Intent-aware Fuzzing for Android Hardened Application

Seongyun Jeong
UNIST
Ulsan, Republic of Korea
jsy01311@unist.ac.kr

Minseong Choi
UNIST
Ulsan, Republic of Korea
liberty@unist.ac.kr

Haehyun Cho
Soongsil University
Seoul, Republic of Korea
haehyun@ssu.ac.kr

Seokwoo Choi
The Affiliated Institute of ETRI
Daejeon, Republic of Korea
seogu.choi@gmail.com

Hyungsub Kim
Indiana University Bloomington
Bloomington, Indiana, USA
hk145@iu.edu

Yuseok Jeon*
Korea University
Seoul, Republic of Korea
ys_jeon@korea.ac.kr

Abstract

The widespread adoption of app hardening techniques in Android applications makes it more challenging for current analysis techniques to analyze hardened apps, leading to limited analysis coverage. This limitation is mainly due to the difficulties in obtaining detailed information about Intents, which is essential for component communication and the execution of specific events in Android applications.

In this paper, we introduce eBPF-based AHA-Fuzz, the first intent-aware greybox fuzzing framework for Android hardened applications. AHA-Fuzz proposes a valid intent generator to create valid intent inputs that can trigger diverse Android app behaviors. To precisely evaluate the impact of these inputs, AHA-Fuzz presents a selective coverage feedback approach. Additionally, AHA-Fuzz introduces approaches for efficiently triggering hard-to-trigger bugs (*e.g.*, scheduled malware) and detecting information leaks in hardened applications. Our evaluation results demonstrate that AHA-Fuzz triggers 92.3% more intents 3.45× faster and executes 23.9% more methods than previous approaches. Additionally, AHA-Fuzz has discovered 47 previously unknown bugs that existing approaches cannot detect. The developers of Google, Firefox, and Facebook have acknowledged 6 out of 47 bugs, and have already fixed three of them.

CCS Concepts

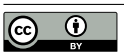
• **Security and privacy → Software and application security**.

Keywords

Android; Hardened Application; Intent; Fuzzing; Obfuscation; eBPF;

ACM Reference Format:
Seongyun Jeong, Minseong Choi, Haehyun Cho, Seokwoo Choi, Hyungsub Kim, and Yuseok Jeon. 2025. Intent-aware Fuzzing for Android Hardened Application. In *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security (CCS '25), October 13–17, 2025, Taipei, Taiwan.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3719027.3744858

*Corresponding author



CCS '25, Taipei, Taiwan

ACM ISBN 979-8-4007-1525-9/2025/10
https://doi.org/10.1145/3719027.3744858

1 Introduction

Android is a mobile operating system holding a 70.69% market share [8], with approximately 3.9 million Android applications (apps) available on the Google Play store [10]. Due to various potential security and privacy issues (*e.g.*, malware or privacy leaks) in Android apps, it is important to analyze Android apps to identify these issues in advance. However, many apps, whether legitimate or malicious, have increasingly adopted app hardening [26, 28, 45, 47, 49] techniques (*e.g.*, obfuscators, packers, protectors, or anti-analysis) to hinder such analyses. In particular, a preliminary analysis we performed discovers that obfuscation and packing app hardening techniques are widely applied to 51% of benign apps (selected from the top 100 most downloaded apps) and 99% of malicious apps (selected 100 apps with the highest download numbers, all of which are later identified as malicious).

The benign apps generally leverage app hardening techniques to protect against reverse engineering for unauthorized use (*e.g.*, source code leakage) and cracking. On the contrary, malicious apps generally use them to hide code related to malicious behaviors. With the widespread use of app hardening techniques, applying existing static analysis approaches to Android apps has become increasingly difficult. More specifically, the main logic of Android apps (*e.g.*, DEX code compiled from Java) is relatively easy to reverse engineer. Therefore, hardening techniques like obfuscation and packing are applied to prevent static analysis of these Android apps, and such approaches make static analysis unsuitable for hardened apps. On the other hand, dynamic analysis, which observes runtime behavior, can circumvent the impact of hardening techniques (*e.g.*, obfuscation or packing), making it a more practical option. Generally, fuzzing techniques [6, 36, 38, 57, 58] are widely used for dynamically analyzing Android apps.

When fuzzing Android apps, to maximize test coverage, an *intent* [24] is a key element used in Android development along with user interactions (*i.e.*, GUIs). In Android, an intent is a message object used for communication between Android app components, mainly used to transmit messages to start an activity, launch a service, or deliver a broadcast. Because Android apps are event-driven programs that interact with various components, intents that directly execute components and change program behavior are crucial to analyze Android apps. Especially, intents are key factors in triggering and analyzing malicious behaviors because 76.2% of them use intents to either hide malicious activities or initiate attacks [12].

Several existing intent fuzzing approaches [6, 13, 38, 40, 46, 57, 67] have been proposed to analyze Android apps. To perform efficient intent fuzzing, it is required to generate valid intents (*i.e.*, requiring key-value information), measure the impact of generated inputs (*e.g.*, with coverage feedback messages), and leverage appropriate bug-trigger and detection strategies (*e.g.*, altering scheduled events and detecting memory leaks). However, when targeting hardened apps, existing intent fuzzing approaches need to tackle the following three challenges.

The first challenge is extracting intent-related information, such as the key-value pairs (data added to the Intent object's extras field) to generate valid intents. For hardened apps, this information is difficult to extract through static analysis. Additionally, extracting such information during runtime is also challenging due to the limitations of existing dynamic analysis approaches [2, 60, 62, 64, 66, 73]. More specifically, these limitations include the detection of analysis attempts (*e.g.*, anti-debugging [9, 41, 61, 63, 68, 72]), the porting effort for each version (*e.g.*, system modification [60, 66]), and significant performance overhead (*e.g.*, DBI [62, 64]). Although Extended Berkeley Packet Filter (eBPF) [2, 73] provides a further reliable dynamic analysis environment with lower overhead and no need for an app or system modification, eBPF cannot directly identify the layout of objects containing intent-related information for generating valid intents.

The second challenge is the difficulty in applying coverage feedback to measure the impact of the intent fuzzer's generated inputs. More specifically, hardened apps use various obfuscation and packing approaches, making it challenging to precisely identify instrumentation targets during static analysis or the app's initial loading phase. As a result, traditional coverage instrumentation methods are hard to apply for these hardened apps. Even when instrumentation is successful, the Android app may invoke various events, such as GUI events or background events, alongside intent events triggered by the intent fuzzer. This causes noisy coverage feedback (*i.e.*, GUIs and background events-related coverage information) that can negatively impact the fuzzer's ability to measure the impact of generated intent inputs accurately.

Lastly, even if the fuzzer successfully reaches suspicious code, the challenge remains in triggering and detecting hidden bugs. For example, although fuzzer generated inputs may successfully reach a suspicious section of code, they may fail to trigger a bug if the event is scheduled under specific conditions. Note that Android apps frequently work by scheduling events through APIs (*e.g.*, 55.2% of malicious activities are triggered through scheduling [12]), meaning that certain events cannot be easily triggered unless specific conditions are met (*e.g.*, an event scheduled to trigger after three days [43]). Additionally, detecting triggered bugs such as information leaks is challenging with existing approaches [18, 59, 62, 64], as they require system modifications [18, 59] or impose high-performance overhead [62, 64] through DBI. In turn, the hardened apps require more practical and efficient bug detection methods.

In this paper, motivated by the lack of effective approaches to tackle the aforementioned challenges, we propose the Android hardened app fuzzer (AHA-Fuzz). To address the first challenge (*i.e.*, generating valid intent), AHA-Fuzz proposes a valid intent generator to efficiently obtain the -value pairs of intents that are actually registered and in use. For this, AHA-Fuzz first recovers the Java object layout to locate intent-related information and then extracts this information to generate valid intents.

To overcome the second challenge of applying coverage feedback messages, which are essential for measuring the impact of generated inputs, AHA-Fuzz proposes an eBPF-based coverage feedback approach. More specifically, this approach can effectively monitor the execution of methods across Android's different code patterns, including AOT-compiled, JIT-compiled, and interpreted code. Additionally, coverage feedback may include irrelevant information, such as GUI events, which can negatively impact intent fuzzing. To handle this issue, we propose a selective coverage feedback approach that detects and filters out such noise for more accurate feedback.

Successfully reaching problematic code is not enough; triggering and detecting hidden bugs, such as scheduled malware, in hardened apps remains difficult (*i.e.*, the third challenge). To address this last challenge, AHA-Fuzz adjusts scheduling APIs to quickly trigger malware scheduled for delayed execution. More specifically, AHA-Fuzz utilizes eBPF to hook scheduler-related functions and modify the associated scheduling times. Additionally, AHA-Fuzz introduces a lightweight detection method for identifying information leaks in hardened apps. This approach also utilizes eBPF to monitor APIs that handle sensitive information, ensuring minimal overhead and avoiding system or app modifications.

We evaluate AHA-Fuzz on Google Play store malware that uses complicated intents. Then, we confirm that AHA-Fuzz triggers 92.3% of intent usage patterns compared to existing approaches while AHA-Fuzz generates appropriate intents 3.45× as fast. Further, we measure AHA-Fuzz's code coverage on 40 top downloaded Android apps, AHA-Fuzz invokes 23.9% more methods than previous approaches. We also run AHA-Fuzz on 300 Android apps; thus, AHA-Fuzz discovers 47 previously unknown bugs including 26 crashes and 21 information leaks. Out of these 47 bugs, the developers of Google, Firefox, and Facebook have acknowledged two crashes and four information leaks, and have already fixed three of the four information leaks. It is worth noting that the previous works cannot find the 47 bugs discovered by AHA-Fuzz.

In summary, we make the following contributions:

- **Designing eBPF-Based Intent Fuzzer.** We develop an eBPF-based intent fuzzing environment that traces all Android events and extracts the intent parameters required to activate components in Android apps. AHA-Fuzz leverages this identified intent-related information as guidance for mutating inputs. To the best of our knowledge, AHA-Fuzz is the first work designing and implementing a greybox intent fuzzer.
- **Increasing Code Coverage.** We measure AHA-Fuzz's coverage on 14 malware and 40 selected Android apps, and confirm that AHA-Fuzz calls, on average, 92.3% more intents (3.45× faster) and executes 23.9% more methods compared to previous works.
- **Discovering Previously Unknown Bugs.** We run AHA-Fuzz on 300 Android apps. AHA-Fuzz discovers a total of 47 previously unknown bugs. Out of the 47 bugs, 26 bugs cause crashes and 21 bugs leads to information leaks. The developers of Google, Firefox, and Facebook have acknowledged two crashes and four information leaks, and have already fixed three of the four information leaks.

Table 1: Prevalence of App Hardening Techniques in Benign and Malicious Android Apps.

App	Obfuscation	Packing	Anti-Debugging	Anti-Emulator	At least one Hardening Tech
Benign	39	22	74	98	100
Malicious	22	94	9	33	100
Total	61	116	83	131	200

2 Background and Motivation

2.1 App Hardening Techniques

App hardening techniques, such as code obfuscation, are designed to protect an app from static and dynamic analysis [39]. Since Android apps are generally more susceptible to reverse engineering compared to other platforms, they frequently incorporate app hardening techniques to enhance security and protect against unauthorized analysis [16].

To examine the prevalence of app hardening techniques in Android apps, we analyze the top 100 most downloaded benign apps from Google Play and the 100 most downloaded malware samples from AndroZoo. To this end, we leverage the APKiD [55] to detect various app hardening techniques. As shown in Table 1, all benign and malicious apps utilize at least one app hardening technique.

