成本机二进制，可带来更高的运行效率与更好的平台兼容性，因此在发布构建中使用较为广泛。

Overview of Proposed Approach

UI语义恢复 (Recover UI Semantics)

- 为了恢复UI语义，作者提出一种**动静态分析**结合的方法：
- 动态：从二进制库 libil2cpp.so 调用 IL2CPP APIs函数的方法来获得元数据(metadata)，
- 静态：从 global-metadata.dat 中还原反射信息。
- 在 Unity **游戏逆向**里，global-metadata.dat 常与 libil2cpp.so（Android）或 GameAssembly.dll（Windows）配合，来还原原始的 C# 结构。global-metadata.dat 中以**加密**的形式存储了需要反射的符号信息。

Overview of Proposed Approach

生成式UI 建模 (Generative UI Modeling)

- UI Events指用户通过交互操作，如点击按钮，触发的事件。
- 背景：Unity 提供了两种事件调用绑定方式（混合绑定）：
- **动态绑定**：

  通过 AddListener() 方法添加函数回调，这种方式只存在**内存中，不会可视化在Editor。**

- **静态绑定**：

  通过序列化、可视化的方法将函数回调“绑定”到Inspector面板

Overview of Proposed Approach

生成式UI 建模 (Generative UI Modeling)

- 由于**绑定的复杂性**，因此在应用运行时动态地收集 UI元素、以及UI绑定的事件调用（即 UI Events）是很困难的，作者提出：
- 通过 IL2CPP **类内省** (Class Introspection) API，在**场景加载开始**时，枚举场景中所有 GameObject 与其挂载的组件，识别、收集实现 IEventSystemHandler 接口的组件（脚本）；
- 比如 Button、Toggle、IPointerClickHandler 等，在场景树里找这些类型的实例，快速收集“哪些对象可能产生 UI 事件”的候选集（按钮、可交互控件等），为后续更细粒度的回调收集做准备。

Overview of Proposed Approach

- **生成式UI 建模 (Generative UI Modeling)**
  - 通过 IL2CPP **字段内省**（Field Introspection） API，在场景**运行时**，动态 hook 收集开发者的事件回调；
  - 使用 IL2CPP 的字段反射接口读取组件实例内部的 UnityEvent 字段，解析其内部的 PersistentCalls（静态序列化调用）和 m_Calls（运行时添加的 listeners）。
  - 即对 UI Events，做 Hook，在触发时把回调目标、目标对象、函数名、参数等信息记录下来。

Overview of Proposed Approach

生成式UI 建模 (Generative UI Modeling)

- 总结：这一步的目的主要是更全面的收集**下游任务**可能需要的UI物体。
- 改进：结合静态和动态绑定，覆盖所有 UI 回调，不再漏掉动态注册或序列化绑定，**更完整的进行生成式UI建模**

Overview of Proposed Approach

生成式物理事件建模 (Generative Physics Modeling)

- 背景——Unity提供两种物理交互组件：
  - collision（碰撞器）：需要包含 collider（碰撞体）和刚体
  - trigger（触发器）：需要将碰撞体的 triggered属性设置为True
- 为了提取、建模物理事件，作者提出：
  - 从**生成式UI 建模（上一步）**提取的信息中，**定位**需要进行物理交互模拟的**物体**、并匹配物理交互规则，将它们**“位移”**到需要和其他物体交互的位置

