

Received 2 September 2026, accepted 17 September 2026, date of publication 21 September 2026,
date of current version 30 September 2026.

Digital Object Identifier 10.1109/ACCESS.2026.3735987

RESEARCH ARTICLE

eSPADE-XR: Static-Runtime Cross-Verification of XR Permission Evidence in Android XR

HYOJOONG JU¹, HYEONJUN PARK¹, AND SEUNGSOO LEE^{1,2}, (Associate Member, IEEE)

¹Department of Computer Engineering, Incheon National University, Incheon 22012, Republic of Korea

²Human-Centered Artificial Intelligence Research Center, Incheon National University, Incheon 22012, Republic of Korea

Corresponding author: Seungsoo Lee (seungsoo@inu.ac.kr)

This work was supported by Incheon National University (International Cooperative) Research Grant, in 2026.

ABSTRACT Static XR permission evidence does not necessarily indicate runtime use, while Binder IPC exposes only low-level metadata. Static artifacts may originate from unused libraries or framework code, and raw transaction codes remain interface-local unless interpreted with receiver and version context. We present eSPADE-XR, a static–runtime cross-verification framework that generates static XR permission candidates and cross-verifies them using eBPF-based Binder observation and interface-aware mappings. Across 31 Android XR apps, eSPADE-XR classified 204 app–permission pairs and selected the 51 manifest-declared pairs for runtime observation. Three 3-second repetitions observed runtime-discovery evidence for 40, 42, and 44 candidates; 10-second repetitions produced comparable counts of 39, 43, and 44. The controlled checks observed the expected `IContentProvider.query` label in 5/5 runs and the endpoint-confirmed `IXrRuntimeServer.createNode` label in 4/5 runs. The commercial evaluation therefore reports a target-attributed `IContentProvider.query` reaching the known OpenXR runtime-discovery receiver as candidate-level evidence, while permission enforcement, authorization correctness, and permission-specific feature or sensor use remain outside its measurement scope. These results demonstrate practical candidate-level cross-verification of static XR evidence against interpretable runtime observations in the evaluated emulator environment and Android build.

INDEX TERMS Extended reality, permission, XR security, eBPF, Binder IPC.

I. INTRODUCTION

High-performance extended reality (XR) technologies are increasingly being deployed across a wide range of industrial and consumer domains. Unlike smartphones, XR devices require continuous sensing capabilities to maximize user experience by collecting eye gaze, hand motion, facial signals, and spatial mapping information in real time [1], [2]. However, the persistent collection of such high-resolution sensitive data introduces serious privacy risks and can expand the attack surface of both applications and system services [3], [4], [5], [6], [7], [8]. In particular, Android-based XR platforms have more complex data collection and processing paths than conventional mobile platforms.

The associate editor coordinating the review of this manuscript and approving it for publication was Cong Pu¹.

Because applications do not directly invoke sensor APIs for all XR resources but instead rely on abstracted runtimes and system services, it becomes difficult to cross-check manifest-declared permissions against Broker-level runtime evidence [9], [10], [11], [12], [13].

This architectural complexity limits the effectiveness of conventional static permission analysis. In Android XR applications, static artifacts such as manifest permissions, API references, strings, or SDK-level code fragments may indicate potential XR functionality, but they do not necessarily imply that the corresponding functionality is executed at runtime. This problem is further amplified by Unity/IL2CPP-based application structures, vendor-provided software development kits (SDKs), and runtime libraries, which can introduce boilerplate or unused references into application packages. Therefore, static analysis

results should be treated as runtime-observation candidates rather than as conclusive evidence of XR resource usage.

Dynamic observation can complement static analysis, but Android XR introduces an additional semantic challenge. XR-related requests are typically mediated by the XR Runtime, XR Broker, and system services through Binder inter-process communication (IPC). Although kernel-level Binder observation can capture low-level runtime metadata, such as sender PID/TGID, receiver PID, and transaction code, these raw transaction records do not directly reveal the XR control method being invoked. As a result, runtime observation alone cannot provide interpretable runtime evidence unless the semantic gap between Binder transaction codes and XR control methods is resolved.

To address these limitations, we propose eSPADE-XR, a static-runtime cross-verification framework that relates static XR permission candidates to observable XR runtime evidence in Android XR architectures. eSPADE-XR redefines static analysis as a candidate generation stage, creates targeted runtime observation windows from selected components, and collects Binder IPC metadata attributable to the target application using extended Berkeley Packet Filter (eBPF). It then combines observed receiver context with interface-aware, version-matched Android Interface Definition Language (AIDL) mappings to interpret transaction codes. The goal of eSPADE-XR is to cross-verify static XR candidates against observable runtime evidence during candidate-guided execution. We use the term candidate-level permission-evidence consistency to denote agreement between statically derived XR permission candidates and runtime evidence observed during targeted execution; it does not denote permission enforcement, authorization correctness, permission-specific feature execution, sensor access, or sensor-data flow.

The main contributions of this paper are summarized as follows:

- We present a candidate-oriented cross-verification model for Android XR applications, where static analysis is used to generate runtime observation targets rather than final threat labels.
- We design an eBPF-based Binder IPC observation mechanism that collects Binder IPC metadata attributable to the target application without modifying the application or the XR runtime.
- We construct an interface-aware semantic interpretation process that combines Binder receiver context with version-matched mappings to interpret transaction codes.
- We quantitatively evaluate eSPADE-XR through candidate analysis across 31 commercial Android XR applications, controlled query and `createNode` method checks, non-XR negative controls, repeated 3- and 10-second observation-window measurements, and runtime monitoring-cost measurements.

II. BACKGROUND

A. XR ARCHITECTURE AND BINDER IPC COMMUNICATION

In a conventional Android environment [14], applications request sensor events through standard framework APIs such as `SensorManager`, while actual hardware access is mediated by the Android framework and system service layers. In contrast, Android-based XR platforms employ a more complex runtime architecture than general mobile devices in order to process advanced continuous sensing data, such as eye gaze, facial signals, hand movements, and spatial information, in real time. As part of this security-oriented design, individual applications are prevented from directly accessing hardware device nodes in kernel space. Instead, applications remain isolated within their sandboxes, and XR-related functional requests are processed through intermediate layers such as the runtime, XR Broker, and system services [10], [15].

In this architecture, the XR runtime layer acts as a key mediation point for control requests between applications and lower-level system services. As illustrated in Fig. 1, when an application initializes continuous sensing functions for user interaction or creates nodes in a virtual space, Binder-based inter-process communication (IPC) typically occurs between the application and the XR Runtime. In this context, Binder IPC should be interpreted not as the transport path for sensor data itself, but as a control path for initializing XR functions or requesting related system resources.

The XR Runtime then forwards requests to lower layers through different paths depending on the nature of the request. General control commands, such as node creation or sensor initialization, are delivered to the hardware abstraction layer (HAL) [16] or related system services through IPC mechanisms such as AIDL or `HwBinder`, and subsequently interact with kernel drivers. In contrast, for continuous sensing streams or hardware-accelerated operations that require ultra-low latency to reduce motion sickness, separate data paths such as shared memory [17] or Fast Message Queue (FMQ) [18] may be used instead of transferring data through Binder IPC for every event. In such cases, Binder may be used as an initial control path for establishing the data path or passing handles, after which the runtime processes sensing data through memory-mapped regions [17].

Therefore, although sensing data itself may flow through non-Binder paths for performance optimization, the initial gateway request for initiating such operations and allocating hardware resources can still be observed as Binder IPC through the XR Broker or runtime layer. This architectural property makes Binder IPC a useful runtime signal for determining whether an application has entered an XR control path.

B. ANDROID INTERFACE DEFINITION LANGUAGE

When different processes exchange data within the Android system, the Binder IPC mechanism is commonly used,

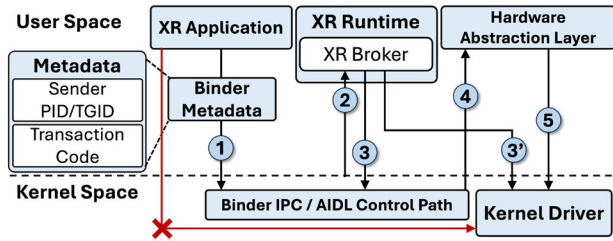


FIGURE 1. Android XR runtime architecture and control-path observation scope. Binder IPC provides observable control-path metadata, while high-rate sensor payloads may follow non-Binder data paths.

and Android Interface Definition Language (AIDL) [19] is employed to define the inter-process calling interface. AIDL defines an interface shared between a client and a service, and method invocations across processes are translated into Binder transactions based on this interface.