App hardening techniques not only *hinder* the functionality of analysis tools, such as debuggers and decompilers, but can also render certain types of analysis *entirely impossible* by employing mechanisms like packing, code encryption, and runtime integrity checks [16]. Therefore, relying solely on a static analysis approach is insufficient for effectively analyzing real-world Android apps, as they often employ complex obfuscation and anti-analysis techniques. This necessitates the use of dynamic analysis methods such as intent fuzzing to complement static approaches and provide deeper insights into application behavior during runtime.

2.2 Intent Fuzzing

Intents are a fundamental component of app communication and behavior [24]. Android apps use intents for three different use cases: launching an activity (*e.g.*, screen), starting a service (*e.g.*, background task), and delivering a broadcast message (*e.g.*, system boot). Analyzing intents provides critical insights into an app's functionality, communication patterns, and potential security flaws. It's an essential part of understanding Android app behaviors. Additionally, various intent fuzzing efforts have been developed to generate and mutate intent data, ensuring the security, stability, and reliability of Android apps [6, 13, 38, 40, 46, 57, 67].

To receive intents from other apps, an app must declare the specific intents it wishes to handle in an intent filter. There are two ways to define the intent filter: defining it in the app's Manifest file and registering it by calling `registerReceiver()` at runtime (*i.e.*, dynamically registered intent). An intent includes standard fields such as *category* and *data*, but the most notable fields are *action* and *extras*. Unlike other standard intent fields, which can easily retrieve related information from the Manifest file, *actions* (especially in dynamically registered intent) and *extras* can only be extracted from the application code. The intent can include (1) an *action* field to request a specific action from another component (*e.g.*, opening a map) and (2) an *extras* field consisting of key-value pairs to carry additional information on the requested action (*e.g.*, `{BATTERY_LEVEL: 97}` to notify battery status). *Extras* play a crucial role in intent event analysis, as they are often used for communicating with backend servers (*e.g.*, Firebase) or delivering specific payloads in malware. *Actions* and *extras* do not have predefined values for each app and can only be identified through application code analysis.

Table 2: Comparison of Major Features of Existing Intent Fuzzing Approaches.

Approach	Input Coverage	Intent Generation Source	Feedback
MATE [6]	GUI & Intent	DEX intra-procedural analysis	None
Iccdroid [38], [13, 46]	Intent	DEX intra-procedural analysis	None
Sasnauskus et al [57]	Intent	DEX inter-procedural analysis	None
IntentFuzzer [67], [40]	Intent	Key Feedback & DEX analysis	None
AHA-Fuzz	GUI & Intent	Key-Value Feedback	Coverage-guided

Therefore, previous intent fuzzing approaches [6, 13, 38, 40, 46, 57, 67] mainly focus on analyzing code to extract intent-related details. Table 2 illustrates the main features of these existing intent fuzzing approaches. Some research works including MATE [6] perform the intra-procedural analysis [13, 38, 46] which maps intents by tracking all invocations of intent-related methods within the entry point methods of each component. While Sasnauskas et al. [57] propose an inter-procedural analysis-based approach that employs path-insensitive CFG analysis to obtain information regarding intents [65]. On the other hand, hybrid-based approaches including IntentFuzzer [67] leverage dynamic analysis techniques to extract the key of extras in runtime [40]. However, all the previous works are ineffective against app hardening techniques such as control-flow obfuscation because they rely on static analysis which can fail to identify the extracting operation of action strings, and keys or values of extras. Moreover, the previous works do not utilize the coverage feedback that provides real-time insights into the areas of code that have been executed during testing, enabling more effective and efficient analysis. We, thus, require an effective and efficient intent fuzzing solution that can overcome challenges in analyzing hardened apps, ensuring broader code coverage, and reducing redundant tests.

2.3 Dynamic Analysis Framework

To measure code coverage and monitor app behaviors during intent fuzzing, a suitable dynamic analysis framework is essential. Common approaches to building such a framework include (1) application modification, (2) Dynamic Binary Instrumentation (DBI), (3) system modification, and (4) eBPF.

Application modification-based approaches instrument the DEX code, which can be easily decompiled. Because this method does not require separate analysis tools at runtime, it is widely used in many fuzzing approaches for tracking code coverage [4, 52, 56]. However, this method is limited in handling dynamically loaded components that are difficult to instrument during static analysis. It, also, requires *repackaging* after instrumentation, which compromises the app's integrity and poses challenges when analyzing hardened apps with integrity checks.

On the other hand, the DBI-based approach instruments a target app at runtime, and thus, it can handle dynamically loaded code. However, the DBI-based approach can impose approximately ten times more overhead during app analysis, including code coverage measurement [64]. Moreover, the DBI-based approach cannot be effectively used to analyze apps that implement anti-debugging

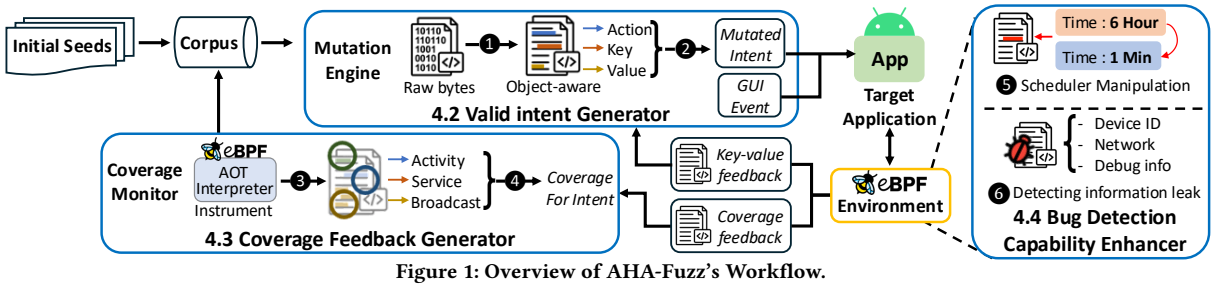


Figure 1: Overview of AHA-Fuzz's Workflow.

techniques because such techniques detect and interfere with instrumentation tools [9, 41, 61, 63, 68, 72].

The system modification approach leverages a customized version of the Android Open Source Project (AOSP). This customized system allows for more precise analysis (*e.g.*, instruction analysis) compared to application-level analysis methods. However, building a customized system requires significant engineering effort to understand and modify the Android Runtime (ART) source code [42, 54]. In addition, the huge manual engineering effort must be repeated each time a new version is released.

Lastly, the Extended Berkeley Packet Filter (eBPF)-based approach enables observation without compromising app integrity by setting tracing points with probes [2, 73]. However, because events can only be observed through predefined probes, the scope of events that can be monitored is inherently limited. For example, using a probe requires knowledge of the target code address. However, code executed by an interpreter or JIT, where the address is unknown at the time of setting tracing points, cannot be monitored. Moreover, eBPF can only access raw bytes in memory, making it difficult to observe Java-specific information.

Consequently, the current analysis frameworks face considerable challenges in collecting code coverage and app behaviors during the fuzzing of hardened apps, highlighting the need for an enhanced analysis framework to effectively handle such apps.

3 Threat Model

Our threat model assumes that both attackers and benign users can install malicious apps on the target victim devices. We also assume that several hardening techniques are applied to the malicious apps to prevent static or dynamic analysis. Specifically, we consider (*i*) runtime-based obfuscators such as packers that hinder static analysis, (*ii*) integrity checks that detect app tampering, and (*iii*) protectors that detect dynamic analysis environments. In addition, we assume that an attacker can directly trigger vulnerabilities (*e.g.*, crash or information leak) in victim apps by sending crafted intents. The primary goal of AHA-Fuzz is to analyze such malicious apps through fuzzing, even in hardening conditions.

4 Design

4.1 Overview of AHA-Fuzz

AHA-Fuzz internally incorporates GUI fuzzing because analyzing all behaviors of Android apps requires both intents and GUI events. More specifically, GUI fuzzing can trigger dynamically registered intents, while intent fuzzing can directly invoke components that are otherwise challenging to access via the normal GUI path (*e.g.*, activities visible only under certain conditions or settings screens launched exclusively through specific intents). By using these two approaches complementarily, the test coverage of Android apps can be significantly increased.

For GUI fuzzing, we leverage APE [36], the state-of-the-art GUI fuzzer that performs fuzzing solely on GUI elements not generally influenced by hardening techniques. To effectively balance GUI and intent inputs according to the characteristics of the target app, AHA-Fuzz dynamically adjusts the ratio between GUI and intent inputs using AHA-Fuzz's customized feedback messages, as discussed in Section 4.3.2. For intent fuzzing, we generate initial seeds by analyzing the Android manifest file, which defines all statically registered intent entry points and is not protected by hardening techniques. Additionally, we capture dynamically registered intent events (Section 4.2.2) by hooking related methods (*e.g.*, `registerReceiver()`). Note that the manifest file does not include information about intent extras. AHA-Fuzz alternates between generating GUI test inputs (using APE) and intent test inputs generated internally by our proposed intent fuzzer.

We note that AHA-Fuzz internally uses eBPF to create a reliable low-management and performance overhead analysis environment. Specifically, eBPF allows observation of system events at the kernel level without modifications to the app or system, enabling stable and accurate monitoring of events. Moreover, the overhead from eBPF—approximately 1.5 times [2, 50]—is considerably less than the roughly 10 times overhead [64] typically incurred by DBI-based solutions. However, eBPF-based app analysis introduces several limitations (*e.g.*, understanding Java object layouts and generating appropriate feedback messages for valid input generation). AHA-Fuzz addresses these limitations (detailed in Section 4.2 and Section 4.3) to provide stable and precise analysis environments.

The overall architecture of the eBPF-based AHA-Fuzz is illustrated in Figure 1. To generate diverse and valid intent inputs and increase bug detection capability, AHA-Fuzz includes the following three main components:

(i) Valid Intent Generator (Section 4.2): To generate valid intent, obtaining the key-value pair values of intent extras that are actually registered and in use is important. For this, AHA-Fuzz first recovers the Java object layout to locate intent-related information (Section 4.2.1) and then extracts this information to generate valid intents (Section 4.2.2).

(ii) Coverage Feedback Generator (Section 4.3): To guide the fuzzer in exploring deeper parts of the hardened app's code, it is essential to evaluate the impact of generated inputs using coverage feedback messages. For this, AHA-Fuzz introduces the first coverage feedback technique among intent fuzzers (Section 4.3.1). Additionally, it proposes a method to reduce

noise from observing non-intent-related events for more accurate coverage feedback (Section 4.3.2).

(iii) Bug Detection Capability Enhancer (Section 4.4): Even if the fuzzer successfully reaches problematic code, triggering and detecting hidden bugs remains a challenge. AHA-Fuzz proposes a technique to quickly trigger malware that is scheduled to execute after a long delay by adjusting scheduling APIs (Section 4.4.1). Additionally, AHA-Fuzz introduces a lightweight detection method to identify information leaks in hardened apps (Section 4.4.2).

4.2 Valid Intent Generator

For effective intent fuzzing, generating intents with valid key-value information of intent extras is important. This subsection describes how AHA-Fuzz obtains currently available key-value information and creates valid intents.