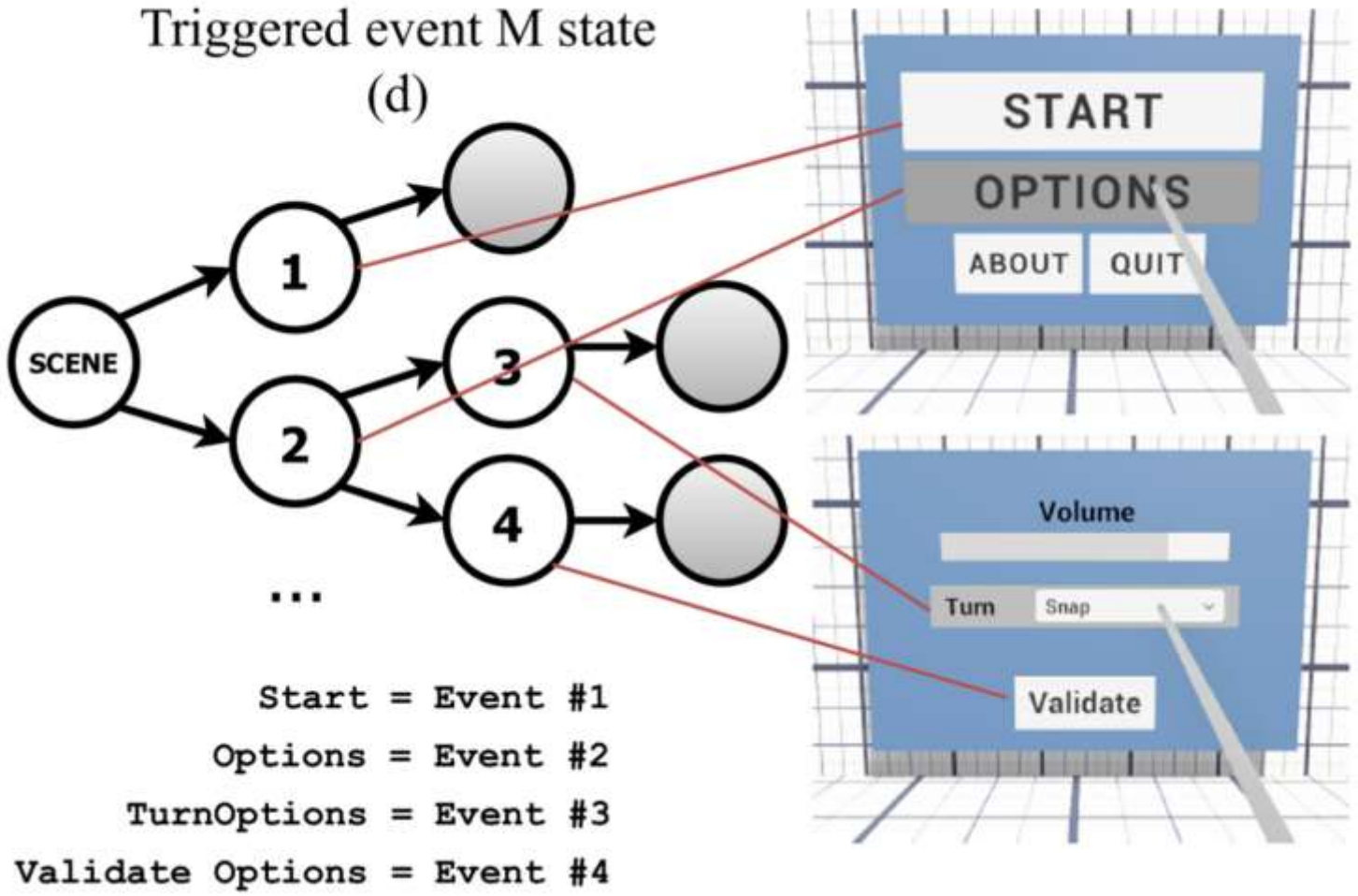
Overview of Proposed Approach

生成式事件执行 (Execute Generative Events)

- 由于Unity存在 UI的动态生成，如果把UI Events进行建模，结构类似于一棵树，然而不同于传统的树遍历问题，由于Unity存在物体销毁的情况，状态回溯不一定可行，完整的 UI事件覆盖十分困难。
- 作者提出，通过Unity的 LoadSceneAsyncNameIndexInter函数，实现**场景的重新加载**，从而实现一种新的意义上的回溯。这样可以完整的覆盖不同的事件路径。

Overview of Proposed Approach

- 生成式事件執行 (Execute Generative Events)




```
Start = Event #1
Options = Event #2
TurnOptions = Event #3
Validate Options = Event #4
```

Overview of Proposed Approach

总结

- UI语义恢复解决了黑盒测试下的数据逆向问题；
- 生成式UI建模，解决了Unity复杂的UI事件绑定背景下，UI Event节点的完整 hook和建模问题；
- 生成式物理事件建模，在UI建模后提供的 GameObjects候选List的基础上，进行物理Event节点的提取和建模；
- 生成式事件执行，通过场景回溯机制，能够完整的探索 前两步提取的Event节点构成的模型。

Technical Details

UI 语义恢复

形象化说明：

`global-metadata.dat` 相当于图书馆的总目录 (项目静态元数据)，把所有书名/作者信息 (函数/类方法签名)印成一本大目录；`libil2cpp.so` 相当于图书馆管理和检索平台 (**运行时**将元数据提取加载到内存，构造一组运行时数据结构)；内存里的指针就像书架上的标签，管理员凭标签找到对应位置并打开书来读取内容（对指针解引用）；而 Frida 则相当于一个外部操作者，它能向管理系统发出指令（调用 `il2cpp_*` 系列 API），从而读取或操作书架上的具体书籍（类、方法、字段等）。

Technical Details

UI 语义恢复

1. 提取类元数据 (Class Metadata)

`libil2cpp.so` 把 `global-metadata.dat` 的静态元数据在运行时组织成可访问的结构，并在运行时加载到内存；IL2CPP 的内省 API 暴露这些结构供外部调用。通过 Frida 等注入工具调用以 `il2cpp_class` 开头的一组类内省API，可以在内存中读取类名、父类、字段、属性、方法列表（`MethodInfo*` 的句柄），无法直接看到方法参数、返回值、函数指针等详细签名信息

Technical Details

UI 语义恢复

2. 提取方法元数据

`libil2cpp.so` 把 `global-metadata.dat` 的静态元数据在运行时组织成可访问的结构，并在运行时加载到内存；调用以 `il2cpp_method` 为前缀的一组内省 API 提取函数信息，包括方法名、参数类型、参数数量、返回类型、方法指针、所属类，并在启动阶段构建全局函数表（Global Function Table，GFT）：以函数入口地址（offset）为键、函数引用句柄为值。该 GFT 可用于后续的事件函数（EFC）识别与分析。

Technical Details

UI 语义恢复

例如，

先通过类内省找到方法的入口句柄

```
il2cpp_class_get_methods(Il2CppClass* klass, void**
iter);
```

得到每个方法的 `MethodInfo*` 。

再通过方法内省读取签名信息

```
const char* name = il2cpp_method_get_name(method);
const Il2CppType* ret =
il2cpp_method_get_return_type(method);
const Il2CppType* param = il2cpp_method_get_param(method,
i);
```