For standard Android framework services, AIDL interfaces are publicly available through the Android Open Source Project (AOSP), and the relationship between interface definitions and transaction codes can be identified relatively clearly. However, in XR runtime services, internal control methods are often encapsulated. As a result, when Binder IPC is observed at the kernel level, only low-level metadata such as sender PID/TGID, receiver PID, and transaction code can be obtained instead of method names. This creates a semantic gap introduced by the abstracted communication layer, which must be addressed to interpret runtime observations using receiver- and version-specific mappings for static-runtime cross-verification.

C. EXTENDED BERKELEY PACKET FILTER

Extended Berkeley Packet Filter (eBPF) [20] is a mechanism that enables user-defined observation programs to run safely in kernel space without modifying the operating system kernel source code or adding kernel modules [21]. In Android systems, eBPF can be used to trace various events occurring in kernel and user space through kernel-supported instrumentation interfaces.

Modern Android environments generally enable BPF and related kernel options by default, making them suitable for event capture through predefined static instrumentation points such as tracepoints. This allows a stable runtime observation environment to be constructed for collecting low-level kernel events, including the aforementioned Binder communication records and system calls, without modifying user-space XR applications or the XR runtime.

III. PROBLEM STATEMENT

Continuous sensing data in extended reality (XR) environments is highly sensitive because, once exposed, it cannot be revoked or changed like conventional credentials [3], [9], [22]. Therefore, strict permission control is essential. However, despite handling such high-risk data, the current Android XR platform ecosystem has three structural

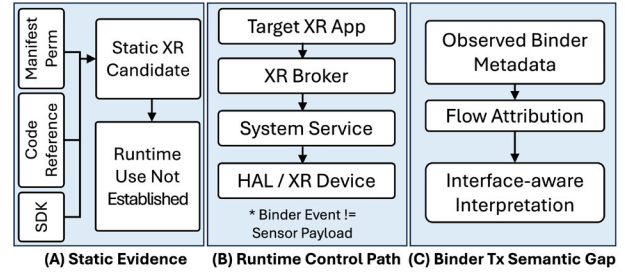


FIGURE 2. Structural challenges in static-runtime cross-verification of Android XR permission evidence. The figure summarizes the static-analysis ambiguity, mediated XR control path, and Binder semantic-gap issues addressed by eSPADE-XR.

limitations in static-runtime cross-verification of XR permission evidence.

A. P1: LIMITATIONS OF STATIC XR PERMISSION EVIDENCE

Existing static analysis approaches [23], [24], [25], [26] have primarily relied on comparing permissions declared in the application manifest with keywords or API references found in the application code. However, XR applications often include complex components such as Unity/IL2CPP, multiple software development kits (SDKs), and runtime libraries to implement three-dimensional environments. As a result, XR-related artifacts discovered statically do not necessarily indicate that the corresponding functionality is actually executed at runtime [27], [28]. Moreover, even if a sensitive XR permission is declared in the manifest, static analysis alone cannot determine whether corresponding runtime evidence is observed during execution. Therefore, as shown in Fig. 2, static analysis results should be interpreted not as conclusive evidence of actual XR resource usage or threats, but as runtime-observation candidates.

B. P2: ARCHITECTURAL COMPLEXITY AND RUNTIME CONTROL-PATH OBSERVATION IN XR

In Android XR environments, applications do not directly access hardware sensors or XR resources. Instead, they operate through a complex architecture mediated by intermediate layers such as the XR Broker. This indirect access model makes it difficult to relate static code artifacts to observed runtime behavior. In addition, because sensor data itself may be delivered through shared memory, Fast Message Queue (FMQ), hardware abstraction layer (HAL) paths, or other data paths, Binder IPC provides an observable signal of interaction with XR runtime endpoints and mediated control paths, not evidence of permission-specific feature or sensor use [16], [17], [18]. Therefore, as illustrated in Fig. 2, observing Binder transactions from the target application at the relevant XR receiver requires a mechanism that can observe Binder IPC between the target application and the receiver at the kernel level without directly modifying the application or the XR runtime.

C. P3: SEMANTIC INTERPRETATION GAP IN BINDER IPC TRANSACTIONS

Even when Binder IPC is observed at the kernel level, the collected information is limited to low-level metadata such as sender PID/TGID, receiver PID, and transaction code [29], [30]. In particular, because the XR Broker may process requests from multiple applications and system components, the mere observation of a Broker-related Binder event is insufficient to attribute the event to a specific application's XR request. Furthermore, raw transaction codes do not reveal which XR control method a call corresponds to. Therefore, as shown in Fig. 2, it is necessary to distinguish the flow from the target application to the XR Broker based on the PID/TGID of the target application and the Broker PID, and then interpret the transaction using its receiver context and an interface-aware, version-matched mapping.

IV. RELATED WORK

Static analysis has been widely used in Android application security research to estimate whether applications may access sensitive resources by analyzing manifest permission declarations and API calls in application code [11], [12], [13], [23], [24], [25]. This approach is effective for large-scale application analysis because it can identify permission declarations, API references, strings, classes, and method-level indicators without executing applications. In the mobile application ecosystem, static analysis has played an important role in analyzing overprivileged permission declarations, sensitive API calls, mismatches between permissions and APIs, and the possibility of permission use introduced by third-party libraries across a large number of applications. However, static analysis inherently estimates possibilities based on artifacts present in code or application packages, and therefore cannot directly guarantee that such artifacts are executed at runtime.

This limitation becomes more significant in applications that include SDKs, obfuscated code, native libraries, or runtime frameworks [26], [27], [28]. API references or strings included in an application package may be part of runtime functionality, but they may also originate from unused library code, SDK boilerplate, build-time template code, or dead code. Interpreting such permission-related artifacts found by static analysis as conclusive runtime evidence may therefore lead to false attribution. This issue is particularly important for Unity/IL2CPP-based XR applications. XR-related references may be distributed across multiple layers, including the Java/Kotlin layer, native binaries, `libil2cpp.so`, `global-metadata.dat`, Unity runtime libraries, and vendor SDKs. As a result, simple string matching or API reference analysis is insufficient to determine whether corresponding XR runtime behavior is observed during execution.

XR privacy studies have emphasized the sensitivity of continuous sensing data, such as eye gaze, hand motion,

facial signals, and spatial mapping information, and have highlighted the need for fine-grained permission control in immersive environments [3], [4], [5]. Prior studies have shown that behavioral and motion data in XR environments can expose users' physical characteristics, behavioral patterns, surrounding spatial structures, attention direction, and interaction habits, and that such information can be used for user identification or behavioral inference [6], [7], [8], [22], [31]. These characteristics create privacy risks that differ from those associated with location, camera, or microphone permissions in conventional mobile environments. For example, data such as eye gaze or hand tracking is not merely an input event, but may be related to users' cognitive states, behavioral intentions, and biometric characteristics. Once exposed, such information cannot be easily changed or revoked like a password.

However, many XR privacy studies have primarily focused on data sensitivity, user perception, privacy policies, permission models, or user consent mechanisms. These studies provide important insights into why XR sensing data is sensitive, what types of permission control models are needed, and how users perceive privacy risks. In contrast, the system-level cross-verification problem of relating static permission candidates found inside Android XR application packages to observed runtime behavior has not been sufficiently addressed. This issue is especially important in Android XR environments because applications often request XR functionality through intermediate layers such as the XR Runtime, XR Broker, system services, and the HAL, rather than directly accessing sensors or hardware resources. Therefore, permission declarations or code references alone are insufficient to determine runtime evidence attributable to the target application.

Dynamic analysis and automated testing can complement static analysis by executing applications and observing runtime behavior. Existing Android dynamic analysis studies have observed application behavior during execution through UI automation, system call tracing, network traffic monitoring, API hooking, and taint analysis. These approaches are useful because they can capture runtime behaviors that static analysis alone cannot confirm. However, XR applications have more complex execution conditions than conventional mobile applications. XR-related code paths may not be activated simply by launching an application. They may require specific session initialization, entry into a virtual space, device state changes, controller or hand tracking states, gesture-based interactions, or spatial anchor creation. As a result, existing dynamic analysis tools or random UI exploration methods such as Android Monkey may fail to reliably reach XR-specific execution paths [32], [33].

Automated exploration studies for XR or VR applications have attempted to mitigate these limitations by generating interaction sequences in virtual environments, exploring UI states, or injecting specific events. These approaches may reach deeper execution paths in XR applications than general

Android UI exploration. However, the problem addressed in this study is not merely to execute more internal application paths. Rather, it is to cross-verify static XR permission candidates against Binder observations attributed to the target application in the Android XR architecture. In other words, automated exploration may help activate certain code paths, but additional system-level interpretation is required to determine whether a transaction from the target application is observed at the relevant XR receiver and which endpoint- and interface-level semantics are supported by the observed transactions.