4.2.1 Recovering Java Object Layout. To create valid intents for fuzzing, required essential information (*i.e.*, key-value information of the intent presently in use and stored in memory) can be extracted from an allocated Java object. However, to accurately extract the required information, the Java object's layout must be recognized. For example, the `value` field, which is the specific value assigned to the intent key, exists within the Java object. To extract information from this `value` field, the offset within the Java object's memory layout needs to be identified. However, we cannot directly identify offset information because memory layout information is lost after AOT-compilation. The existing solution [64] for observing Java information from assembly involves manually analyzing each offset, but it cannot support all Java objects, as offsets continuously change with each version and device.

To address this issue, AHA-Fuzz proposes a method for reconstructing Java object layouts by cross-referencing DEX code with its corresponding AOT-compiled assembly code. During the AOT-compilation, field names and type information in the DEX code are optimized away or removed, making it difficult to determine the offset of specific fields by examining the assembly alone. However, we note that AOT assembly follows the JNI calling convention, which passes the second argument (*e.g.*, the `this` pointer) to `r1` and assigns subsequent arguments to `r2` and beyond. This convention allows us to distinguish pointers to Java objects and identify memory access instructions involving these pointers. Consequently, we consider only memory access instructions performed through these pointers as object field access instructions. By mapping these field access instructions in the DEX bytecode to actual operations (*e.g.*, memory load/store) in the AOT assembly, AHA-Fuzz enables the inference of field offsets and their relationships within Java objects.

Note that analyzing the layout of every object used across all Android methods is inefficient. Since the extracted layout information is used for extracting intent-related information (Section 4.2.2), key-value feedback generation (Section 4.2), selective coverage feedback generation (Section 4.3.2), and scheduler manipulation (Section 4.4.1), AHA-Fuzz only focuses on extracting the layouts of objects utilized in methods relevant to these four features. For example, AHA-Fuzz targets Intent-related classes, such as `Intent` and `IntentFilter`, and only analyzes the layouts of objects used by methods defined in these classes, which can be easily identified from DEX code.

Algorithm 1: The algorithm of Java object layout recovering

Input : F, T_c
F: a given OAT file contains DEX and assembly code;
T_c: a given target class;
S_p: a set of pointers pointing to target Java object;
$M_e \Leftarrow$ a map for storing fields with their exact offset;
$M_p \Leftarrow$ a map for storing fields with their possible offsets;
Output: Target classes' layout information

// **Step 1:** Extract field offset information in target instruction

```
1  forall class ∈ F do
2      if class.isClassIncluded(T_c) then
3          method ⇐ class.method;
4          forall inst ∈ method.DEX do
5              if is_Field_Access(inst) then
6                  method.field_update(inst.field);
7          S_p ⇐ getTargetPointer(method);
8          forall inst ∈ method.Asm do
9              TrackingPointer(inst, S_p);
10             if is_Field_Access(inst, S_p) then
11                 if Direct_Mapping(inst, method.field) then
12                     M_e[method.field] ⇐ inst.offset;
13                 else
14                     M_p[method.field].add(inst.offset);
```

// **Step 2:** Recursively eliminate possible offsets, find exact offset

```
15 offsets ⇐ ∅;
16 forall field ∈ M_e.field do
17     offsets.add(M_e[field]);
18 forall field ∈ M_p.field do
19     M_p[field] ⇐ M_p[field] \ offsets;
20     if len(M_p[field]) ≡ 1 then
21         offsets.add(M_p[field]);
22         M_e[field] ⇐ M_p[field];
23         goto line 20
24 return M_e, M_p
```

Algorithm 1 illustrates the DEX and assembly field access mapping process. First, AHA-Fuzz targets all methods of target classes (lines 1–3) and collects field access instructions from DEX bytecode (lines 4–6). AHA-Fuzz extracts pointer sets pointing to target Java objects in assembly code (line 7). Then, using alias analysis over the pointer sets extracted earlier (line 7), AHA-Fuzz identifies additional field access instructions in the assembly (lines 8–10). Direct mappings allow certain offsets to be immediately resolved (lines 10–12), while other cases (*e.g.*, related to branch conditions) are saved for later resolution (lines 13–14). In the next step, AHA-Fuzz performs additional mapping of unresolved offsets for each class. First, AHA-Fuzz initialize the offset set of the target class (lines 16-17) using directly mapped values (lines 11-12). AHA-Fuzz then iteratively refines the set by removing any offset from M_p that also exists in `offsets` (line 19). If a possible offset set contains only one value (line 20), it becomes an exact offset and is updated recursively (lines 21–23). Consequently, AHA-Fuzz returns the exact offsets

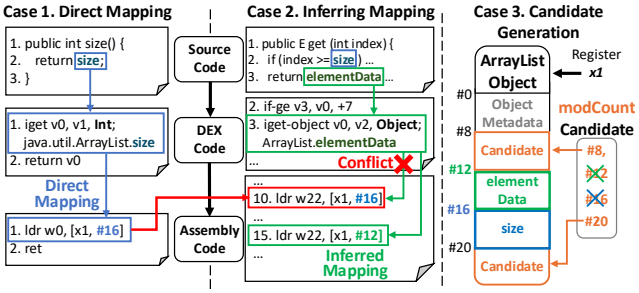


Figure 2: Object Layout Recover Examples in `java.util.ArrayList`.

and the possible offset candidates for still unresolved fields (line 24).

Figure 2 shows examples of how AHA-Fuzz identifies object layouts in `java.util.ArrayList`. In Case 1 (left of Figure 2), the `size` function is a simple getter that returns the size field of an ArrayList object. Because it only includes a load operation, we can directly map the field access in the DEX code (line 1, `iget Int; ArrayList.size`) to the corresponding assembly instruction (line 1, `ldr w0, [x1, #16]`). Consequently, we know the action field is at offset 16.

In contrast, the `get` function in Case 2 (middle of Figure 2) includes branch instructions, making it more complex to establish a direct mapping between DEX and assembly. Because AOT-compiled assembly follows the JNI calling convention, we can determine which pointer is stored in each register during the function prologue. Based on this information, path-insensitive pointer tracking allows us to identify potential field access instructions. For example, lines 10 and 15 in the `get` assembly indicate access to the `elementData` field of the ArrayList. From this, we can infer that `elementData` could be at offset 12 or 16. Additionally, since Case 1 (*i.e.*, `size`) already confirms that offset 16 corresponds to the size field, we infer that the `elementData` field must be at offset 12.

However, the `modCount` field in Case 3 (right of Figure 2) is not covered by direct mapping (Case 1) and inferring mapping (Case 2), where precise offset inference is not possible. AHA-Fuzz handles this unresolved field by continuously selecting its offset at random from the candidate set during fuzzing. For example, `modCount` has possible offsets of 8 or 20, which are not resolved in the direct mapping or inferred mapping steps. During fuzzing, AHA-Fuzz randomly selects an offset from the candidate set for each unresolved field to simulate various object layouts.

We note that recovering Java object layouts is challenging when direct and inference mapping do not occur frequently enough. However, we find that sufficient mapping occurs in most classes. This is mainly because most classes contain getter/setter methods, which introduce conflicts and enable offset retrieval without explicit inference. As shown in Table 3, according to our analysis on all 2,541 classes within the Android framework module that define Intent-related classes, we find that 42.2% of cases involve direct mapping (Case 1), generating conflict. Consequently, this allowed us to accurately infer offsets for another 40.6% of cases (Case 2). Relatively, only a small portion—17.2% of fields—cannot be precisely resolved to a fixed offset. However, even for these, AHA-Fuzz continues fuzzing by randomly selecting offsets from the candidate set to simulate various potential layouts of the intent object.

Table 3: Java object layout recovery result. Cases 1, 2, and 3 correspond to direct mapping, inferring mapping, and candidate generation, respectively.

Module name	Classes	Fields	Case 1 (%)	Case 2 (%)	Case 3 (%)
content	982	1,729	669 **(38.7%)**	681 **(39.4%)**	379 **(21.9%)**
app	1,559	3,465	1,523 **(44.0%)**	1,428 **(41.2%)**	514 **(14.8%)**
Total	2,541	5,194	2,192 **(42.2%)**	2,109 **(40.6%)**	893 **(17.2%)**

4.2.2 Key-value Feedback Generator. Section 4.2.1 allows AHA-Fuzz to recover the Java object layout. Based on this layout information, this section explains how to extract intent-related information stored in objects. Unlike C/C++, Java inherently manages elements such as strings ("msg") as objects. Therefore, understanding the layout of objects (*e.g.*, strings) that store frequently used intent key-value pairs is essential for accurate extraction. Using the key and value extraction methods introduced in this section, these extracted values are sent to the fuzzer as key-value feedback messages (not coverage feedback). These messages are utilized to construct valid intents.

Extracting Intent Key Set. To extract the key used in the extras field of intents, we utilize the following two patterns, as in previous research [40, 67]. First, since the parameter of the intent-related extra getter functions is the key, hooking and analyzing these functions allows us to obtain the currently used keys. For this, AHA-Fuzz hooks all related getter functions using eBPF. For example, encountering `getStringExtra("format")` indicates that `"format"` is currently used as a key for a particular intent.

The second approach for retrieving currently used keys is to hook Bundle-related functions, such as `getChar()`. This is because, the extras field is internally implemented as an Android Bundle [21], as a Bundle is essentially a map structure used by Android for storing key-value pairs. By tracking these Bundle-related functions, AHA-Fuzz can additionally extract currently used keys.

Extracting Intent Value Set. Finding the value corresponding to a key is important for testing an app's diverse and deep code. For example, Android apps can change execution flow depending on the value in the intent parameter (*e.g.*, line 4 in Figure 3). However, unlike the method of observing keys from intents, there are no predefined patterns or APIs to retrieve values from intents.

To address this challenge, we manually analyze various key-value code usage patterns on a total of 405 apps, including all 105 malware samples summarized by Cao et al. [12] and the top 300 most downloaded benign apps from the Google Play Store as of October 1, 2024. Our analysis find that, in most cases, extras values are used in comparison operations, which then trigger the execution of additional code or events. More specifically, all extra values used in the malware samples are detected by checking comparison operations. In the 300 most downloaded apps, we identify 6,106 unique intents using String extras, 67% (4,076 cases) of which are detectable through comparison operations. The remaining 33% are used purely for data transmission, not directly contributing to new code/event execution. Consequently, we find patterns that most values are checked through comparison operations (*e.g.*, `equals()` or `contains()`). Therefore, AHA-Fuzz hooks comparison operations through eBPF, extracting values used in comparisons as candidate values, and sends them to the fuzzer through key-value feedback messages.

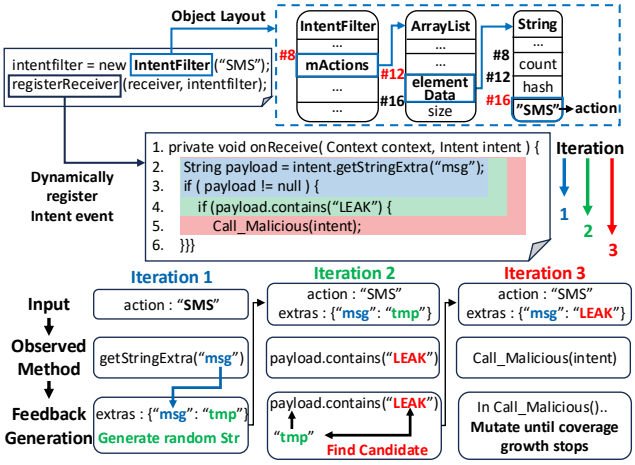


Figure 3: AHA-Fuzz's Key-Value Information Extraction Examples.