Technical Details

UI 语义恢复

3. 提取物体

通过IL2CPP运行时使用内存快照(memory snapshot)来识别垃圾回收句柄的方式，指向物体。

> 1. 读取内存快照得到一份静态的"堆视图"。基于快照可以枚举当前存活的托管对象及其引用关系。
> 2. 先获取或枚举 GC 句柄表里的每个句柄 → 把句柄解引用得到托管对象指针（`Il2CppObject*`）→ 从对象指针解析出该对象的类/字段（比如是否为 `GameObject`、组件、UI 元素），从而"找到物体"

Technical Details

UI 语义恢复

4. 恢复UI元素

由于Unity中将UI元素和其他物体"一视同仁"，UI元素上会附加UI组件(脚本实例化)。Unity官方SDK的UI组件容易识别，然后开发者可能会定制自己的UI组件 (不派生自官方SDK)。作者发现，所有的UI组件都会实现 `IEventSystemHandler` 接口，因此通过上步骤提取到的元数据，可以搜索相应的UI元素。

Technical Details

生成式事件建模

识别UI事件

从 `UnityEventBase` 提供的两个字段 `m_calls` （运行时回调和序列持久化回调转换）和 `m_PersistenCalls` 中（序列化回调）解析、提取回调函数元数据。

内部成员	存储内容	生命周期	添加方式
InvokableCallList m_Calls	- **Runtime callbacks**（通过 AddListener() 动态注册）- **部分持久化回调**（加载时从 PersistentCallGroup 转换而来）	仅在程序运行期间有效	运行时调用 AddListener() 或加载时自动添加持久化回调
PersistentCallGroup m_PersistentCalls	开发者在 Inspector 中分配的 **持久化回调（Persistent callbacks）**	存储于场景/Prefab 文件中，随项目保存	在编辑器 Inspector 中配置

Technical Details

生成式事件建模

识别物理事件

	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.

作者提取出 Trigger Functions（`OnTriggerEnter, OnTriggerStay, and OnTriggerExit`）和 Collider Functions（`OnCollisionEnter, OnCollisionStay, and OnCollisionExit`）的虚拟地址，为下游分析做准备。

Technical Details

■ 生成式事件建模

事件执行 (To be more holistic)

1. 调用 `Frida` 工具（结合通过GFT (Global Function Table)），提取Function Offset 和参数
2. 初始EFC (Event Function Callbacks) 随机选择
3. 将新生成的EFC 作为 Dependency，并通过树结构建模
4. **依赖回溯机制**：如果某个Child EFC 失效(Disabled，比如挂载物体被销毁)，
   a. 若其Parent EFC 是当前场景的**根事件**，或者出现**不可逆状态**，则进行**场景重载**(Scene Reloading);
   b. 若其Parent EFC 是非根事件且可逆，则进行**父事件重放**(Parent EFC Replaying);

Technical Details

生成式事件建模

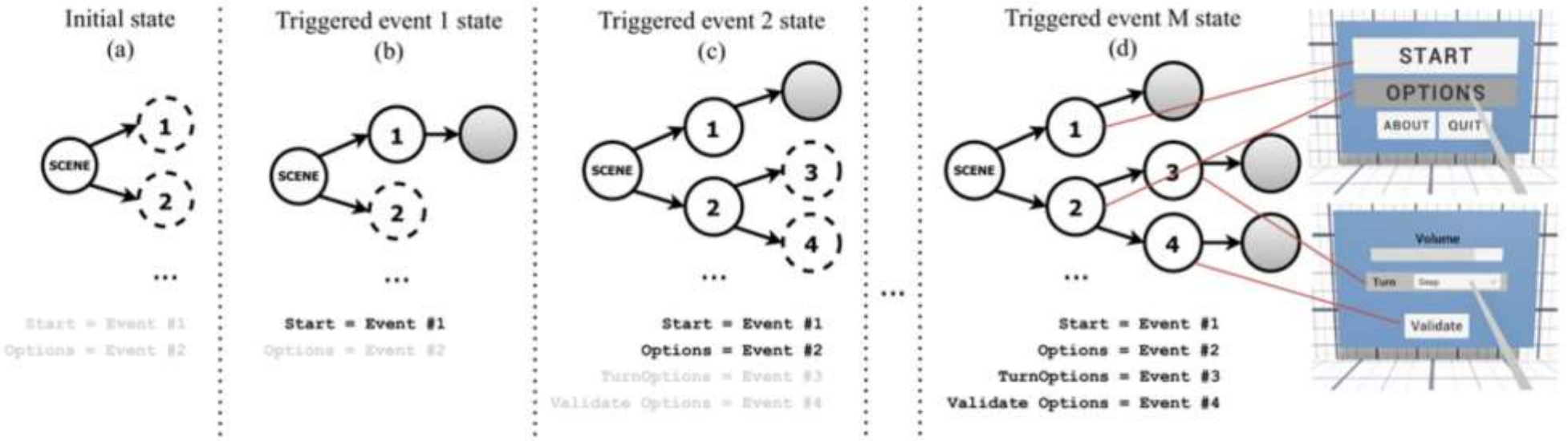


Figure 4: Generative UI-driven model of the running example. This is analogous to Scene 0 in Figure 3, where the modeling process per scene is repeated for every scene in a VR game. The lighter colored text indicates the UI event has been found but not yet triggered, whereas the darkened colored text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