In Android systems, Binder IPC is a core inter-process communication mechanism between applications and system services. Observing Binder transactions at the kernel level can collect metadata such as sender PID and TGID, receiver PID, and transaction code, which is useful for determining whether an application communicated with a specific system service or runtime component [29], [34]. eBPF provides a mechanism for collecting tracepoint-based runtime events without directly modifying the kernel, enabling Binder IPC metadata to be observed without user-space hooking. This characteristic is suitable for observing runtime control paths in closed XR environments where applications or the XR Runtime are difficult to modify directly.

However, Binder IPC metadata alone is insufficient. Binder transactions are generally represented as integer transaction codes, and raw transaction codes observed at the kernel level do not directly reveal which service methods were invoked. In addition, because the XR Broker may simultaneously communicate with the target application, other applications, system services, and runtime components, the mere observation of Broker-related Binder events is insufficient to associate those events with a specific XR permission candidate of the target application. It is therefore necessary to distinguish transactions from the target application using sender and receiver identities, and then combine receiver context with an interface-aware, version-matched mapping to interpret a method only when the receiver, interface, and build mapping agree [19], [30]. Without this interface-aware interpretation process, runtime observations remain at the level of low-level metadata and cannot relate XR permission candidates to interpretable runtime evidence.

Unlike prior studies, eSPADE-XR focuses on cross-verifying static XR permission candidates with runtime observations attributable to the target application in the Android XR architecture. eSPADE-XR does not treat static analysis results as final evidence of runtime XR resource use or security threats. Instead, it redefines them as candidate generation inputs for runtime evidence collection. It then creates observable runtime windows through targeted execution of candidate components and collects Binder transaction metadata from the target application using eBPF-based Binder IPC observation. Finally, it applies interface-aware semantic interpretation with version-matched mappings to interpret supported methods.

Thus, eSPADE-XR complements prior research on Android static analysis, XR privacy, dynamic analysis, and UI exploration. While existing static analysis studies are effective for large-scale candidate identification, eSPADE-XR collects runtime observations attributable to the target application for those candidates. While prior XR privacy studies establish the sensitivity of continuous sensing data and the need for permission control, eSPADE-XR cross-verifies such permission candidates in Android XR applications with observable XR runtime evidence. In addition, while existing dynamic analysis studies focus on observing behaviors during application execution, eSPADE-XR combines Binder IPC direction analysis with interface-aware transaction interpretation to transform raw runtime metadata into candidate-level semantic evidence. Through this design, eSPADE-XR provides a system-level cross-verification framework linking static XR permission candidates with interpretable runtime evidence.

Table 1 compares representative static-analysis and XR/VR dynamic-analysis approaches with eSPADE-XR at the analysis-capability level.

V. eSPADE-XR DESIGN

As shown in Fig. 3, eSPADE-XR is a hybrid framework that takes the base directory path of decompiled target applications as input and executes a pipeline consisting of static candidate generation, runtime control-path observation, and evidence integration and evaluation. The overall system consists of three phases: the candidate generation phase, which structures verification targets through static analysis; the runtime observation phase, which observes kernel-level Binder IPC during candidate-guided execution windows; and the cross-verification and evaluation phase, which integrates multi-layer data and produces candidate-level evidence results. Each phase controls the data flow through its internal components.

A. DESIGN CONSIDERATIONS

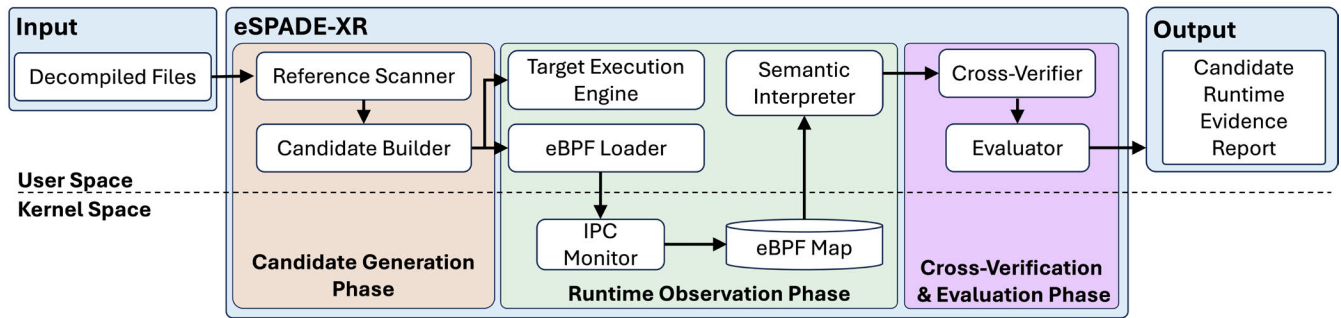
To address the gap between static XR permission evidence and observable runtime behavior defined above, this study derives three key design requirements that a new hybrid analysis framework should satisfy.

1) REDEFINING STATIC ANALYSIS AS CANDIDATE GENERATION

Existing Android permission analysis techniques have primarily used static analysis results as final indicators for threat detection. However, in XR environments, application packages may include complex components such as three-dimensional runtime engines including Unity, vendor-provided software development kits (SDKs), runtime libraries, and boilerplate code. Therefore, static analysis alone cannot reliably determine whether corresponding runtime behavior is observed during execution. Accordingly, eSPADE-XR redefines the role of static analysis not as

TABLE 1. Comparison with representative static-analysis and XR/VR dynamic-analysis approaches. Binder IPC and interface semantics denote system-level IPC observation and context-aware transaction interpretation, respectively.

Approach	Static app	Runtime	XR/VR-specific	Binder IPC	Interface semantics
PScout [23]	Yes	–	–	–	–
FlowDroid [24]	Yes	–	–	–	–
OVRseen [31]	–	Yes	Yes	–	–
AutoVR [33]	Yes	Yes	Yes	–	–
eSPADE-XR	Yes	Yes	Yes	Yes	Yes

**FIGURE 3.** Overall architecture of eSPADE-XR. The framework generates static XR permission candidates, executes selected components, collects Binder IPC metadata with eBPF, interprets transactions using receiver context and interface-aware mappings, and integrates the results into candidate-level XR evidence.

final detection, but as target extraction and candidate generation for subsequent dynamic analysis. Specifically, it compares permissions declared in the application manifest with XR-related APIs, references, and methods discovered in decompiled code, and structures them as runtime evidence candidates, each an app–permission pair with associated suspect components. These candidates determine the observation targets for the subsequent dynamic analysis stage.

2) OBSERVING CONTROL PATHS THROUGH KERNEL-LEVEL IPC OBSERVATION

XR applications perform functional requests through intermediate layers such as the XR runtime, XR Broker, and system services, rather than directly accessing hardware sensor resources. Under this indirect access model, corresponding runtime behavior cannot be established from application code alone, and conventional user-space hooking requires instrumentation within user-space components. In addition, even if Binder IPC is observed, the collected data is limited to low-level metadata, which creates a semantic gap in understanding the method represented by the transaction. To address this issue, eSPADE-XR adopts an eBPF-based design that collects Binder transaction metadata from trace-points in the Linux kernel. This enables observation of Binder metadata attributable to the target application without directly modifying the application or the XR runtime.

3) IDENTIFYING IPC FLOWS AND INTERPRETING SUPPORTED TRANSACTIONS

Because the XR Broker can process requests from multiple applications and system components simultaneously,

the mere observation of a Broker-related Binder event is insufficient to determine that the event corresponds to a specific application’s XR control request. Therefore, eSPADE-XR identifies Binder transactions directed from the application to the Broker based on the PID/TGID of the target application and the PID of the XR Broker, while treating Broker-sent transactions to non-target receivers as same-window supporting observations. In addition, Binder transactions collected at the kernel level are represented as integer transaction codes rather than method names. eSPADE-XR combines Binder receiver context with an interface-aware, version-matched mapping to interpret supported methods. Through this process, raw IPC metadata is transformed into human-interpretable semantic evidence, and static candidates can be evaluated according to whether they are associated with observed runtime events.

B. OVERALL ARCHITECTURE

The first phase, candidate generation, takes the base directory path of decompiled target applications as input. It first parses each application’s `AndroidManifest.xml` file and logically groups target directories by package name. The Reference Scanner then scans the metadata of three-dimensional runtime engines and the Smali code layer within the grouped directories to locate regions where XR-specific permissions are referenced. At this stage, static analysis results are interpreted not as conclusive runtime evidence, but as candidates for runtime evidence collection. The Candidate Builder structures the discovered static evidence as app–permission pairs, associates each pair with suspect components, and selects primary candidates for

dynamic observation. The generated static candidates are stored in `01_static_candidates.jsonl` and passed to the next pipeline stage.