Figure 3 shows how AHA-Fuzz extracts key and value set through API hooking and key-value feedback. AHA-Fuzz starts fuzzing with an intent (*e.g.*, "SMS") as an initial seed that does not include key-value information. Note that AHA-Fuzz obtains the initial seed by analyzing the object layout of `IntentFilter`, which is used to register the intent event dynamically. Since AHA-Fuzz hooks all extra-related getter functions, including `getStringExtra()`, to extract key values, AHA-Fuzz can capture the "msg" key from the getStringExtra() function (line 2). This allows AHA-Fuzz to create a key-value feedback message that specifies the intent parameter and requires "msg" as a key. However, `getStringExtra()` returns `null` at line 2 because the initial intent does not have a key (*e.g.*, "msg").

Next, AHA-Fuzz sends a new intent that has "msg" as the key and a random string as the value. Thus, the executed intent triggers lines 2 – 4. In this second fuzzing attempt, AHA-Fuzz fails to visit line 5 because this basic block requires a specific value (*i.e.*, "LEAK") in the intent parameter. Since AHA-Fuzz hooks comparison-related APIs including `contains()`, AHA-Fuzz can obtain an additional candidate value "LEAK". In the next fuzzing attempts, AHA-Fuzz eventually visits line 5 utilizing an intent that includes the needed key-value information.

Mutation Strategy. AHA-Fuzz applies different mutation strategies to the values of extras, depending on their type: primitive, string, and Java object types (*e.g.*, `Binder`), which we refer to as special types. For Java primitive types, AHA-Fuzz utilizes AFL-style mutations, including random bit flips. String values are mutated using our key-value feedback (Section 4.2.2) algorithm. Mutating special types is challenging because their object layouts differ across apps. Currently, AHA-Fuzz supports special type mutation only through constructor invocation, and we leave this implementation improvement (*e.g.*, recovering specific special type layout using our Java object layout recovery Algorithm 1) as future work (see Section 8). We also note that string objects are more commonly used as values, whereas special types are relatively rare. Table 4 shows the analysis of extras value types used in the 300 most downloaded apps from Google Play. According to this analysis, string objects account for 49.5% and primitive types for 44.2%, while special types comprise only 6.5% of all extra values.

Table 4: Type ratio analysis of extras values used in 300 most downloaded apps.

Type category (#)	Mutation Strategy	# of unique extras values (%)
Primitive (8)	AFL-based	9,317 **(44.2%)**
String (1)	Key-Value feedback	10,395 **(49.3%)**
Special (5)	Invoke constructor	1,369 **(6.5%)**
Total (14)	-	21,081 **(100%)**

4.3 Coverage Feedback Generator

Although all existing intent fuzzers are blackbox-based fuzzers, coverage feedback on how many/new methods are triggered through generated intents enables more efficient input generation. For example, Figure 3 shows that key-value feedback can further trigger possible execution flow paths, but it requires several iteration steps. Without coverage feedback, the fuzzer cannot estimate whether further iteration are necessary or if the current input is sufficient. For this reason, AHA-Fuzz, as the first intent fuzzer, introduces coverage feedback to evaluate the quality of each generated intent. Additionally, coverage feedback may include irrelevant information, such as GUI events, which can negatively impact intent fuzzing. To address this issue, we propose a selective coverage feedback approach that detects and ignores such noise, enabling more accurate utilization of feedback messages.

4.3.1 Coverage Instrumentation. AHA-Fuzz, based on eBPF, utilizes method-level coverage feedback. eBPF observes methods through probes that require the target code address, which can be obtained from AOT-compiled code. Therefore, using eBPF, we can monitor which methods are executed within AOT-compiled code due to intents. However, it is difficult to obtain address information for eBPF hooking in JIT-compiled code and the interpreter. For this, we disable JIT-compiled code execution (will be handled as interpreter code) through simple option change, without modifying the system, but still cannot disable interpreter code.

Note that interpreter code is not compiled and is executed at runtime via the interpreter, making it challenging to attach eBPF probes. To address this, we attach probes to the interpreter engine that executes the interpreter code to observe its execution using eBPF. For this, we analyze the interpreter engine in ART and identify three general interpreter execution patterns as follows. First, each entry point of interpreter execution requires the following parameters to obtain the program's current state: `ArtMethod`, `Thread`, and stack frame objects. Second, ART updates the 'hotness' counter, which is used in JIT-compiled code, before executing interpreter code. Third, ART uses architecture-specific assembly code within its core interpreter mechanism to optimize performance.

Based on these observations, we leverage CodeQL [20] to identify the entry point of the interpreter. Note that with CodeQL, code (*e.g.*, Android code) is transformed into a database, enabling in-depth analysis through custom queries (*e.g.*, three defined patterns). Additionally, to evaluate whether the method of identifying interpreter entry points using CodeQL and the three identified patterns is valid across different Android versions, we test Android versions from the past five years (*i.e.*, since 2019) and find that all possible interpreter entry points are identified through our three patterns.

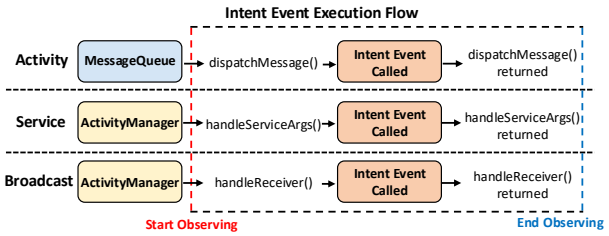


Figure 4: Overview of Intent Event Execution Flows.

4.3.2 Selective Coverage Feedback. Coverage feedback can be generated by monitoring all methods; however, observing every method may lead to noisy feedback from events unrelated to intents, negatively impacting intent fuzzing. For example, specific Android events include GUI events and background events that are executed by service components, and these unrelated events can also generate coverage feedback that negatively affects intent fuzzing.

To address this problem, one is willing to wait until the app is idle after sending the intent without delivering any events. Yet, this method is not only time-consuming but also, given the event-driven nature of Android apps, background tasks may cause the apps to exit this idle state unexpectedly. Therefore, we need to distinguish between events caused by AHA-Fuzz's intent fuzzing and other Android events caused by GUI fuzzing.

Analyzing Events Invoked by Intents. To distinguish events generated by intents, understanding the execution path of events via intents in Android is important. As shown in Figure 4, intents can be delivered to three main components: Activities, Services, and Broadcast Receivers. These intents pass through the `MessageQueue` (for Activities) or the `ActivityManager` (for Services and Broadcast Receivers) and corresponding events are executed via predefined methods (*e.g.*, dispatchMessage() for Activities). Based on these patterns, we can track the start (*e.g.*, `dispatchMessage()`) and end (*e.g.*, `dispatchMessage() returned`) of these methods using eBPF, focusing only on events occurring during the method's execution. This helps mitigate the noise issue described earlier. Furthermore, we validate the consistency of these patterns across Android version updates by formalizing them and checking them using CodeQL. Consequently, we find that this pattern has remained consistent over the past five years of Android version updates (*i.e.*, since 2019), ensuring the ongoing effectiveness of this approach.

Execution Balancing Between GUI and Intent Fuzzers. By identifying events triggered by intents and further distinguishing GUI events occurring on the main UI thread, the impact of inputs generated by each GUI and intent fuzzer can be evaluated. Initially, the GUI and intent fuzzers are executed at an equal 1:1 ratio to ensure fairness. However, AHA-Fuzz dynamically adjusts the execution ratio of each fuzzer based on the observed impact of their generated inputs. For example, if an app generates a higher proportion of events through intents compared to GUI interactions, the execution ratio of the intent fuzzer is increased accordingly.

4.4 Bug Detection Capability Enhancer

Although AHA-Fuzz generates various intents using AHA-Fuzz's key-values and coverage feedback, it may still miss some bugs triggered under specific conditions (*e.g.*, scheduled malware) or fail to detect triggered bugs (*e.g.*, information leak due to performance overhead). By addressing these primary challenges, such as detecting scheduled malware and efficiently identifying information leaks, we further enhance AHA-Fuzz's bug detection capability.

4.4.1 Scheduler Manipulation. Scheduling events in dynamic analysis reduces fuzzing effectiveness. For example, an Android app can execute code at a point in time one day after installation through a scheduling event. Additionally, some malware leverages the scheduler to enable periodic activation and deactivation, helping it evade detection. More specifically, by avoiding continuous activation, the malware only operates when necessary, reducing its chances of being detected. A straightforward solution is to modify the application or system code to fast-forward the scheduling time, but identifying scheduling time under hardening techniques is challenging and can cause system instability [11].

To address these issues, we utilize eBPF to hook scheduler-related functions and adjust the associated scheduling times. More specifically, we manipulate the scheduling event to occur quickly by overwriting the scheduling time stored in memory (*e.g.*, adjusting 12 hours to 10 minutes). In particular, we focus on a scheduling event registered by Android framework APIs (*e.g.*, `AlarmManager`, `JobScheduler`, or `WorkManager`). For scheduling events requiring long waiting time (*e.g.*, several hours), we leverage a helper function '`bpf_probe_write_user()`' in eBPF to decrease the waiting time stored in user space memory. For this, we insert an eBPF program into the return statement of the scheduling time setter method, enabling manipulation of the scheduling time for faster invocation. Note that scheduling APIs are well-documented [27], and we can retrieve the value of the scheduling time value as described in Section 4.2.1.

4.4.2 Detecting Information Leak. Both benign and malware apps can leak users' sensitive usage data (*e.g.*, locations or contacts) without informing the user or obtaining consent. To detect such information leaks, existing approaches [3, 7, 18, 59, 62, 64] conduct static/dynamic taint analysis to track source APIs (*e.g.*, return sensitive data) to sink APIs (*e.g.*, potential leak sites). However, existing approaches are not directly applicable to hardened apps. More specifically, propagating taint at the instruction level during runtime requires modifying the application to statically instrument the app code [3, 7, 59], which not only necessitates bypassing integrity checks but also provides limited support for dynamically loaded code, as discussed in Section 2.3. Moreover, starting with Android 7, the presence of diverse execution models—AOT-compiled, JIT-compiled, and interpreted code [33]—has limited the effectiveness of existing taint analysis approaches that rely on system modification [18] or DBI [62, 64], as these approaches typically support only a subset of execution modes (*e.g.*, DBI-based tool named Malton [64] can track taint only in AOT-compiled code).

To address these issues, AHA-Fuzz conducts a lightweight method-level taint propagation analysis instead of instruction-level taint propagation. As commonly mentioned in existing information leak detection approaches [18, 59, 62, 64], sensitive information is typically stored and managed as primitive types, such as int or String objects. For example, important network information (*e.g.*, SSID) to be protected is generally stored as a String object. To monitor these sensitive information flows, AHA-Fuzz leverages eBPF to monitor APIs that process these primitive values, such as methods in the

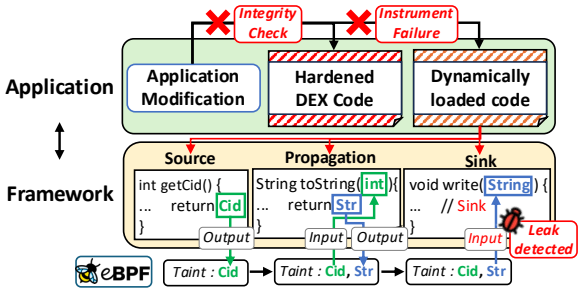


Figure 5: Overview of method-level taint propagation.

Java standard library. Specifically, by tracking sensitive primitive values as they pass through API parameters (inputs) and return values (outputs), AHA-Fuzz can detect potential information leaks.