Implementation & Appilication

敏感数据监测 (Sensitive Data Detection)

- AUTOVR: application-agnostic
- 采用 *AntMonitor* (基于VPN服务, 在安卓设备运行, 在设备上建立一个用户空间的 VPN 接口，从而截取设备所有进出网络的数据包，而 无需设备 root 权限) 拦截VR Apps SSL/TLS Traffic
- 然而，在第三方 Unity 应用中，SSL/TLS Pinning 会加密所有网络流量，使得AntMonitor等无法解密通信，使得研究者无法通过常规方式分析通信内容或检测其中可能存在的数据泄露与安全隐患。

Implementation & Appilication

敏感数据监测 (Sensitive Data Detection)

- **TLS (Transport Layer Security)** 是互联网通信中用于**加密数据传输**的协议。
- 内容是加密的，无法直接看到 HTTP 请求头、正文等；只能观察到“元数据”（metadata）
- **SSL（Secure Sockets Layer）** 是一种用于在网络上传输加密数据的安全协议。 后来它被 TLS 取代，但很多人仍习惯统称为“SSL”。
- **SSL/TLS Pining：** 应用在建立 HTTPS/TLS 连接时，不仅验证服务器证书是否有效，还会“固定（pin）”一个特定证书或公钥，只有匹配的服务器才能通过验证。
- 普通 HTTPS 连接只要证书由受信任的 CA(Certificate Authority（证书颁发机构) 签发就可以通过验证。

Implementation & Appilication

敏感数据监测 (Sensitive Data Detection)

- 问题是：
  - 攻击者可以在客户端和服务器之间插入伪造证书（即 中间人攻击，MITM）
  - 如果客户端只信任 CA 而不检查具体证书，攻击者就可以解密流量。
- **Pinning 的作用就是：**
  - 指定只能和某个 **固定证书/公钥** 的服务器通信；
  - 任何证书替换都会导致连接失败，从而阻止 MITM 攻击。

Implementation & Appilication

解决方法：绕过Pining (Bypassing)

Android 层常见绕过思路

- **Java 层拦截**：许多 Android 应用使用 Java/Android API（如 OkHttp、HttpsURLConnection、X509TrustManager）做 TLS 验证。可以在运行时通过 API hooking（如 Frida）拦截这些高层接口，并让验证函数返回“可信”。
- **原生库拦截**：某些库在 native 层（例如通过 openssl 或 BoringSSL）做验证，则需要拦截相应的 native 函数。

Implementation & Appilication

解决方法：绕过Pining (Bypassing)

Unity层

- Unity 应用可能既有 C#（托管层）网络栈（UnityWebRequest、WWW），也可能用 **native** 的 TLS 库（例如 Unity 自带或第三方的 mbedTLS）。而且 Unity 的运行时（Mono 或 IL2CPP）会把 C# 逻辑编译或转为 native，导致 hook 点不一致。

Implementation & Appilication

解决方法：绕过Pining (Bypassing)

- **Unity层解决方法**：
  - 在 native 层用动态分析/注入工具（如 Frida 的 native hooking 能力）定位到这些函数或它们的返回检查点；
  - 在运行时**修改函数的返回值或相关 flag**，使证书验证路径被视为“成功”；或修改 callback 行为使其跳过严格检查。
  - 对于托管层（C#）的证书检查，可在 Mono/IL2CPP 层拦截相应的证书验证函数（或 Unity 的 CertificateHandler），让其接受证书或暴露明文数据。