The second phase, runtime observation, monitors dynamic kernel communication based on the JSONL file generated by static analysis. The Target Execution Engine uses candidate information to create observable runtime windows, obtains the actual PID of the target application, and attaches the eBPF Collector using that PID. The IPC Monitor then filters and collects transaction metadata attributable to the target application from Binder-related tracepoints, and records the metadata in an eBPF map in real time. The collected transactions are interpreted from their receiver context using the corresponding interface-aware, version-matched mapping only when the receiver, interface, and build mapping agree; otherwise they remain uninterpreted.

The final phase, cross-verification and evaluation, integrates the outputs of the preceding modules. The Evaluator links candidates generated by static analysis with runtime observation results and summarizes the evidence retained for each accepted candidate-run. The final label records whether XR runtime-discovery evidence was observed, no corresponding interface evidence was observed, or the execution was invalid. The final results are exported in CSV format and summarized at the candidate level.

VI. eSPADE-XR DETAILS

A. MULTI-LAYER STATIC SCANNING AND STRUCTURAL NOISE REDUCTION

1) MANIFEST PARSING AND CODE-LEVEL XR EVIDENCE EXTRACTION

To address the limitations of static analysis discussed in P1 and to structure runtime observation targets, eSPADE-XR first performs a preprocessing step that filters the input directories. It parses `AndroidManifest.xml` files from multiple decompiled file paths to identify package names, and uses this information to logically group directories that may be distributed across split APKs into a single application unit. At the same time, it establishes a baseline of XR-specific permissions explicitly requested by each application. eSPADE-XR then performs multi-layer scanning to mitigate the structural opacity of Unity/IL2CPP-based XR applications. Specifically, it checks for the presence of the `libil2cpp.so` binary and `global-metadata.dat`, attempts to generate `dump.cs` using `IL2CPPDumper`, and falls back to string-based exploration if the dumping process fails. Using the recovered symbols and the Smali code layer of Java/Kotlin components, eSPADE-XR performs string- and regular-expression-based matching with an XR keyword set derived from official documentation [9], [10], thereby extracting primary code-level clues that may indicate access to hardware sensors or spatial information.

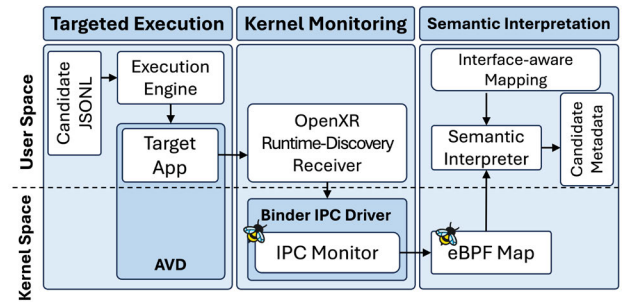


FIGURE 4. Targeted execution and kernel-level Binder IPC monitoring in the eSPADE-XR pipeline. Static candidates are executed in fixed observation windows, and Binder transactions are filtered using target and receiver context before interface-aware semantic interpretation.

2) STRUCTURING STATIC CANDIDATES THROUGH EVIDENCE MODELING

The extracted primary clues may include static code artifacts introduced by third-party libraries or three-dimensional engines, such as the Unity XR Interaction Toolkit. Considering this structural ambiguity, eSPADE-XR applies evidence modeling to the collected clues. Rather than merely determining the presence or absence of permission references, it compares XR permissions declared in the manifest with the presence of code-level evidence and classifies the results into *MATCH*, *OVER*, *GAP*, and *NONE*. Here, *MATCH* denotes a base-manifest declaration with corresponding permission-specific keyword/API-pattern evidence detected in the base or split APKs; *OVER*, a declaration without such detected evidence; *GAP*, such detected evidence without a declaration; and *NONE*, neither. These labels characterize static evidence configurations only; they do not denote vulnerability status, security severity, or runtime behavior. Among these, only *MATCH* and *OVER* candidates for which sensitive XR permissions are declared in the manifest are structured as targets for subsequent runtime observation because a permission absent from the manifest cannot normally be granted under the Android permission model; *GAP* cases therefore remain static findings outside the runtime cross-verification scope and are not treated as benign or false positives [9]. Across 34 normalized package units and six target XR permissions, the procedure generated 204 app-permission rows (49 *MATCH*, 2 *OVER*, 59 *GAP*, and 94 *NONE*); retaining the manifest-declared *MATCH/OVER* rows yielded 51 primary runtime candidates.

3) STRUCTURING TARGET COMPONENTS FOR TARGETED EXECUTION

Finally, eSPADE-XR structures which concrete application components, such as activities or services, are associated with the static candidates retained by the manifest-declared *MATCH/OVER* rule. To this end, it compares class information inferred from code evidence with manifest component

information and selects components to be executed in the subsequent stage. For each retained candidate, at most five `suspect_components` are ordered by code evidence, launcher status, exported activity/service status, Unity hints, and an activity fallback. The resulting candidates are stored in JSON Lines (JSONL) format, including the component name, target permission, and static label. This output serves as the key input for targeted execution in the subsequent pipeline, rather than for random exploration. Each retained runtime-observation candidate is a manifest-declared app–permission pair associated with an ordered list of `suspect_components`.

B. TARGETED EXECUTION AND KERNEL COMMUNICATION MONITORING

1) CREATING RUNTIME OBSERVATION WINDOWS THROUGH TARGETED EXECUTION

To observe runtime behavior associated with candidate components derived from static analysis, eSPADE-XR creates dynamic observation windows through targeted execution. In typical XR applications, relevant code paths may be activated only after entering a specific virtual space or receiving user gestures. As a result, conventional random UI exploration tools, such as Android Monkey, may fail to reach XR execution paths. For candidate-guided observation, eSPADE-XR parses the JSONL candidate report delivered from the preceding module, as shown in the Targeted Execution stage of Fig. 4, and iterates the `suspect_components` field for each primary runtime candidate. For activity components, the runner uses Android activity manager commands, including `adb shell am start -S -W -n package/component` for initial PID acquisition and `adb shell am start -W -n package/component` for the observation trigger. This process is not intended to reproduce the full sequence of normal user interactions or to guarantee XR session initialization, complete user-flow coverage, gesture-driven execution, or spatial interaction coverage. Rather, it is designed to create a reproducible window in which an OpenXR runtime-discovery transaction from the target application can be observed. For the final evaluation, the primary condition used a 3-second observation window over three repetitions, and the sensitivity condition used the same 51 candidates and execution logic with a 10-second window over three repetitions. Infrastructure-level execution failures were handled through recovery and retry procedures before retaining an accepted candidate-run result.

2) eBPF-BASED BINDER IPC METADATA COLLECTION

For the commercial evaluation, eSPADE-XR collects IPC metadata between the target application and the OpenXR runtime-discovery receiver in kernel space. A C-based eBPF program is attached to the `binder_transaction` tracepoint in the Linux kernel, and it filters Binder messages

TABLE 2. Runtime observation protocol used for targeted execution. The table lists launch commands, timing windows, PID lookup, and failure handling used to keep observations comparable.

Protocol element	Protocol value
Component source	<code>suspect_components</code> in the static candidate JSONL
Initial activity launch	<code>adb shell am start -S -W -n package/component</code>
Observation activity launch	<code>adb shell am start -W -n package/component</code>
Primary observation window	3 seconds
Primary repetitions	3
Sensitivity observation window	10 seconds
Sensitivity repetitions	3
Collector attach wait	0.6 seconds
Post-trigger wait	0.5 seconds
Target PID lookup	<code>pidof package</code> , 3 retries, 3 seconds per lookup
Runtime-discovery receiver PID lookup	OpenXR receiver/Broker package context via <code>pidof</code>
Failure handling	Recovery and retry precede one accepted candidate-run result

generated in the Android system by retaining transactions associated with the pre-identified PID of the receiver process in the Broker package context. The detailed filtering logic follows Algorithm 1. For each transaction event, the eBPF hook extracts the sender PID/TGID (s), receiver PID (r), and transaction code (c) (Lines 2–4). If the sender is the target application (`target_pid`) and the receiver is that process (`broker_pid`), the event is recorded as the primary application-to-receiver observation, and the corresponding transaction code is stored in an eBPF map (Lines 5–9). In addition, when a transaction sent by the XR Broker toward the target application or a non-target receiver is observed, it is separately recorded as a same-window supporting observation (Lines 10–16). The collected transaction data are stored in the eBPF map, following the Kernel Monitoring flow in Fig. 4, and are used in the subsequent stages to associate the static candidate with the observed runtime event and to support interface-aware interpretation of transaction codes based on receiver context. Because this kernel-level observation method collects runtime communication flows without directly modifying the application or the XR Runtime, it avoids user-space instrumentation of both components.