Figure 5 shows how taint propagation is performed at the method-level. Method-level propagation tracks primitive type values and string values passed through framework APIs, enabling the tracing of data flows from sources (`getCid`) to sinks (`write`). AHA-Fuzz utilizes eBPF to hook each API call (`toString`), including both its entry and return points, to track input parameters and output return values. Consequently, method-level taint analysis can detect information leaks when tainted values reach sink APIs—even for dynamically loaded code.

Furthermore, we identify additional possible source APIs to detect information leaks in recent Android apps. To detect information leaks through taint analysis, recent research [14, 37, 69, 71] mainly relies on the source and sink API set defined by FlowDroid [5]. However, we observe that FlowDroid's API set does not include some APIs introduced in the last five years of Android releases. To detect additional information leaks in recent apps, we manually review the official Android documentation [31, 32] to identify newly introduced APIs. We also analyze the official documentation [34, 35], particularly discussions of potential security and privacy issues, to identify additional possible information leak attack scenarios. As a result, we extend the original FlowDroid API set by defining 14 additional source APIs.

5 Implementation

We adapt the BPF Compiler Collection (BCC) framework [44] to create an eBPF analysis environment. We also integrate AHA-Fuzz with an existing GUI-based fuzzing framework [36] to compare its effectiveness. Our implementation consists of Python for the BCC framework, Java for intent fuzzing, and C for eBPF programs. The total lines of code (LoC) are around 3,000.

eBPF Environment. The eBPF feature supported by the Linux kernel can be used in Android kernels based on Linux systems without any modifications. However, it does not provide a user-friendly environment because most of the documentation for the eBPF toolchain assumes standard Linux instead of Android. We utilize scripts provided by the Extended Android Tools [19] to leverage the BCC framework [44], through cross-compilation using the Android native development kit [25].

To observe AOT-compiled code, we extract method addresses from OAT files using `oatdump` and hook them in the Zygote process to monitor all subsequent app processes, similar to BPFroid [2]. For interpreter execution, we hook the interpreter entry points identified in Section 4.3.1 and extract the DEX method index contained in the `ArtMethod` structure through eBPF to monitor which method is called. To monitor only the targeted app's calls, we filter each app process through a unique UID using eBPF.

Fuzzing Environment. We note that AHA-Fuzz performs GUI fuzzing and intent fuzzing simultaneously. For GUI fuzzing, we select APE [36] because it shows the best performance and is implemented on top of Android's default GUI fuzzer (Monkey [30]). We implement AHA-Fuzz's intent fuzzing on top of APE, and it operates separately from the APE fuzzer. Thus, AHA-Fuzz's intent fuzzing can run with other GUI fuzzers.

6 Evaluation

We evaluate AHA-Fuzz's effectiveness in analyzing hardened apps. Specifically, we present our evaluation results by addressing the following research questions.

- **RQ1** Is AHA-Fuzz capable of creating valid intents (*i.e.*, an intent requiring specific extras) called by malware? (Section 6.2)
- **RQ2** How does AHA-Fuzz's fuzzing algorithm (key-value feedback and selective coverage feedback) affect the performance of the fuzzing? (Section 6.3)
- **RQ3** Does AHA-Fuzz guarantee increased coverage compared to previous works? (Section 6.4)
- **RQ4** Can AHA-Fuzz discover previously unknown bugs that previous works cannot find? (Section 6.5)

6.1 Evaluation Setup

We execute AHA-Fuzz on a machine running 64-bit MacOS Sonoma 14.4.1 (23E224) with a 24-core CPU (Apple M2 Ultra) and 192 GB of RAM. We configure each emulator using Android 13 with 2 cores and 8 GB of RAM.

Ground-truth Malware Dataset. To evaluate the performance of intent fuzzers, measuring the number of triggered malicious behaviors is more appropriate than simply counting code coverage, given that 76.2% of malware uses intents to hide malicious behavior or initiate attacks [12]. Therefore, we use the number of triggered malicious behaviors in malware as the metric for evaluating fuzzers. We also use the Google Play store malware dataset provided by Cao et al. [12]. In particular, this dataset identifies 105 different malware families and creates one report per family. We thus select a total of the 105 malware that has the report from the dataset. The report describes preconditions for triggering malicious behaviors. For instance, preconditions include payload (*e.g.*, a key-value pair {`status: StartAdService`}) for intent events or scheduling events that malware abuse. We use such events required to trigger malicious behaviors to evaluate whether AHA-Fuzz can trigger them. Furthermore, among the Android malware, we select 14 out of the 105 apps satisfying the following criteria:

(1) The malware exploits an intent that requires key-value pairs and/or a scheduling event to execute malicious behaviors.
(2) The malware can be run on a 64-bit ARM Android machine.
(3) We can reproduce malicious behaviors reported in study [12] (*e.g.*, connecting command and control servers).

Table 5: Intent usage patterns on 14 malicious apps. 'Improved' column denotes improvements achieved by AHA-Fuzz in the terms of patterns and times to find them, compared with the others. ✓denotes patterns invoked by AHA-Fuzz. ✗expresses that AHA-Fuzz fails to invoke the patterns. 'Time' represents improved time to invoke a pattern compared with the most fast one that invokes the same pattern.

Family	Patterns	APE (sec)	MATE (sec)	ICCBot (sec)	IntentFuzzer (sec)	AHA-Fuzz (sec)	Improved Patterns	Improved Time (X)	Pattern Details
AceCard	M	-	235	108	115	88	-	1.23×	E : **SMS**, **Payload**
	E	-	-	-	-	89	✓	-	
AgentBKY	E	-	-	-	-	30	✓	-	E : **SMS**
	E	-	-	121	136	89	-	1.36×	E : Google Firebase
Bahamut	M	-	1,251	647	829	38	-	17.03×	M : Multiple Entrypoints
	E	-	-	-	-	44	✓	-	E : **SMS**
ClickerGenG	M	-	283	171	140	74	-	1.89×	S : Run After 6 hours
	S	-	-	-	-	741	✓	-	
HiddenAdOS	M	-	643	322	464	86	-	3.74×	E : Google Firebase
	E	-	-	279	501	172	-	1.61×	
IndexY	E	-	-	-	-	55	✓	-	E : **SMS**, Multiple Entrypoints
ProjectSpyHRX	M	-	192	44	50	35	-	1.26×	E : **SMS**
	E	-	-	-	-	30	✓	-	E : Malware developer defined
	E	-	-	-	-	-	✗	-	
Reputation1_2019	M	-	281	141	101	32	-	3.15×	E : Dynamically registered, developer defined
	M	-	306	87	128	42	-	3.05×	
	E	-	-	201	133	43	-	3.09×	
SMSAndroidOSWesp	E	-	-	-	-	52	✓	-	E : **SMS**, **Payload**
Solid	M	-	174	61	55	50	-	1.1×	E : **Payload**, Google Firebase
	E	-	-	-	-	393	✓	-	
Sonyvpay	M	-	420	112	128	33	-	3.40×	E : **SMS**, **Payload**
	E	-	-	-	-	40	✓	-	
SpyBankerHU	M	-	301	104	114	31	-	3.35×	E : **SMS**
	E	-	-	-	-	43	✓	-	
TrojanDropperAgentCIQ	S	-	-	-	-	1,394	✓	-	S : Run after 24 hours
Vilny	M	-	362	98	109	32	-	3.06×	
	M	-	265	104	99	33	-	-	S : Run after 26 hours
	S	-	-	-	-	156	✓	-	
Total (%)		0/28 (0.0%)	11/28 (39.3%)	15/28 (53.4%)	15/28 (53.4%)	27/28 (96.4%)	92.3%	3.45×	

We confirm that: (*i*) 45 out of 105 apps require key-value pairs, and/or abuse the scheduling API; (*ii*) 38 out of 45 apps are compatible with 64-bit ARM Android machines; and (*iii*) we successfully reproduced malware behaviors in 14 out of these 38 apps.

Benign App Dataset. We evaluate AHA-Fuzz's performance on benign apps and find previously unknown bugs that prior works are hard to find. To this end, we collected the most downloaded 300 apps from the Google Play Store on October 1, 2024.

Malware Dataset. This dataset is intended to evaluate AHA-Fuzz's performance on real-world Android malware. Since finding malware directly from the Google Play Store is challenging, among the 2,230 malware samples available in AndroZoo from January 1, 2023 to August 1, 2024, we identify the top 20 most downloaded apps on the Google Play, which are also uploaded there, as our target apps.

Evaluation Baselines. To evaluate the effectiveness of AHA-Fuzz's fuzzing algorithms, we create the following four different baselines and six variant modes:

(1) **APE** [36]: It runs only GUI fuzzing. Hence, it does not perform intent fuzzing. We use APE's default configurations.
(2) **AHA-Fuzz**: As a default mode, we use AHA-Fuzz with APE.
(3) **AHA-KV**: It is a variant version of AHA-Fuzz. It only uses the key-value feedback (Section 4.2) from AHA-Fuzz's algorithm.
(4) **AHA-CV**: It only leverages the coverage (Section 4.3.1) and key-value feedback (Section 4.2) from AHA-Fuzz's algorithm.
(5) **MATE** [6]: It performs GUI fuzzing as well as intent fuzzing based on intra-procedural static analysis results. We note that MATE requires modifications of apps to enable the debug mode.
(6) **MATE-AHA**: It is a variant version of MATE that uses MATE's GUI fuzzing and AHA-Fuzz's intent fuzzing algorithm.
(7) **ICCBot**: It is an intent fuzzer based on inter-procedural analysis. We note that Sasnauskas et al. [57] proposed an intent fuzzing based on inter-procedural static analysis. However the source code is no longer available. We, thus, implemented inter-procedural intent fuzzer with ICCBot [65] that leverages key-value pairs extracted by using inter-procedural static analysis.
(8) **ICCBot-AHA**: It does not perform the inter-procedural static analysis. Also, it uses AHA-Fuzz's intent fuzzing algorithm.
(9) **IntentFuzzer** [67]: It is an intent fuzzing approach based on a hybrid method. IntentFuzzer leverages runtime feedback to obtain the `key` and intra-procedural static analysis to retrieve the corresponding `value`. IntentFuzzer performs fuzzing only on service and broadcast components. Because it is a closed-source project, we implemented it based on the paper and adapted it to an eBPF-based environment to extract key-value pairs.
(10) **IntentFuzzer-AHA**: It is a variant version of IntentFuzzer that leverages AHA-Fuzz's intent fuzzing algorithm.

6.2 Effectiveness of Intent Fuzzing

To answer **RQ1** ("Is AHA-Fuzz capable of creating valid intents called by malware?"), we run AHA-Fuzz and four different fuzzers—APE, MATE, ICCBot, and IntentFuzzer—on the Ground-truth Malware Dataset. Table 5 shows the evaluation results. It shows intent usage patterns per the malware family, along with the time spent to invoke these patterns and their descriptions. Regarding the patterns, "M" denotes intent events that do not require key-value pairs in extras, while "E" indicates intent events that require extras. "S" represents long-term scheduling events.

```
1 // ProjectSpyHRX
2 public void onReceive(Context context, Intent intent) {
3     String target = intent.getStringExtra("package");
4     int hashed_target = target.hashCode();
5     if (hashed_target == 7696199939) {
6         target = "com.whatsapp";
7         Do_malicious(target);
8     }
9 }
```

Figure 6: AHA-Fuzz's Failure Pattern on ProjectSpyHRX.