Experiment

实验对象及规模

- One Custom VR Unity App

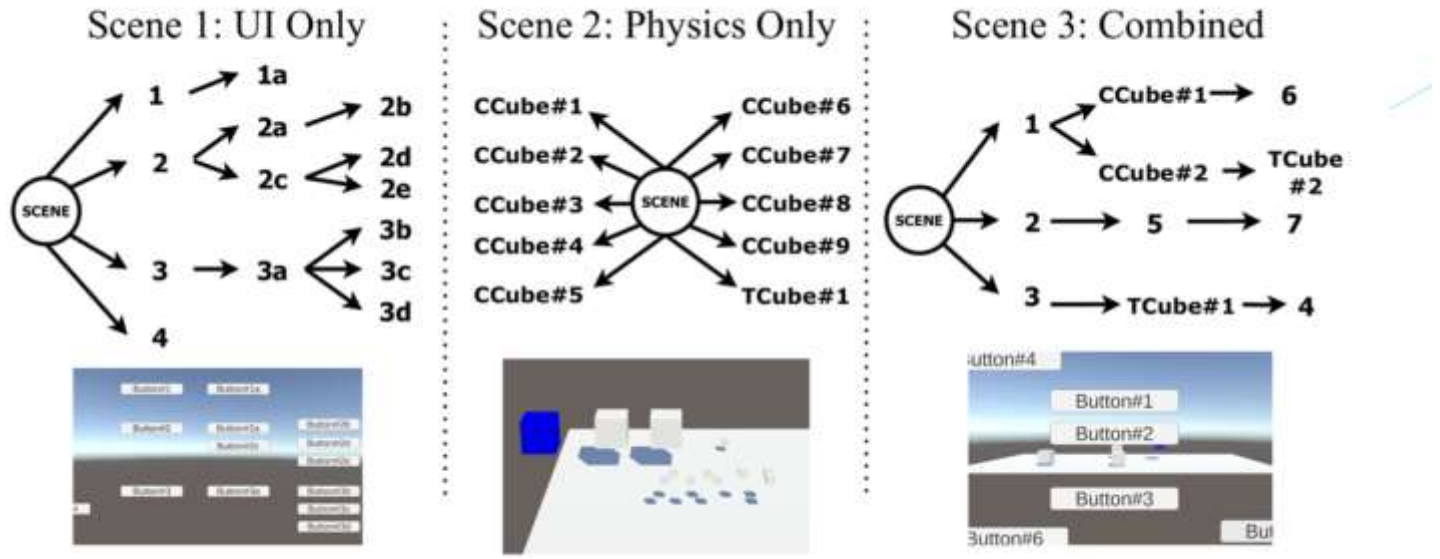


Figure 8: Custom VR app describing the dependency structure for each scene, where each alphanumerical value (e.g., 1, 2d, 3a, etc.) indicates a UI event, each "Cube" prefixed with "C" indicates a collisionable event and "T" indicates a triggerable event.

Experiment

实验对象及规模

- 其他 Unity Apps:
  - 263 免费App(84个来自Meta Quest store，179 个来自SideQuest store)
  - 103 个来自Meta Quest app store 的付费App
  - 总共366个App

Experiment

事件触发结果

- Custom Scene的三个场景的Event 触发情况 （对比Monkey 方法明显更优）

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`).

Experiment

- 366个Unity App 的信息

Rating	Free Apps							Paid Apps						
	# 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.

Experiment

数据流发生次数结果

- Unity Apps 上的数据流发生次数

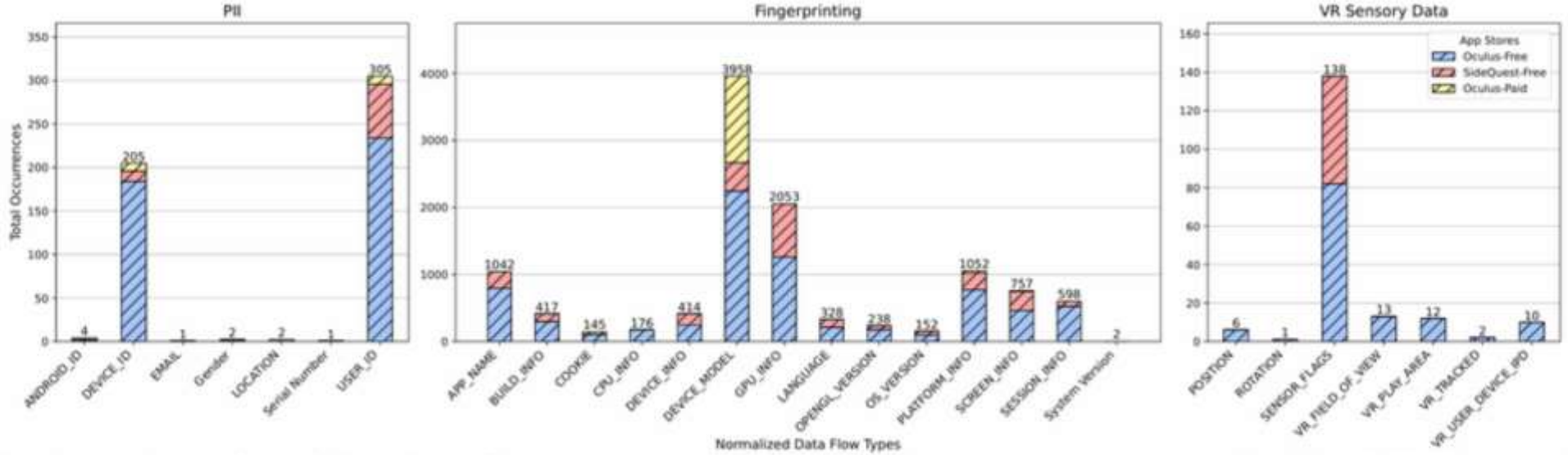
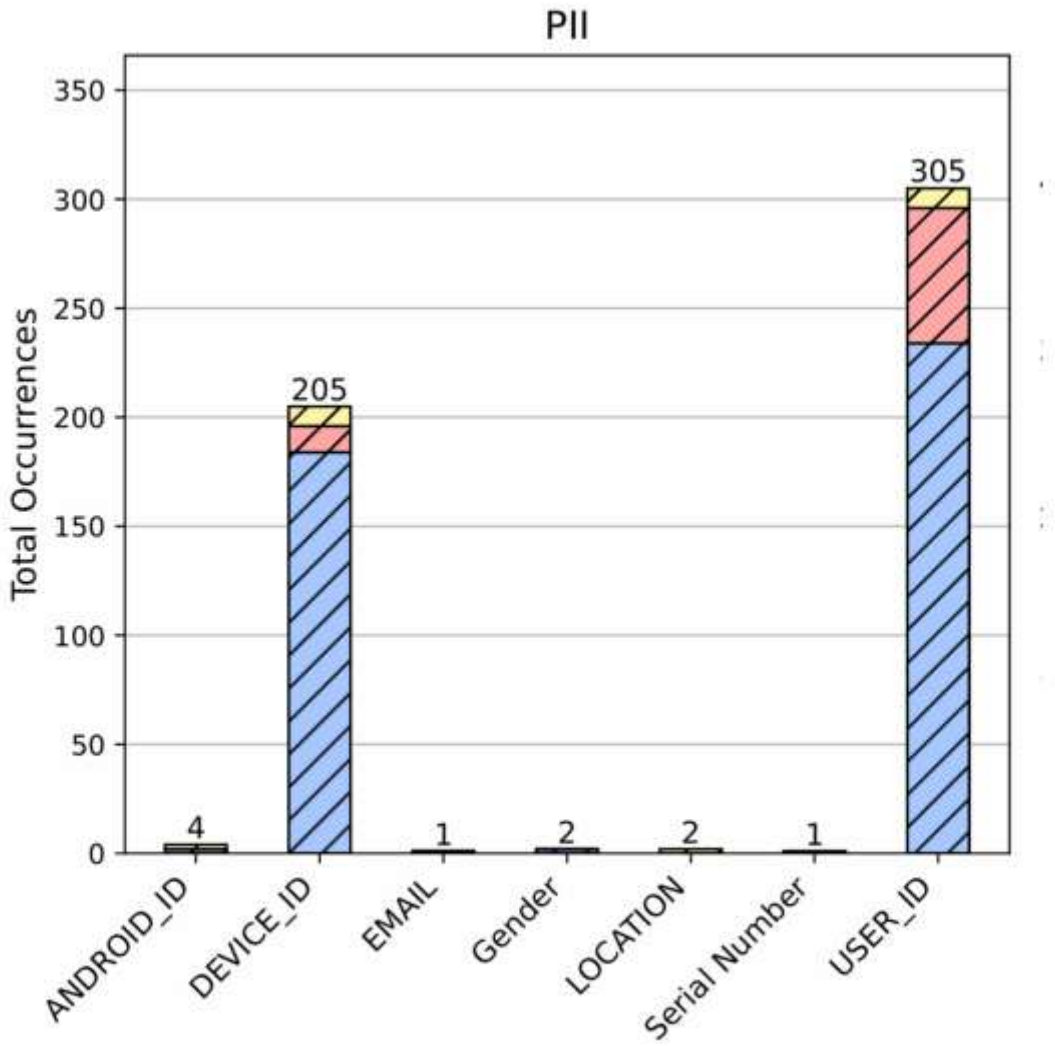