In addition to Binder transactions, eSPADE-XR also records whether the XR Broker performs callbacks to the target application and whether Broker-sent transactions to non-target receivers are observed within the same observation window. This information does not replace the primary evidence in the application-to-receiver direction. Instead, it is used as a same-window supporting observation that does not establish a causal continuation.

Algorithm 1 Binder IPC Metadata Collection for Application Attribution at the Runtime-Discovery Receiver. Application-to-Receiver transactions are stored as primary observations, While Broker Callbacks and Broker-Sent Transactions to Non-Target Receivers Are Retained as Same-Window Supporting Observations, not Causal Links

Input: binder transaction event t

Output: eBPF map components

```

1 hook binder_transaction( $t$ );
2  $s \leftarrow t.sender\_tgid$ ;
3  $r \leftarrow t.recv\_pid$ ;
4  $c \leftarrow t.code$ ;

5 if  $s = target\_pid$  then
6   map_update(Map_TargetToRecv,  $r$ );
7   if  $r = broker\_pid$  then
8     map_update(Map_TargetToBrokerCode,  $c$ );
9   map_update(Map_PrimaryEvidence, ( $s, r, c$ ));

10 if  $s = broker\_pid$  then
11   map_update(Map_BrokerToRecv,  $r$ );
12   map_update(Map_BrokerToCode,  $c$ );
13   if  $r = target\_pid$  then
14     map_update(Map_BrokerCallback,  $c$ );
15   else
16     map_update(Map_BrokerToService, ( $r, c$ ));

17 return;
```

C. BINDER IPC SEMANTIC INTERPRETATION AND DIRECTIONAL FLOW ANALYSIS

1) IDENTIFYING FLOWS AMONG THE TARGET APPLICATION, XR BROKER, AND NON-TARGET RECEIVERS

To integrate the multi-layer data derived from static analysis and runtime observation, eSPADE-XR first identifies the requesting subject and directionality of Binder IPC. Because the XR Broker may simultaneously process requests from multiple applications and system components, a Broker-related transaction alone cannot be interpreted as an XR control request from the target application. Accordingly, based on the sender PID/TGID, OpenXR runtime-discovery receiver PID, and transaction code collected during eBPF observation, eSPADE-XR prioritizes the identification of Binder transactions that occur from the target application to the OpenXR runtime-discovery receiver. This flow is used as primary evidence for determining whether a static candidate is associated with an observed application-to-receiver event. In contrast, Broker-sent transactions to non-target receivers are separately recorded as same-window supporting observations. This distinction reduces the risk

TABLE 3. Interface-aware mapping summary. The same transaction code is interpreted using its receiver context and the applicable interface mapping; the `IXrRuntimeServer` count reports inventory size, whereas empirical method coverage is limited to `createNode`.

Item	Scope in this study
Commercial discovery receiver	OpenXR ContentProvider in Broker package context
Commercial discovery interface	android.content.IContentProvider
Commercial discovery code 1	query
Controlled XR endpoint	cpm
Controlled XR interface	android.gui.IXrRuntimeServer
Version-matched IXrRuntime-Server mapping inventory entries	117
Empirically exercised IXrRuntimeServer method	<code>createNode</code> (code 1; 1 of 117 inventory entries)

of incorrectly attributing background system events or communications from other processes to the execution result of the target application, and retains the observed receiver context with each transaction code for subsequent interface-aware interpretation. However, this process is limited to distinguishing direction and receiver contexts within the same observation window. It does not reconstruct causal chains among individual Binder transactions.

2) INTERFACE-AWARE INTERPRETATION OF BINDER TRANSACTIONS

Even when Binder IPC metadata is collected at the kernel level, observed calls are represented as integer transaction codes rather than method names, leaving a semantic gap. eSPADE-XR resolves this gap by combining the observed receiver context with the applicable interface-aware mapping. Commercial runtime-discovery events use the known OpenXR runtime-discovery ContentProvider in the Broker package context, where transaction code 1 maps to `query` on `IContentProvider`. For the controlled cpm path, eSPADE-XR first verifies the `android.gui.IXrRuntimeServer` endpoint and descriptor and then applies the version-matched 117-entry mapping inventory, in which transaction code 1 maps to `createNode`. The controlled experiment exercises only `createNode`; the remaining entries are treated as build-specific mapping inventory rather than empirically validated methods. Keeping these mappings separate allows the same integer code to be interpreted according to its runtime context.

For the controlled cpm path, service registration information and `dumpsys` output identified the endpoint and descriptor, while `DUMP_TRANSACTION` confirmed the Binder node. A version-matched runtime mapping and proxy disassembly identified transaction code 1 as `createNode`.

Specifically, transactions attributed to the target application in the preceding flow identification stage are used as interpretation targets. This prevents Broker-related

transactions from being interpreted as the target application's execution flow solely because the XR Broker also processes requests from other applications or system components. The receiver context selects the applicable mapping. The endpoint and descriptor are first checked, after which a mapping derived for the evaluated build is compared with proxy/disassembly results to verify the code–method correspondence. A method label is assigned only when these checks agree; otherwise, the transaction remains uninterpreted. For another build, the mapping must be regenerated offline from available build artifacts and validated with a known call before use. This regeneration is not automated, and vendor-private or obfuscated interfaces remain uninterpreted when verifiable artifacts are unavailable.

Finally, the interpreted semantic result, transaction code, target permission, and observed component information are forwarded to the candidate-level evaluation stage and used to connect static candidates with interpretable runtime evidence.

VII. EVALUATION

A. EXPERIMENTAL DATASET AND SETUP

The goal of this evaluation is to examine whether eSPADE-XR can cross-verify Android XR permission candidates derived from static analysis with runtime observations. The commercial evaluation measures target-attributed `IContentProvider.query` transactions reaching the known OpenXR runtime-discovery receiver at the candidate level, while separate controlled experiments check the expected method labels under known endpoint and interface conditions.

The experiments were conducted on the Android Studio XR Headset Emulator running Android 14. The dataset consists of 31 Android XR applications collected from Google Play Store, including 21 Made for XR applications and 10 XR Ready applications. After APK decomposition and package-level normalization, split APK components and distributed files were grouped by package, resulting in 34 normalized package-level analysis units, with package identifiers available for all 34 units and `versionName/versionCode` metadata for 31. The dataset is used as a commercial-app evaluation sample rather than as a statistically representative sample of the Android XR ecosystem.

In the static analysis stage, eSPADE-XR collected XR permission declarations and code-level references from the Android manifest, Smali code, and Unity/IL2CPP artifacts. The six target permissions cover hand tracking, coarse- and fine-grained eye tracking, face tracking, and coarse- and fine-grained scene understanding. This process produced 204 app–permission pairs (49 *MATCH*, 2 *OVER*, 59 *GAP*, and 94 *NONE*); the 51 manifest-declared *MATCH/OVER* pairs formed the runtime-observation scope because an undeclared permission cannot normally be granted under the Android permission model, while *GAP* pairs remained static findings rather than benign or false-positive classifications [9].

TABLE 4. Dataset summary and final runtime-observation scope. The table reports the normalized app scale, app–permission pair counts, primary and sensitivity conditions, and negative-control scope.

Metric	Verified value
Collected commercial XR applications	31
Made for XR applications	21
XR Ready applications	10
Normalized package-level analysis units	34
App–permission pairs (34×6)	204
Manifest-declared runtime-observation pairs	51
Primary condition	3 seconds $\times$ 3 repetitions
Sensitivity condition	10 seconds $\times$ 3 repetitions
Non-XR control applications	10
Valid non-XR control runs	30

eSPADE-XR then used candidate-guided execution to create runtime observation windows and attached an eBPF-based Binder IPC monitor using the actual target application PID. The collected transactions were grouped by their observed sender and receiver context, interpreted with the applicable interface-aware mapping, and aggregated at the candidate level. If multiple app–permission pairs share a suspect component, the same component-level discovery behavior may be reflected in multiple pair-level results. Therefore, the commercial counts represent pair-level runtime-discovery associations, not distinct permission operations.

The final evaluation used a primary 3-second condition and a 10-second observation-window sensitivity condition. Both conditions retained the same 51 runtime-observation candidates and their suspect-component lists, collector filtering, and classification logic, and each was repeated three times. Recovery and retry handling preceded the accepted result for each candidate-run.

Table 4 summarizes the verified dataset scale and runtime observation scope used in the evaluation.

B. RUNTIME-OBSERVATION SCOPE SELECTION AND PRIMARY RUNTIME EVALUATION

The 204 app–permission pairs represent static evidence configurations and do not directly indicate whether the corresponding functionality is executed at runtime. For the runtime study, eSPADE-XR restricts observation to the 51 manifest-declared pairs under the scope rule defined by the static evidence model. Fig. 5 summarizes the quantitative results from static candidate generation to candidate-level evidence classification. Across the analyzed application set, 204 app–permission pairs were classified, and 51 manifest-declared pairs were selected as primary runtime candidates. The resulting 51/204 ratio reflects this runtime-observation scope definition; it is not a measure of detection accuracy or false-positive removal.