We check whether intents generated by the fuzzers invoke the patterns. We confirm that AHA-Fuzz successfully invokes 27 out of the 28 patterns. However, APE fails to trigger any patterns when it performs GUI fuzzing only. MATE, ICCBot, and IntentFuzzer succeed in triggering patterns that do not require extras values, but fail to trigger patterns that require scheduling and extras. On the contrary, AHA-Fuzz triggers all intent usage patterns and scheduling events, except for one pattern that requires extras.

Payload. A payload case refers to a scenario where malware leverages the extras field of an intent to deliver a payload. For instance, *Solid* malware uses a Google Firebase intent to send a payload to victims. The malicious behavior is triggered only when the Firebase message contains a certain key-value payload (`status`: `StartAdService`) within an intent. In our evaluation, all baseline fuzzers—APE, MATE, ICCBot, and IntentFuzzer—fail to extract the proper payload triggering a malicious behavior. They use static analysis to extract values from extras, but fail to identify the specific payload used by the malware. This is because there are no predefined patterns or APIs to retrieve values from intents, making it impossible to generate the correct payload through static analysis alone. Baseline fuzzers typically extract payload values by collecting constant values (*e.g.*, predefined strings). However, app hardening techniques, such as obfuscation, can hinder the extraction of these payload-related values. On the other hand, AHA-Fuzz successfully generates the required payload values through the runtime analysis based on compare operations. Therefore, AHA-Fuzz successfully invokes four patterns (as in Table 5) that require specific payloads.

SMS. SMS-related intents (*e.g.*, `SMS_RECEIVED`) require a specific key-value pair such as `{format, 3gpp}`. While the baseline fuzzers notice that the `format` key is needed, they fail to extract proper values (*e.g.*, `3gpp`). This is because the Android framework, rather than the application, is responsible for checking the values in intents. Therefore, extracting such proper values requires not only analyzing the application code but also analyzing the framework code, which is a challenging problem due to the complexity and of the framework. On the other hand, AHA-Fuzz monitors all Android events, including those involving the Android framework. AHA-Fuzz, thus, can extract the necessary information that `format` key requires a value of `3gpp` through the key-value feedback.

Time-to-exposure. AHA-Fuzz is approximately three times faster than previous methods in triggering patterns. We note that simply invoking an intent event does not directly lead to malware's malicious behaviors. The internal state of the malware, including global variables, also affects the execution of malicious behaviors. It highlights the importance of selecting the correct intent input. We observe that our coverage-guided feedback enables AHA-Fuzz to select input seeds that are likely to trigger malicious behaviors.

Table 6: Ablation Study Results. Improve(%) indicates the performance improvement rate compared to the previous mode

Fuzzer	# of invoked pattern	Avg. TTE (Improve(%))
IntentFuzzer	15	206
AHA-KV	27	146 (41.1%)
AHA-CV	27	124 (17.7%)
AHA-Fuzz	27	66 (87.9%)

In contrast, all other fuzzers randomly select intents, which significantly reduces the likelihood of triggering intent events that lead to malicious behavior.

Failure Case. AHA-Fuzz fails to invoke 1 out of the 28 patterns in the *ProjectSpyHRX* malware family. Figure 6 illustrates a failure case where the malware receives a payload through an intent and initiates malicious activities by comparing the hash values of the payload (at Line 5). To pass the hash comparison, the correct pre-hash value is required. However, AHA-Fuzz's feedback messages provide the post-hash value, causing AHA-Fuzz to fail in generating the appropriate feedback value for the intent parameter. In addition, generating the pre-hash value is not possible with static analysis, as it also requires hash guessing.

6.3 Ablation Study

We present an ablation study conducted to assess the contributions of two types of feedback in AHA-Fuzz—key-value feedback and selective coverage feedback—to its overall fuzzing performance. To this end, we compare four fuzzers: IntentFuzzer, which receives the key values of the extras field as runtime feedback, along with AHA-Fuzz and its two variant modes—AHA-KV and AHA-CV. We measure the time spent calling 28 intent usage patterns (as in Section 6.2) and the number of invoked usage patterns.

AHA-KV invokes an additional 12 patterns that IntentFuzzer does not trigger, and its invocation speed improves 41.1%, as shown in Table 6. This result demonstrates that our key-value feedback mechanism can generate payloads to invoke new intent usage patterns, thereby reducing the time required to trigger new patterns. AHA-CV further improves performance over AHA-KV by incorporating additional coverage feedback, resulting in a 17.7% increase in call speed compared to AHA-KV. Lastly, AHA-Fuzz achieves a 87.9% faster call speed compared to AHA-CV, which utilizes non-selective coverage feedback. The performance improvement of AHA-Fuzz demonstrates that our selective coverage feedback enhances the efficiency of intent fuzzing. Our ablation study results demonstrate that coverage feedback, which measures all Android events, including GUI interactions and intents, can negatively impact the performance of intent fuzzing. In contrast, our selective coverage feedback focuses exclusively on tracking intent events, effectively reducing noise from unrelated events and improving fuzzing performance.

6.4 Code Coverage

We measure method-level code coverage (*i.e.*, the number of executed methods) to compare AHA-Fuzz with previous works (**RQ3**). We note that the input spaces of each baseline fuzzer are different. Therefore, to ensure a fair comparison, we evaluate code coverage of each baseline fuzzer and the variant versions of the baselines that use AHA-Fuzz's intent fuzzing algorithm (Section 4.2, Section 4.3).

Table 7: Measured method-level coverage on 40 apps. MATE can only successfully fuzz 11 benign apps out of 20, and 0 malware apps out of 20.

Dataset	Triggered Method (Incre.(%))							
	Ape	AHA-Fuzz	MATE	MATE-AHA	ICCBot	ICCBot-AHA	IntentFuzzer	IntentFuzzer-AHA
Benign (20)	6,880	7,734 (12.4%)	4,761	5,363 (12,7%)	6,021	6,979 (15.9%)	6,010	6,804 (13.2%)
Malware (20)	5,110	5,470 (7.0%)	Failed	Failed	3,006	3,988 (32.6%)	2,360	3,565 (51.0%)
Total (Avg)	5,995.1	6,602.1 **(10.1%)**	4,760.7	5,363.0 **(12.7%)**	4,513.7	5,483.3 **(21.5%)**	4,185.0	5,184.5 **(23.9%)**

We use a total of 40 Android apps, including the most downloaded 20 apps from AndroZoo and the most downloaded 20 malware. Although we do not intentionally select hardened apps, we observe that all 40 apps are hardened. Specifically, among the benign apps, 7 out of 20 use obfuscation, and 17 out of 20 employ anti-debugging techniques. Among the malware, 19 out of 20 apps use packers. We execute each fuzzer for one hour on each of the 40 Android apps.

As a result, fuzzers that adopt AHA-Fuzz's intent fuzzing algorithm execute more methods than a baseline fuzzer, as shown in Table 7. AHA-Fuzz calls 10.1% more methods than GUI fuzzing (APE), 21.5% more methods than the static analysis-based intent fuzzer (*i.e.*, ICCBot), and 23.9% more methods than IntentFuzzer.

APE. AHA-Fuzz achieves the highest coverage among all the baselines and variant versions using AHA-Fuzz's intent fuzzing algorithm. Meanwhile, APE (*i.e.*, GUI fuzzing only) achieves the best method-level coverage compared to other previous intent fuzzing approaches. This result indicates that GUI events are crucial for analyzing Android app behaviors, and the inclusion of GUI fuzzing in the AHA-Fuzz design is reasonable.

In terms of code coverage, APE's GUI fuzzing significantly enhances app analysis. However, from the perspective of malicious behavior, GUI fuzzing alone cannot trigger certain events, limiting its effectiveness in comprehensive app analysis as discussed in Section 6.2. Therefore, we believe that AHA-Fuzz, when combined with APE's GUI fuzzing, performs intent fuzzing that efficiently triggers a wide range of intent events associated with malicious behaviors.

MATE. We note that MATE's app-under-test fuzzing strategy requires repackaging the target app. Hence, it is not feasible to analyze hardened apps with integrity checks. Consequently, MATE fails to test 29 out of the 40 apps. This result demonstrates that AHA-Fuzz's design is effective in analyzing real-world hardened apps. Also, MATE-AHA (a variant mode of MATE) outperforms MATE's intent fuzzing algorithm. This is because MATE randomly generate intents. On the other hand, AHA-Fuzz's coverage feedback algorithm can generate a wide range of effective intents.

ICCBot and IntentFuzzer. The variant modes of ICCBot and IntentFuzzer (*i.e.*, ICCBot-AHA and IntentFuzzer-AHA) call more methods than the original modes, particularly in malware. ICCBot-AHA triggers 32.6% more methods, and IntentFuzzer-AHA triggers 51.0% more methods. This result is due to the use of packers in malware. Albeit benign apps are obfuscated, static analysis can still identify some intent-related operations. We observe developers apply obfuscation to their main logic code but exclude third-party code. However, packers used in malware, such as Qihoo [1], pack all DEX code and unpack it at runtime. As a result, the static analysis used in ICCBot and IntentFuzzer is ineffective to extract intent-related information in packed apps. In contrast, AHA-Fuzz extracts intent-related information at runtime; thus, it is unaffected by packers and obfuscators and yields better code coverage.

6.5 Discovering Unknown Bugs

We evaluate AHA-Fuzz's effectiveness in discovering bugs that previous analysis tools cannot find (**RQ4**). We run AHA-Fuzz for one hour on each of the 300 Top downloaded apps. AHA-Fuzz discovers 26 crashes and 21 information leaks, and we report them to corresponding developers. At the time of writing, four information leaks and two crashes have been confirmed. Additionally, two information leaks have been fixed by Google and Firefox.

Crash. We select ICCBot and IntentFuzzer as comparison fuzzers and verify the uniqueness of the crashes found by AHA-Fuzz by checking whether the comparison fuzzer could detect them. We exclude MATE from the comparison due to the difficulty in performing reliable analysis, as explained in Section 6.4. As in previous works, we only count crashes that lead to fatal errors (*e.g.*, excluding crashes caused by null intent fuzzing). To analyze the differences caused by the intent fuzzing algorithm, we count only the crashes where the root cause is related to the intent.

AHA-Fuzz discovers 26 crashes that other baseline fuzzers cannot reach and discover. We note that previous Android fuzzers [6, 36] mainly discover crashes caused by null pointer exceptions. On the contrary, AHA-Fuzz primarily detects crashes involving `IllegalArgumentException` and `ClassCastException`. AHA-Fuzz triggers such crashes when the extras field value from AHA-Fuzz is outside the expected range (`IllegalArgumentException`), or when an unexpected type is encountered (`ClassCastException`), *e.g.*, expecting `JSON` but receiving a `String`.

We observe that the primary reason AHA-Fuzz can discover unknown bugs lies in the diversity of intent inputs. As the fuzzer explores deeper code paths, AHA-Fuzz extracts more intent-related information, regardless of whether apps use hardening techniques. In contrast, all baseline fuzzers rely on static analysis to mutate values from extras, resulting in limited input diversity.

Information Leak. We discover 21 information leaks via our lightweight dynamic taint analysis (as explained in Section 4.4.2). We identify four information leakage patterns: network information, authentication code, device ID, and app installation paths, as shown in Table 8. We note that AHA-Fuzz can detect various information leaks through method-level taint propagation. In the case of the 21 information leaks AHA-Fuzz found, we confirm that sensitive information is processed and leaked through the Java standard library, regardless of the hardening technique (*e.g.*, `String.append`).