Figure 10: Total number of sensitive data flow occurrences grouped by app store, categorized by PII (Personal Identifiable Information), Fingerprinting, VR Sensory Data.

Experiment

数据流发生次数结果

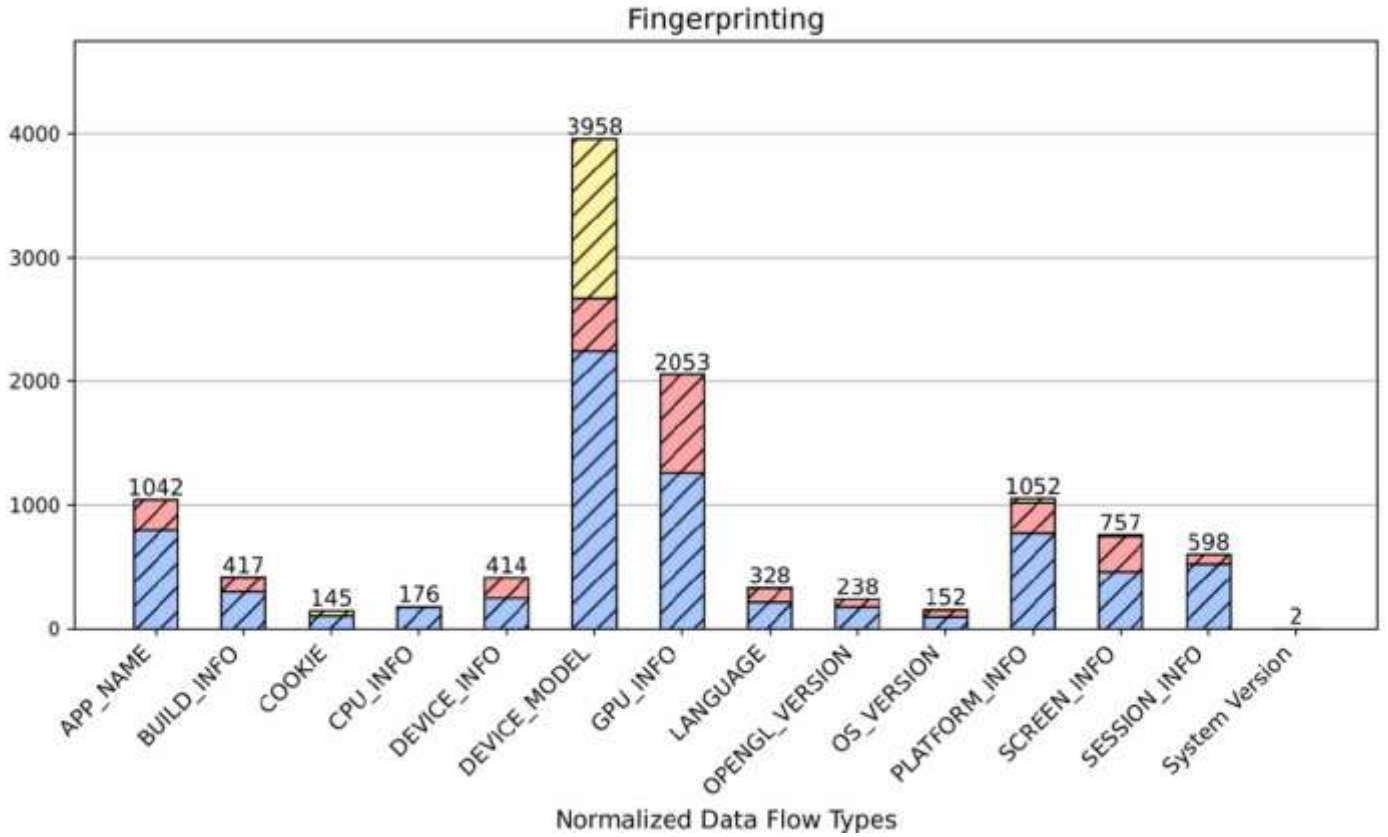
- 数字指纹相关信息中，Device_ID和User_ID数据流最多
- 免费项目次数明显更多