For the 51 selected primary runtime candidates, the primary 3-second evaluation observed XR runtime-discovery

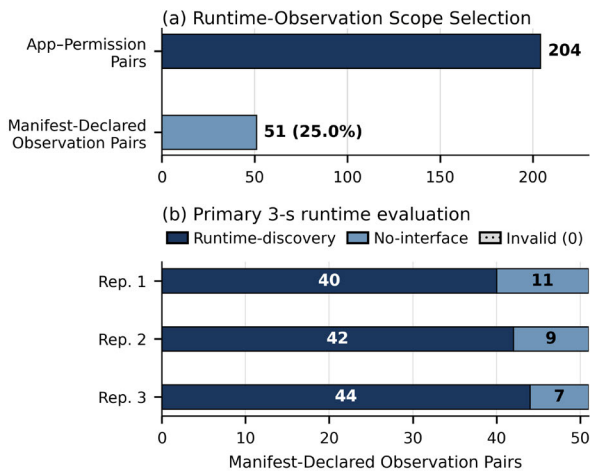


FIGURE 5. Runtime-observation scope selection and primary 3-second runtime evaluation. The figure shows the selection of 51 manifest-declared pairs from 204 app-permission pairs and the runtime-discovery, no-interface, and invalid counts across three repetitions.

evidence for 40, 42, and 44 candidates across repetitions 1–3. The corresponding no-interface counts were 11, 9, and 7, and no repetition contained an invalid execution.

XR runtime-discovery confirmed indicates that a target-attributed query transaction reached the known OpenXR runtime-discovery receiver context; it does not establish permission enforcement, authorization correctness, or permission-specific feature or sensor use. No interface evidence indicates a valid execution in which no corresponding event from the target application was observed during the post-attachment observation window, while invalid execution records candidates without usable execution evidence. These results show that eSPADE-XR does not treat static analysis results as final judgments, but uses them to select a runtime-observation scope and cross-checks the selected candidates against interpretable runtime evidence.

C. CONTROLLED METHOD-LABEL CHECKS AND NEGATIVE CONTROLS

We evaluated the semantic interpretation procedure with two controlled paths whose endpoints, interfaces, and expected transaction semantics were known; these controls check method labels rather than permission-specific behavior. The first path exercised the OpenXR runtime-discovery ContentProvider, and the second exercised the cpm/IXrRuntimeServer endpoint using the version-matched mapping inventory. Table 3 summarizes the mapping scopes, and Table 5 distinguishes the evidence semantics.

The ContentProvider control produced the expected query interpretation in 5/5 repetitions. The controlled IXr path produced endpoint-confirmed createNode evidence in 4/5 repetitions; all five traces were valid, with one containing no createNode event. With the endpoint, descriptor, and mapping version matched, the procedure recovered the tested

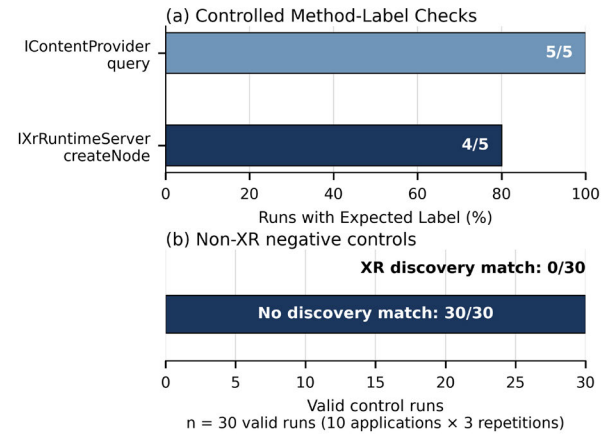


FIGURE 6. Controlled method-label checks and non-XR negative controls. The figure summarizes expected query and createNode labels observed in the controlled paths and runtime-discovery results for non-XR control runs.

createNode method, the only IXrRuntimeServer method exercised from the 117-entry inventory.

We also evaluated 10 non-XR applications over three repetitions each, yielding 30 valid control runs. No runtime-discovery match was observed in the 30 non-XR control runs.

D. OBSERVATION-WINDOW SENSITIVITY AND REPEATABILITY

To evaluate repeatability and observation-window sensitivity rather than interaction coverage, we fixed the same 51 manifest-declared runtime-observation pairs and repeated the measurements three times using both 3-second and 10-second windows. Each candidate-run was classified using the final evidence labels: XR runtime-discovery confirmed, no interface evidence, or invalid execution.

For the 3-second window, XR runtime-discovery evidence was confirmed for 40, 42, and 44 candidates across repetitions 1–3, respectively. The corresponding 10-second results were 39, 43, and 44 candidates. After recovery and retry handling, all six repetitions yielded accepted results for all 51 candidates.

At 3 seconds, 40 of the 47 candidates that showed runtime-discovery evidence in at least one repetition were observed in the first repetition, giving an 85.1% first-hit rate; 38/51 candidates retained the same label across all three repetitions, corresponding to 74.5% label stability; the 3/3, 2/3, 1/3, and 0/3 recurrence counts were 34, 11, 2, and 4. At 10 seconds, the corresponding first-hit rate was 39/46 (84.8%), and 42/51 candidates retained the same label, corresponding to 82.4% label stability, with corresponding recurrence counts of 37, 6, 3, and 5.

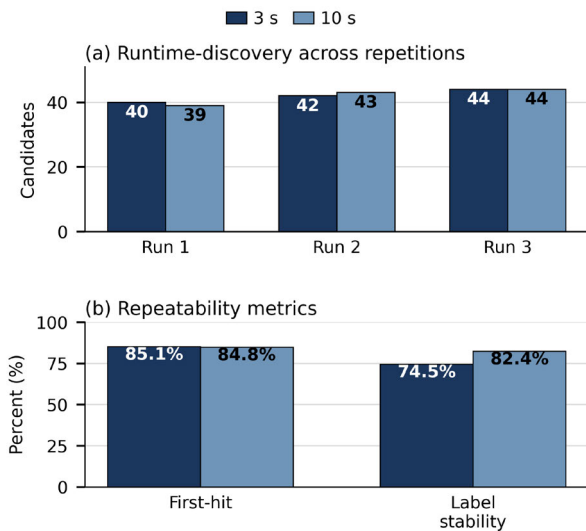
The 3-second and 10-second conditions produced comparable runtime-discovery counts; 37 candidates retained their recurrence-count class and 14 changed it between the conditions. In these repetitions, the 10-second window was associated with higher three-run label agreement than the

TABLE 5. Evidence semantics of the commercial and controlled conditions. The commercial condition associates a static candidate with a runtime-discovery event; the controls check only the listed method labels.

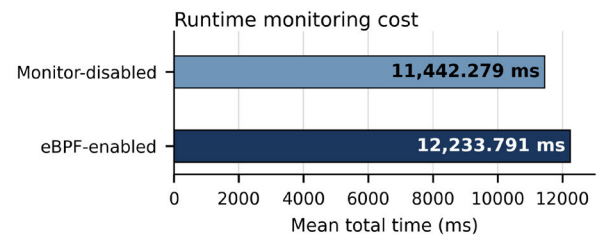
Condition	Static permission candidate	Observed endpoint / interface / method	Permission-specific evidence established?
Commercial applications	One of 51 manifest-declared pairs	OpenXR runtime-discovery receiver in Broker package context / <code>IContentProvider.query</code>	No; runtime-discovery association only
ContentProvider control	Not applicable (method control)	Controlled OpenXR ContentProvider / <code>IContentProvider.query</code>	No; method-label check only
IXr control	Not applicable (method control)	<code>cpm / IXrRuntimeServer.createNode</code>	No; method-label check only

TABLE 6. Candidates whose runtime-discovery recurrence count changed between the 3-second and 10-second observation windows. Recurrence denotes the number of discovery-confirmed runs among three repetitions.