We also confirm that AHA-Fuzz's intent fuzzing reaches buggy code (causing information leaks) that previous works cannot. For instance, regarding network information leak, AndroidX uses a custom wrapper class for `Log`. During execution, the custom `Log` class is initialized in release mode, which sanitizes debug logs. Yet,

Table 8: 47 Previously Unknown Bugs Discovered by AHA-Fuzz.

Oracle	Bug pattern	Package	Number
Crash	IllegalArgumentException	Gmail + 13 more	14
Crash	ClassCastException	Google Home + 7 more	8
Crash	NullPointerException	Roblox + 4 more	4
Leak	Network	Android-sdk + 5 more	6
Leak	Authentication code	Android-sdk	1
Leak	Device id	Facebook	1
Leak	App installation path	Firefox + 12 more	13

AHA-Fuzz calls the uninitialized custom `Log` class, causing it to be initialized in debug mode. In the debug mode of their custom `Log`, network capabilities are logged, which require permissions (*e.g.*, `ACCESS_NETWORK_STATE`) to access. We also confirmed that the Facebook device ID leak and Firebase authentication code leak also occur only through the intent event.

To verify that the information leaks we discover are unique bugs that other tools cannot find, we first attempt to use dynamic taint analysis tools but is unsuccessful. System modification-based taint analysis tools [18, 59, 62, 66] only support versions up to Android 6, while most of the apps we test (98%) require Android 8 or higher versions. Other dynamic taint analysis tools [3, 64], including DBI, require manual configuration to set exact source and sink points, which makes them unsuitable for discovering new information leaks. Thus, we use FlowDroid [5], a popular static taint analysis tool, to verify whether it detects any of the 21 bugs. We confirm that FlowDroid cannot detect any of these bugs due to app hardening techniques that reduce the accuracy of static analysis.

For instance, AHA-Fuzz detects a device ID leak in the Facebook app, triggered by a `GET_PHONE_ID` intent generated by AHA-Fuzz. This leaked device ID enables user identification, which leads to a privacy leakage that enables cross-app tracking [74] and unauthorized user profiling [17]. Existing taint analysis approaches, such as FlowDroid [5] and Vialin [3], fail to detect the issue because the component is currently fully hardened. More specifically, the buggy code is located in a dynamically loaded module, which is invisible to static analysis (FlowDroid) or fails to instrument the app code (Vialin). In contrast, AHA-Fuzz successfully detects this bug by leveraging method-level taint propagation, and this bug is fixed by Facebook.

Bug Report. Since Android apps on Google Play are closed-source, direct communication with developers for bug confirmation is not feasible. Among the 46 apps in which AHA-Fuzz identified bugs, only Google, Facebook, and Firefox allow users to report bugs in public security bug reporting platforms. They acknowledge four information leaks and two crashes and fix three information leaks.

7 Discussion

Testing Real Devices. Research literature [57, 62] recommends fuzzing on real Android devices rather than testing the inputs on Android emulators due to the anti-analysis problems. We note that users can run AHA-Fuzz on real Android devices with root permissions since AHA-Fuzz leverages a dynamic analysis environment based on eBPF. Yet, we evaluate AHA-Fuzz on emulators rather than the real devices. The main reason is that our evaluation tests more than 300 different Android apps with ten different modes; thus, testing with real devices is impractical to evaluate all sets.

Requiring Root Permissions. AHA-Fuzz leverages an eBPF-based dynamic analysis environment. To activate eBPF kernel subsystem, it requires root permissions. Yet, it is worth noting that Android rooting can be easily accomplished on Android emulators (*e.g.*, using "adb root" command [29]). Further, certain Android malware exclusively functions on rooted Android [51]. Thus, we believe that the rooted Android environment is not a significant limitation and can be beneficial for analyzing Android malware.

Information leak detection coverage. Sensitive information is generally stored and managed as primitive or String types, and AHA-Fuzz is capable of detecting information leaks involving these types. However, when the data is stored in other types or transformed [15] (*e.g.*, via encoding or encryption), AHA-Fuzz is unable to detect the associated information leaks. These information leaks can be detected by introducing new automatic approaches for identifying related APIs, or by manually specifying them as additional tracking targets.

8 Future work

Special Type Mutation. AHA-Fuzz currently faces limitations in mutating special types (*e.g.*, `Binder` or `Serializable`) because the object layouts of such types differ across apps. Although our analysis in Table 4 (Section 4.2.2) shows that only 6.5% of the extras in tested apps involve special types, as our future work, we plan to extend AHA-Fuzz to support special type mutation by recovering the layout of these special type objects based on our Java object layout recovery techniques (Section 4.2.1).

Bug Analysis. AHA-Fuzz currently supports detecting two types of bugs, crashes and information leaks, but it faces challenges when analyzing identified bugs (*e.g.*, root cause analysis). While AHA-Fuzz records all framework API calls to trace which APIs are invoked by buggy code, conducting in-depth analyses such as root cause investigation or patch generation remains difficult. In future work, we plan to enhance AHA-Fuzz with the capabilities to effectively analyze bugs in hardened apps, such as reconstructing backtraces that are resistant to hardening techniques.

Cross-platform Framework App Support. Recently, many apps have been developed using new cross-platform frameworks (*e.g.*, Flutter [23], Xamarin [48], and React Native [53]). Consequently, there is a growing need for analysis tools tailored to these frameworks. Since AHA-Fuzz is based on eBPF and does not heavily depend on specific cross-platform frameworks at runtime, we plan to extend AHA-Fuzz testing scope to newly emerging cross-platform framework apps as our future work.

9 Related Work

Fuzzing. Prior fuzzing works for Android apps attempt to maximize code coverage similar to AHA-Fuzz [6, 36, 57, 58, 70], but they deal with GUI fuzzing and simple intents. APE [36] is an automated model-based GUI testing technique that dynamically refines GUI models to improve code coverage and crash detection. This approach does not focus on the generation of intents. Stoat [58] is a model-based testing approach by generating stochastic models that describe GUI interactions. It constructs models dynamically and iteratively mutates them to generate diverse test cases, while also injecting system-level events, including broadcast intents, to uncover

intricate bugs. Yet, it is limited to sending Manifest-defined system broadcast intents without intent-related feedback (*e.g.*, *action* and *extras* fields). Furthermore, Auer et al. [6] suggest that combining UI inputs with intents triggers higher code coverage than sending either alone. Their empirical study identified a specific ratio that yielded the best results, but considering the unique combination for each app, as in our approach, allows for more efficient fuzzing. Intent fuzzing tools [6, 13, 38, 40, 46, 57, 67] focus on generating appropriate extras (*i.e.*, key-value pairs) to maximize code coverage. However, these methods rely on static analysis, making them ineffective against obfuscation techniques. DroidFuzzer [70] and Sasnauskas et al. [57] propose testing approaches sending intents to components of target apps. Yet, these works only treat activity intent and its simple usage (*e.g.*, null value and predefined constant).

Dynamic Analysis for Android Security. Many frameworks attempt to detect security threats (*e.g.*, information leak, privilege escalation) in Android apps by tracing code executions on dynamic instrumentation tools [60, 62, 64, 66, 73]. They reconstruct or track data flows to figure out the behaviors of the apps by modifying Android systems and emulators, or by using DBI [60, 62, 64, 66]. Yet, these works require Android system modification or incur high overhead; thus, users also need to update their frameworks with each system version update (*i.e.*, Android version update) or face high runtime overhead. To fill this gap, our approach does not need system modifications by leveraging eBPF. This is because it uses kernel subsystem with support from Android OS [22]. NCScope [73] also uses eBPF programs to retrieve the in-memory data. However, it only covers native code execution and requires specific hardware (ARM ETM) to trace CPU-level instructions.

10 Conclusion

We introduce AHA-Fuzz, the first intent-aware greybox fuzzing framework for hardened Android apps. AHA-Fuzz has three key aspects that distinguish it from previous works: (*i*) introducing a valid intent generator by recovering object layouts and leveraging key-value feedback to create valid intent inputs, (*ii*) precisely evaluating the impact of these generated inputs using a selective coverage feedback approach, and (*iii*) introducing approaches for efficiently triggering hard-to-trigger bugs and detecting information leaks in hardened app. Our evaluation results show that AHA-Fuzz is effective in triggering actions that previous works cannot reach. Especially, compared to previous works, AHA-Fuzz triggers 92.3% more intents (3.45× faster) and 23.9% more methods. AHA-Fuzz also discovers 47 previously unknown bugs that previous works cannot find. Among the 47 bugs found, 6 have been acknowledged by developers from Google, Firefox, and Facebook, and three have been fixed. The open-source version of AHA-Fuzz is available at https://github.com/S2-Lab/AHA-fuzz.

Acknowledgments

This work was supported by a Korea Internet & Security Agency (KISA) grant funded by the Korean government (PIPC) (No. 2780000017). This work was partly supported by ICT Creative Consilience Program through the Institute of Information & Communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT)(IITP-2025-RS-2020-II201819, 10%). This work was also supported by the MSIT (Ministry of Science and ICT), Korea, under the ITRC (Information Technology Research Center) support program (No. 2710078921) supervised by the IITP. This work was also supported by the IITP grant funded by the Korea government (MSIT) (RS-2024-00341722, RS-2024-00437306, RS-2024-00341722, and RS-2022-000563, Development of Cyber Resilience Method for Intelligent Service Robots, Development of Integrated Platform for Expanding and Safely Applying MemorySafe Languages, AI-Based Automated Vulnerability Detection and Safe Code Generation, and Automatic Deep Malware Analysis Technology for Cyber Threat Intelligence).

References

[1] Qihoo 360. *Products and Services*, Accessed: Jan. 10, 2025. http://www.360.cn/about/englishversion.html.
[2] Yaniv Agman and Danny Hendler. Bpfroid: Robust real time android malware detection framework. *arXiv preprint arXiv:2105.14344*, 2021.
[3] Khaled Ahmed, Yingying Wang, Mieszko Lis, and Julia Rubin. Vialin: Path-aware dynamic taint analysis for android. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2023, page 1598–1610, 2023.
[4] Saswat Anand. *ELLA: A Tool for Binary Instrumentation of Android Apps*, Accessed: Jan. 10, 2025. https://github.com/saswatanand/ella.
[5] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *ACM sigplan notices*, 49(6):259–269, 2014.
[6] Michael Auer, Andreas Stahlbauer, and Gordon Fraser. Android fuzzing: Balancing user-inputs and intents. In *2023 IEEE Conference on Software Testing, Verification and Validation (ICST)*, pages 37–48, 2023.
[7] Michael Backes, Sven Bugiel, Oliver Schranz, Philipp von Styp-Rekowsky, and Sebastian Weisgerber. Artist: The android runtime instrumentation and security toolkit. In *2017 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 481–495. IEEE, 2017.
[8] backlinko. *iPhone vs Android Statistics*, Accessed: Jan. 10, 2025. https://backlinko.com/iphone-vs-android-statistics.
[9] Baidu. *Baidu*, Accessed: Jan. 10, 2025. https://app.baidu.com.
[10] bankmycell. *How many apps in google play store? (2024)*, Accessed: Jan. 10, 2025. https://www.bankmycell.com/blog/number-of-google-play-store-apps/.
[11] Frederic Besler, Carsten Willems, and Ralf Hund. Countering innovative sandbox evasion techniques used by malware. In *29th Annual FIRST Conference*, 2017.
[12] Michael Cao, Khaled Ahmed, and Julia Rubin. Rotten apples spoil the bunch: An anatomy of google play malware. In *Proceedings of the 44th International Conference on Software Engineering*, pages 1919–1931, 2022.
[13] Kwanghoon Choi, Myungpil Ko, and Byeong-Mo Chang. A practical intent fuzzing tool for robustness of inter-component communication in android apps. *KSII Transactions on Internet & Information Systems*, 12(9), 2018.
[14] Minseong Choi, Yubin Im, Steve Ko, Yonghwi Kwon, Yuseok Jeon, and Haehyun Cho. Dryjin: Detecting information leaks in android applications. In *IFIP International Conference on ICT Systems Security and Privacy Protection*, pages 76–90. Springer, 2024.
[15] Andrea Continella, Yanick Fratantonio, Martina Lindorfer, Alessandro Puccetti, Ali Zand, Christopher Kruegel, Giovanni Vigna, et al. Obfuscation-resilient privacy leak detection for mobile apps through differential analysis. In *NDSS*, volume 17, pages 10–14722, 2017.
[16] Shuaike Dong, Menghao Li, Wenrui Diao, Xiangyu Liu, Jian Liu, Zhou Li, Fenghao Xu, Kai Chen, Xiaofeng Wang, and Kehuan Zhang. Understanding android obfuscation techniques: A large-scale investigation in the wild. In *Security and privacy in communication networks: 14th international conference, secureComm 2018*, pages 172–192. Springer, 2018.
[17] Zikan Dong, Tianming Liu, Jiapeng Deng, Li Li, Minghui Yang, Meng Wang, Guosheng Xu, and Guoai Xu. Exploring covert third-party identifiers through external storage in the android new era. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 4535–4552, 2024.
[18] William Enck, Peter Gilbert, Seungyeop Han, Vasant Tendulkar, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. volume 32, June 2014.
[19] ExtendedAndroidTools. *facebookexperimental*, Accessed: Jan. 10, 2025. https://github.com/facebookexperimental/ExtendedAndroidTools.
[20] Github. *CodeQL*, Accessed: Jan. 10, 2025. https://codeql.github.com/.
[21] Google. *Bundle*, Accessed: Jan. 10, 2025. https://developer.android.com/reference/android/os/Bundle.