Experiment

数据流发生次数结果

- 常规数据流中DEVICE_MODEL占比最多



Experiment

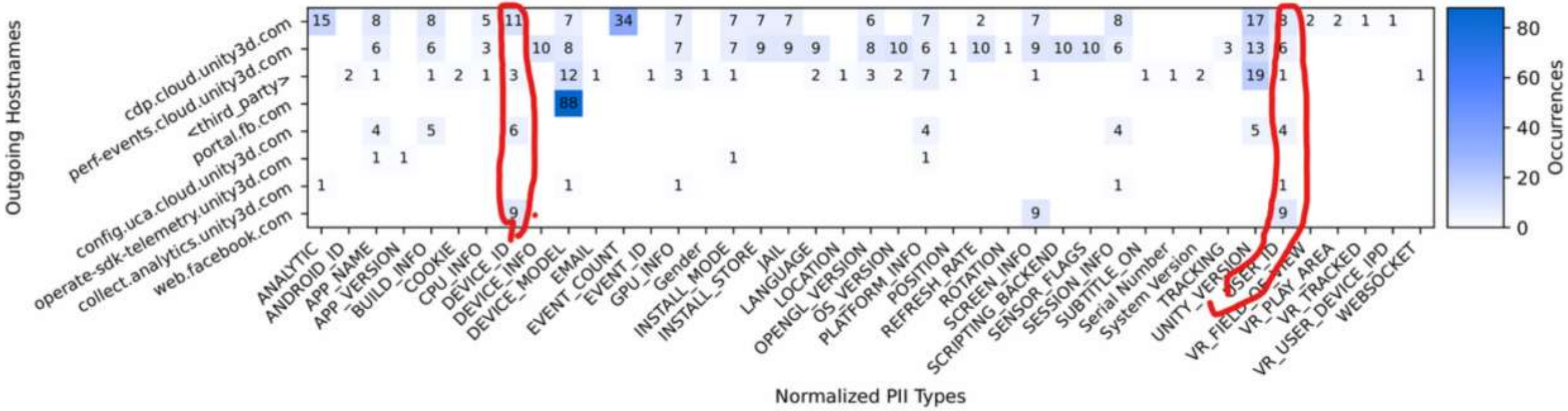
结论

- 排名前几位的数据类型有 **APP_INFO、PLATFORM_INFO、SESSION_DATA**，这些往往可用于**数字指纹（digital fingerprinting）**。
- 诸如 **SCREEN_INFO、GPU_INFO、CPU_INFO、DEVICE_INFO** 等信息被收集以用于更精确地跟踪用户行为与可识别性。
- 在可解密的数据中，**USER_ID 与 DEVICE_ID** 是最稳定的标识数据。
- **付费应用通常做得更规范/安全**（实验中多数付费 App 评分 ≥3.5），因此更可能在应用层做了额外加密；而**免费应用**更多依赖 Unity 内置分析工具，因而更频繁地向 **Unity 的分析域名发送数据。**