Package	Permission	3 s	10 s
com.adobe.pulsar	HAND_TRACKING	3/3	2/3
com.algoriddim.djay_free	HAND_TRACKING	3/3	2/3
com.algoriddim.djay_free	SCENE_UNDERSTANDING_FINE	2/3	1/3
com.atlanticstudios.blackholes	EYE_TRACKING_COARSE	2/3	3/3
com.atlanticstudios.blackholes	EYE_TRACKING_FINE	2/3	1/3
com.atlanticstudios.blackholes	HAND_TRACKING	1/3	0/3
com.fitxr.boxvr	EYE_TRACKING_COARSE	2/3	3/3
com.fitxr.boxvr	EYE_TRACKING_FINE	2/3	3/3
com.fitxr.boxvr	HAND_TRACKING	2/3	3/3
com.fitxr.boxvr	SCENE_UNDERSTANDING_COARSE	2/3	3/3
com.google.android.apps.immersiveviewxr	EYE_TRACKING_COARSE	3/3	2/3
com.google.android.apps.immersiveviewxr	EYE_TRACKING_FINE	2/3	3/3
com.google.android.apps.immersiveviewxr	HAND_TRACKING	2/3	3/3
com.liteboxer.LBVR	SCENE_UNDERSTANDING_COARSE	3/3	2/3

**FIGURE 7. Primary 3-second evaluation and 10-second observation-window sensitivity. The figure compares run-wise runtime-discovery evidence and candidate-level repeatability metrics.**

3-second window (82.4% versus 74.5%). The 74.5% and 82.4% values quantify within-condition three-run label agreement, whereas 37/51 quantifies cross-window preservation of the discovery recurrence count. These measures therefore capture distinct aspects of repeatability.



Mean paired relative increase: 6.95% (n = 27 pairs)

Collector: 0.01428% CPU | 48.231 MB RSS | 20.92 events/s

FIGURE 8. Runtime monitoring cost comparison between monitor-disabled and eBPF-enabled execution across 27 paired repetitions using the same 10-second workflow. The measurements characterize analysis-time monitoring cost rather than production application overhead.

Table 6 lists the 14 candidates whose recurrence count changed between the two window conditions. Seven candidates increased and seven decreased in recurrence under the 10-second condition, while both conditions yielded 126 discovery-labeled candidate-runs out of 153. The longer window therefore did not systematically increase discovery recurrence. Ten of the 14 changes occurred in three application packages, indicating that the observed variability was concentrated in a subset of the evaluated applications rather than distributed uniformly across candidates.

E. RUNTIME MONITORING COST

To quantify runtime monitoring cost, we compared monitoring-disabled execution with eBPF-enabled observation over 27 paired repetitions using the same 10-second workflow. Candidates were ranked by Binder-event count; the first, middle, and last three formed the low-, medium-, and high-traffic subsets. Each subset comprised three candidates repeated three times (nine paired runs), and positional rather than fixed-threshold selection allowed the observed event ranges to overlap.

The mean paired relative total-time increase was 6.95%. The mean total time was 11,442.279 ms with monitoring disabled and 12,233.791 ms with eBPF enabled. The mean `am start -W WaitTime` was 1,070.704 ms without monitoring and 911.889 ms with monitoring. Across the paired runs, the collector used 0.01428% mean CPU and 48.231 MB mean RSS while recording an average of 219.1 Binder events, or 20.92 events per collector second. Across the low-, medium-, and high-traffic groups, mean collector CPU remained at or below 0.0287% and mean RSS ranged from 47.74 to 49.20 MB.

VIII. DISCUSSION

A. EFFECTIVENESS AND LIMITATIONS

eSPADE-XR is designed to cross-verify static XR permission candidates with interpretable runtime evidence. From 204 app-permission pairs, it selected the 51 manifest-declared pairs for runtime observation. The 3- and 10-second conditions produced comparable runtime-discovery counts, while 14 candidates changed recurrence-count class between conditions, with higher three-run label agreement at 10 seconds (82.4% versus 74.5%). The controlled checks observed the expected query label in 5/5 repetitions and the endpoint-confirmed `createNode` label in 4/5 repetitions; no runtime-discovery match was observed in 30 valid runs from 10 non-XR control applications. In the paired monitoring-cost experiment, eBPF-enabled observation increased mean total analysis time by 6.95% across 27 paired runs; mean collector CPU and RSS were 0.01428% and 48.231 MB, respectively.

The commercial evaluation uses the known OpenXR receiver context for runtime-discovery evidence and does not establish that the corresponding permission was exercised. The controlled evaluation separately checks the `createNode` interpretation using verified endpoint and descriptor identity as the only `IXrRuntimeServer` method exercised from the 117-entry mapping inventory. Permission-specific feature execution and downstream sensor-data flow remain outside the current scope.

The experiments were conducted on the Android Studio XR Headset Emulator, and therefore they do not observe physical sensors, OEM HAL implementations, kernel drivers, shared memory, or FMQ-based data paths that may appear on real XR devices. Binder IPC can serve as runtime-discovery or control-path evidence, but it does not capture the data

path through which high-rate sensing data may be delivered. In addition, AIDL transaction mapping may depend on the XR runtime implementation, Android build/API level, and OEM-specific interface artifacts. Therefore, a mapping table constructed for one environment cannot be assumed to generalize to all XR platforms, and emulator-observed paths cannot be assumed to occur unchanged on physical devices or OEM-specific implementations. The mapping is specific to the evaluated Android build and must be revalidated for other builds; interfaces without verifiable mapping artifacts remain uninterpreted. Future work should extend the evaluation to real XR devices and AR glasses and should refine mapping generation procedures to account for differences in OEM Broker implementations and AIDL mappings.

Because monitoring begins after target-PID acquisition, events occurring only during the initial cold-start phase may precede collector attachment and fall outside the current observation window.

The commercial dataset covers 31 collected Android XR applications, comprising 21 Made for XR and 10 XR Ready applications, and yields 34 normalized package-level analysis units, with package identifiers for all 34 units and `versionName/versionCode` metadata for 31. The dataset represents the collected Play Store XR sample used in this study rather than the full Android XR ecosystem; therefore, the commercial-app results should not be interpreted as population-wide prevalence estimates. As the Android XR market grows, the evaluation can be extended with broader inventories and additional metadata.

B. FUTURE WORK

Because eSPADE-XR collects eBPF-based Binder IPC metadata without directly modifying applications or the XR Runtime, it can observe runtime control paths in closed XR application environments. It also reduces the structural ambiguity of static analysis caused by Unity/IL2CPP, SDK boilerplate, and native libraries by redefining static analysis results as runtime evidence candidates rather than final judgments. However, applying eSPADE-XR in real deployment environments requires stable management of several factors, including XR Broker identification, target PID tracking, AIDL mapping table maintenance, and reproduction of application-specific execution conditions. Access to Binder tracepoints also requires a privileged or developer-enabled environment, which should be considered when deploying the collector on commercial devices.

The current targeted execution method creates observable runtime windows through candidate-guided execution. The 3- and 10-second conditions use this protocol to assess observation-window sensitivity and repeatability, not interaction coverage. It does not fully reproduce normal user flows or complex XR interaction sequences. The current runner does not implement a service-specific trigger; when direct component launch is unsuccessful, fallback execution may still provide app-level runtime evidence, which is interpreted at the candidate level rather than as evidence

of successful service execution. Therefore, eSPADE-XR should be interpreted as a candidate-driven framework that compares static candidates with observations collected in these windows, rather than as a complete UI exploration framework. Future research may combine eSPADE-XR with automated XR interaction generation or UI exploration techniques and include non-candidate component controls to observe candidate evidence under more diverse execution conditions.

IX. CONCLUSION

This paper addressed the difficulty of cross-verifying static XR permission candidates with observable runtime evidence in Android XR platforms, as well as the semantic interpretation gap introduced by XR Broker-based Binder IPC structures. To address these issues, eSPADE-XR redefines static analysis results as runtime evidence candidates rather than final detection results, and combines eBPF-based Binder IPC observation with receiver-context analysis and interface-aware, version-matched transaction mappings to analyze control paths and interpretable runtime evidence.

In the evaluation, eSPADE-XR classified 204 app-permission pairs across 34 normalized package-level units derived from 31 Android XR applications and selected the 51 manifest-declared pairs for the runtime-observation scope. Across the primary 3-second condition, XR runtime-discovery evidence was observed for 40, 42, and 44 candidates; the 10-second sensitivity condition produced comparable counts of 39, 43, and 44 candidates. The controlled checks observed the expected query label in 5/5 repetitions and the endpoint-confirmed createNode label—the only IXrRuntimeServer method exercised from the 117-entry inventory—in 4/5 repetitions, and no runtime-discovery match was observed in 30 valid runs from 10 non-XR control applications.

eSPADE-XR therefore supports candidate-level cross-verification of static XR permission candidates with interpretable runtime evidence in the evaluated emulator environment and Android build; it does not establish permission-specific feature or sensor use or equivalent paths on physical devices or OEM-specific implementations. Future work will extend the evaluation to real XR devices and AR glasses and will broaden the analysis scope by examining post-Binder data paths, including HAL, shared memory, and FMQ.