Experiment

结论

- DEVICE_ID 和 USER_ID 分布更稳定



Experiment

数据流捕获对比

- Monkey & AutoVR：都进行20分钟运行

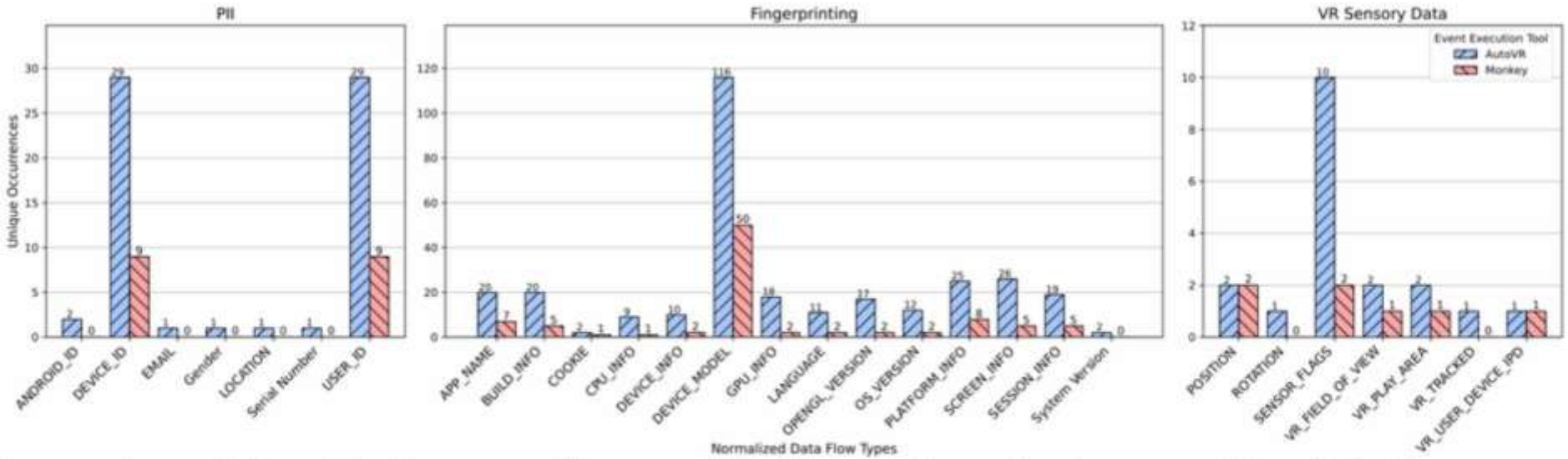
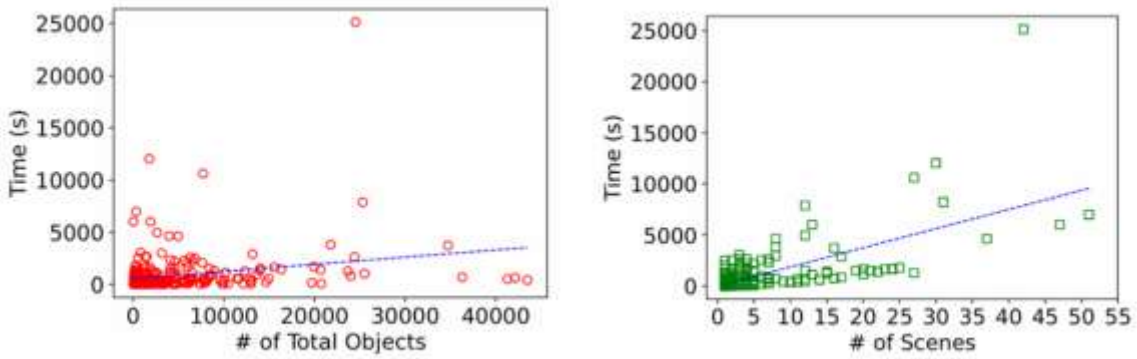


Figure 11: Comparison of the effectiveness with respect to the number of unique sensitive data flow occurrences found using AUTOVR vs Monkey.

Experiment

数据流捕获对比

- 由于AUTOVR能够触发更多的物体及其UI/物理事件，能够捕获更多的数据流。
- AUTOVR：平均4分钟不到每个项目（222.40s；Monkey显然时间更短，但是捕获的数据流更少
- 作者发现物体/场景和Runtime强相关：（显而易见）



(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.

Discussion

局限性

- AUTOVR 能够捕捉到数据流，但并不是所有数据流都泄露用户隐私，AUTOVR还不具备识别是否泄露用户隐私的能力。例如：通过拦截网络数据包，发现用户名、密码，但是这些数据可能属于开发者，并不是用户的数据，因此不对用户构成隐私泄露风险。因此需额外的上下文分析等操作

Discussion

对比SOTA

- 现有工具的局限性：Monkey、DroidBot、AutoDroid 等工具都基于 MonkeyRunner， 即通过 模拟屏幕操作（点击、滑动、双击） 来触发2D UI 事件。
- 而作者提出的AUTOVR
  - **不依赖屏幕图像或 VR 控制器操作**；
  - 而是直接在 **IL2CPP 编译层（即 Unity 的底层二进制层）** 触发事件。
  - 种做法绕过了传统输入层的限制，可以直接“命令” VR 内部逻辑触发事件。因此 AUTOVR 能够有效自动化 VR 游戏的交互与动态分析。

Future Work

- **整合符号执行 (Integration of symbolic execution)**
  - 未来，还可以整合**符合变量**，通过它代替具体输入，从而**求解约束（Constraints Solving）**探索所有可能的执行路径。
  - 求解约束（Constraint Solving）就是**自动计算出一组输入，使得程序逻辑条件成立**的过程。

Future Work

- **整合符号执行 (Integration of symbolic execution)**
  - 具体来说，Unity 的事件系统（UnityEvent）经常要求**复杂对象类型**作为参数（比如 Transform、GameObject、Vector3 等）。
  - 这些参数不是普通的数值类型（int、bool），而是结构体或类对象。因此，要想在符号执行中处理这些输入， 可以集成 **SMT 求解器**（例如 Z3）；
  - 当事件参数可以被**抽象**成符号变量时，SMT 求解器就能自动计算哪些输入可以触发特定事件；