REFERENCES

- [1] R. Acheampong, T. C. Balan, D.-M. Popovici, E. Tuyishime, A. Rekeraho, and G. D. Voinea, "Balancing usability, user experience, security and privacy in XR systems: A multidimensional approach," *Int. J. Inf. Secur.*, vol. 24, no. 3, p. 112, Jun. 2025.
- [2] S. Mansour, V. Winterhalter, F. Alt, and V. Paneva, "A multi-layered privacy permission framework for extended reality," in *Proc. New Secur. Paradigms Workshop*, Aug. 2025, pp. 50–65.
- [3] V. Nair, W. Guo, J. Mattern, R. Wang, J. F. O'Brien, L. Rosenberg, and D. Song, "Unique identification of 50,000+ virtual reality users from head and hand motion data," in *Proc. 32nd USENIX Secur. Symp. (USENIX Secur. 23)*, 2023, pp. 895–910.
- [4] D. Cayir, A. Acar, R. Lazzeretti, M. Angelini, M. Conti, and S. Uluagac, "Augmenting security and privacy in the virtual realm: An analysis of extended reality devices," *IEEE Secur. Privacy*, vol. 22, no. 1, pp. 10–23, Jan. 2024.
- [5] R. Acheampong, D.-M. Popovici, T. C. Balan, A. Rekeraho, and I.-A. Oprea, "A cybersecurity risk assessment for enhanced security in virtual reality," *Information*, vol. 16, no. 6, p. 430, May 2025.
- [6] V. C. Nair, G. Munilla-Garrido, and D. Song, "Going incognito in the metaverse: Achieving theoretically optimal privacy-usability tradeoffs in VR," in *Proc. 36th Annu. ACM Symp. User Interface Softw. Technol.*, Oct. 2023, pp. 1–16.
- [7] A. Giarretta, "Security and privacy in virtual reality: A literature survey," *Virtual Reality*, vol. 29, no. 1, p. 10, Dec. 2024.
- [8] C. Warin, V. Pak, and D. Reinhardt, "Privacy perceptions across the XR spectrum: An extended reality cross-platform comparative analysis of a virtual house tour," *Proc. Privacy Enhancing Technol.*, vol. 2025, no. 1, pp. 150–168, Jan. 2025.
- [9] Android Developers. (2026). *Understand Permissions for XR*. Accessed: Jun. 16, 2026. [Online]. Available: <https://developer.android.com/develop/xr/permissions>
- [10] Android Developers. (2026). *Build With Supported OpenXR Extensions*. Accessed: Jun. 16, 2026. [Online]. Available: <https://developer.android.com/develop/xr/openxr/extensions>
- [11] R. Baalous, A. Althobaiti, D. Alyoubi, R. Alzahrani, and M. Aljohani, "Detecting the inconsistency between Android apps' data collection and Google play's data safety using static analysis," *Cybern. Inf. Technol.*, vol. 25, no. 1, pp. 110–125, Mar. 2025.
- [12] Z. Wu, H. Lee, and S. U. J. Lee, "An empirical study on the impact of permission smell in Android applications," *J. Korea Soc. Comput. Inf.*, vol. 26, no. 6, pp. 89–96, 2021.
- [13] C. Yue, K. Chen, Z. Guo, J. Dai, X. Sun, and Y. Yang, "What's done is not What's claimed: Detecting and interpreting inconsistencies in app behaviors," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2025.
- [14] Android Open Source Project. (2025). *Application Sandbox*. Accessed: Jun. 16, 2026. [Online]. Available: <https://source.android.com/docs/security/app-sandbox>
- [15] Khronos Group. (2021). *OpenXR 1.0 Specification, Version 1.0.34*. Accessed: Jun. 16, 2026. [Online]. Available: <https://registry.khronos.org/OpenXR/specs/1.0-khr/pdf/xrspec.pdf>
- [16] Android Open Source Project. (2025). *Hardware Abstraction Layer (HAL) Overview*. Accessed: Jun. 16, 2026. [Online]. Available: <https://source.android.com/docs/core/architecture/hal>
- [17] Android Developers. (2025). *Sensor (NDK): Direct Channel*. Accessed: Jun. 16, 2026. [Online]. Available: <https://developer.android.com/ndk/reference/group/sensor>
- [18] Android Open Source Project. (2025). *Fast Message Queue With AIDL*. Accessed: Jun. 16, 2026. [Online]. Available: <https://source.android.com/docs/core/architecture/aidl/fmq>
- [19] Android Open Source Project. (2025). *AIDL Overview*. Accessed: Jun. 16, 2026. [Online]. Available: <https://source.android.com/docs/core/architecture/aidl>
- [20] eBPF.io. (2025). *What is EBPF?*. Accessed: Jun. 16, 2026. [Online]. Available: <https://ebpf.io/what-is-ebpf/>
- [21] Android Open Source Project. (2025). *Extend the Kernel With EBPF*. Accessed: Jun. 16, 2026. [Online]. Available: <https://source.android.com/docs/core/architecture/kernel/bpf>
- [22] G. Munilla Garrido, V. Nair, and D. Song, "SoK: Data privacy in virtual reality," *Proc. Privacy Enhancing Technol.*, vol. 2024, no. 1, pp. 21–40, Jan. 2024.
- [23] K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. Lie, "PScout: Analyzing the Android permission specification," in *Proc. ACM Conf. Comput. Commun. Secur.*, Oct. 2012, pp. 217–228.
- [24] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Oceau, and P. McDaniel, "FlowDroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps," *ACM SIGPLAN Notices*, vol. 49, no. 6, pp. 259–269, Jun. 2014.
- [25] F. Wei, S. Roy, X. Ou, and Robby, "Amandroid: A precise and general inter-component data flow analysis framework for security vetting of Android apps," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Nov. 2014, pp. 1329–1341.

- [26] H. Guo, H.-N. Dai, X. Luo, Z. Zheng, G. Xu, and F. He, "An empirical study on Oculus virtual reality applications: Security and privacy perspectives," in *Proc. IEEE/ACM 46th Int. Conf. Softw. Eng.*, Apr. 2024, pp. 1–13.
- [27] C. Zuo and Z. Lin, "Playing without paying: Detecting vulnerable payment verification in native binaries of unity mobile games," in *Proc. 31st USENIX Secur. Symp. (USENIX Secur.)*, 2022, pp. 3093–3110.
- [28] D. Rodriguez, J. A. Calandrino, J. M. Del Alamo, and N. Sadeh, "Privacy settings of third-party libraries in Android apps: A study of Facebook SDKs," *Proc. Privacy Enhancing Technol.*, vol. 2025, no. 2, pp. 173–187, Apr. 2025.
- [29] Android Developers. (2025). *NdkBinder (FIRST_CALL_TRANSACTION / LAST_CALL_TRANSACTION)*. Accessed: Jun. 16, 2026. [Online]. Available: <https://developer.android.com/ndk/reference/group/ndk-binder>
- [30] Android Open Source Project. (2026). *Stable AIDL*. Accessed: Jun. 16, 2026. [Online]. Available: <https://source.android.com/docs/core/architecture/aidl/stable-aidl>
- [31] R. Trimananda, H. Le, H. Cui, J. T. Ho, A. Shuba, and A. Markopoulou, "OVRseen: Auditing network traffic and privacy policies in oculus VR," in *Proc. 31st USENIX Secur. Symp. (USENIX Secur. 22)*, 2022, pp. 3789–3806.
- [32] D. Suo, L. Xue, W. Huang, R. Tan, and G. Sun, "Assessing the capability of Android dynamic analysis tools to combat anti-runtime analysis techniques," *J. Syst. Softw.*, vol. 234, Apr. 2026, Art. no. 112747.
- [33] J. Y. Kim, C. Zuo, Y. Zhao, and Z. Lin, "AUTOVR: Automated UI exploration for detecting sensitive data flow exposures in virtual reality apps," in *Proc. 34th USENIX Secur. Symp. (USENIX Secur. 25)*, Aug. 2025, pp. 7937–7955.
- [34] Android Open Source Project. (2025). *Binder Overview*. Accessed: Jun. 16, 2026. [Online]. Available: <https://source.android.com/docs/core/architecture/ipc/binder-overview>



HYOJOONG JU is currently pursuing the bachelor's degree with the Department of Computer Science and Engineering, Incheon National University. His research interests include Android systems security, XR privacy, and runtime behavior analysis.



HYEONJUN PARK is currently pursuing the M.S. degree with the Department of Computer Science and Engineering, Incheon National University. His research interest includes automated policy generation for advanced cloud-native environments.



SEUNGSOO LEE (Associate Member, IEEE) received the B.S. degree in computer science from Soongsil University, South Korea, and the M.S. and Ph.D. degrees in information security from Korea Advanced Institute of Science and Technology (KAIST), South Korea, in 2020. He is currently an Associate Professor with the Department of Computer Science and Engineering, Incheon National University. His research interests include developing secure and robust cloud/networks systems against potential threats.

• • •