[22] Google. *Extending the Kernel with eBPF*, Accessed: Jan. 10, 2025. https://source.android.com/docs/core/architecture/kernel/bpf.

[23] Google. *Flutter - Build for any screen*, Accessed: Jan. 10, 2025. https://flutter.dev.

[24] Google. *Intents and intent filters*, Accessed: Jan. 10, 2025. https://developer.android.com/guide/components/intents-filters.

[25] Google. *NDK*, Accessed: Jan. 10, 2025. https://developer.android.com/ndk.

[26] google. *Play Integrity API*, Accessed: Jan. 10, 2025. https://developer.android.com/google/play/integrity.

[27] Google. *Schedule alarms*, Accessed: Jan. 10, 2025. https://developer.android.com/develop/background-work/services/alarms/schedule.

[28] Google. *Shrink, obfuscate, and optimize your app*, Accessed: Jan. 10, 2025. https://developer.android.com/build/shrink-code.

[29] Google. *Start the emulator from the command line*, Accessed: Jan. 10, 2025. https://developer.android.com/studio/run/emulator-commandline.

[30] Google. *UI/Application Exerciser Monkey*, Accessed: Jan. 10, 2025. https://developer.android.com/studio/test/other-testing-tools/monkey.

[31] Google. *Android 12 features and changes list*, Accessed: May. 5, 2025. https://developer.android.com/about/versions/12/summary.

[32] Google. *Android 13 features and changes list*, Accessed: May. 5, 2025. https://developer.android.com/about/versions/13/summary.

[33] Google. *DexGuard*, Accessed: May. 5, 2025. https://source.android.com/docs/core/runtime/jit-compiler.

[34] Google. *File-based encryption*, Accessed: May. 5, 2025. https://source.android.com/docs/security/features/encryption/file-based.

[35] Google. *Log Info Disclosure*, Accessed: May. 5, 2025. https://developer.android.com/privacy-and-security/risks/log-info-disclosure.

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

[37] Yujiang Gui, Dongjie He, and Jingling Xue. Merge-replay: Efficient ifds-based taint analysis by consolidating equivalent value flows. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 319–331. IEEE, 2023.

[38] Hui Guo, Ting Su, Xiaoqiang Liu, Siyi Gu, and Jingling Sun. Effectively finding icc-related bugs in android apps via reinforcement learning. In *2023 IEEE 34th International Symposium on Software Reliability Engineering (ISSRE)*, pages 403–414, 2023.

[39] Vincent Haupert, Dominik Maier, Nicolas Schneider, Julian Kirsch, and Tilo Müller. Honey, i shrunk your app security: The state of android app hardening. In *Detection of Intrusions and Malware, and Vulnerability Assessment: 15th International Conference, DIMVA 2018*, pages 69–91. Springer, 2018.

[40] Roee Hay, Omer Tripp, and Marco Pistoia. Dynamic detection of inter-application communication vulnerabilities in android. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ISSTA 2015, page 118–128, 2015.

[41] Ijiami. *Ijiami*, Accessed: Jan. 10, 2025. https://www.ijiami.cn.

[42] Hiroki Inayoshi, Shohei Kakei, and Shoichi Saito. Execution recording and reconstruction for detecting information flows in android apps. *IEEE Access*, 11:10730–10750, 2023.

[43] Satori Threat Intelligence and Research Team. *TERRACOTTA Android Malware: A Technical Study*, Accessed: Jan. 10, 2025. https://www.humansecurity.com/learn/blog/terracotta-android-malware-a-technical-study.

[44] iovisor. *bcc*, Accessed: Jan. 10, 2025. https://github.com/iovisor/bcc.

[45] Jiagu. *Jiagu*, Accessed: Jan. 10, 2025. https://jiagu.360.cn.

[46] Anatoli Kalysch, Mark Deutel, and Tilo Müller. Template-based android inter process communication fuzzing. In *Proceedings of the 15th International Conference on Availability, Reliability and Security*, ARES '20, 2020.

[47] Licle. *DexProtector: Multi-layered RASP solution that secures your Android and iOS apps against static and dynamic analysis, illegal use and tampering*, Accessed: Jan. 10, 2025. https://dexprotector.com.

[48] Microsoft. *What is Xamarin?*, Accessed: Jan. 10, 2025. https://learn.microsoft.com/previous-versions/xamarin/get-started/what-is-xamarin.

[49] Mobile-IoT-Security-Lab. *Obfuscapk*, Accessed: Jan. 10, 2025. https://github.com/Mobile-IoT-Security-Lab/Obfuscapk.

[50] Andy Nisbet, Nuno Miguel Nobre, Graham Riley, and Mikel Luján. Profiling and tracing support for java applications. In *Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering*, pages 119–126, 2019.

[51] OWASP. *Android Anti-Reversing Defenses*, Accessed: Jan. 10, 2025. https://github.com/OWASP/owasp-mastg/blob/master/Document/0x05j-Testing-Resiliency-Against-Reverse-Engineering.md.

[52] Aleksandr Pilgun, Olga Gadyatskaya, Yury Zhauniarovich, Stanislav Dashevskyi, Artsiom Kushniarou, and Sjouke Mauw. Fine-grained code coverage measurement in automated black-box android testing. *ACM Trans. Softw. Eng. Methodol.*, 29(4), jul 2020.

[53] Meta Platforms. *React Native*, Accessed: Jan. 10, 2025. https://reactnative.dev.

[54] Bradley Reaves, Jasmine Bowers, Sigmund Albert Gorski III, Olabode Anise, Rahul Bobhate, Raymond Cho, Hiranava Das, Sharique Hussain, Hamza Karachiwala, Nolen Scaife, Byron Wright, Kevin Butler, William Enck, and Patrick Traynor. *droid: Assessment and evaluation of android application analysis tools. *ACM Comput. Surv.*, 49(3), oct 2016.

[55] rednaga. *APKiD: Fast Identification of Mobile RASP SDKs*, Accessed: Jan. 10, 2025. https://github.com/rednaga/APKiD.

[56] Andrea Romdhana, Mariano Ceccato, Gabriel Claudiu Georgiu, Alessio Merlo, and Paolo Tonella. Cosmo: Code coverage made easier for android. In *2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST)*, pages 417–423, 2021.

[57] Raimondas Sasnauskas and John Regehr. Intent fuzzer: crafting intents of death. In *Proceedings of the 2014 Joint International Workshop on Dynamic Analysis (WODA) and Software and System Performance Testing, Debugging, and Analytics (PERTEA)*, pages 1–5, 2014.

[58] Ting Su, Guozhu Meng, Yuting Chen, Ke Wu, Weiming Yang, Yao Yao, Geguang Pu, Yang Liu, and Zhendong Su. Guided, stochastic model-based gui testing of android apps. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, page 245–256, 2017.

[59] Mingshen Sun, Tao Wei, and John CS Lui. Taintart: A practical multi-level information-flow tracking system for android runtime. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 331–342, 2016.

[60] Kimberly Tam, Salahuddin J Khan, Aristide Fattori, and Lorenzo Cavallaro. Copperdroid: Automatic reconstruction of android malware behaviors. In *Ndss*, pages 1–15, 2015.

[61] Tencent. *Tencent Cloud*, Accessed: Jan. 10, 2025. https://www.tencentcloud.com.

[62] Lei Xue, Chenxiong Qian, Hao Zhou, Xiapu Luo, Yajin Zhou, Yuru Shao, and Alvin TS Chan. Ndroid: Toward tracking information flows across multiple android contexts. *IEEE Transactions on Information Forensics and Security*, 14(3):814–828, 2018.

[63] Lei Xue, Hao Zhou, Xiapu Luo, Le Yu, Dinghao Wu, Yajin Zhou, and Xiaobo Ma. Packergrind: An adaptive unpacking system for android apps. *IEEE Transactions on Software Engineering*, 48(2):551–570, 2020.

[64] Lei Xue, Yajin Zhou, Ting Chen, Xiapu Luo, and Guofei Gu. Malton: Towards {On-Device}{Non-Invasive} mobile malware analysis for {ART}. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 289–306, 2017.

[65] Jiwei Yan, Shixin Zhang, Yepang Liu, Jun Yan, and Jian Zhang. Iccbot: fragment-aware and context-sensitive icc resolution for android applications. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*, ICSE '22, page 105–109, 2022.

[66] Lok Kwong Yan and Heng Yin. {DroidScope}: Seamlessly reconstructing the {OS} and dalvik semantic views for dynamic android malware analysis. In *21st USENIX security symposium (USENIX security 12)*, pages 569–584, 2012.

[67] Kun Yang, Jianwei Zhuge, Yongke Wang, Lujue Zhou, and Haixin Duan. Intentfuzzer: detecting capability leaks of android applications. In *Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security*, ASIA CCS '14, page 531–536, 2014.

[68] Wenbo Yang, Yuanyuan Zhang, Juanru Li, Junliang Shu, Bodong Li, Wenjun Hu, and Dawu Gu. Appspear: Bytecode decrypting and dex reassembling for packed android malware. In *International Symposium on Recent Advances in Intrusion Detection*, pages 359–381. Springer, 2015.

[69] Zhi Yang, Zhanhui Yuan, Shuyuan Jin, Xingyuan Chen, Lei Sun, Xuehui Du, Wenfa Li, and Hongqi Zhang. Fsaflow: Lightweight and fast dynamic path tracking and control for privacy protection on android using hybrid analysis with state-reduction strategy. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 2114–2129. IEEE, 2022.

[70] Hui Ye, Shaoyin Cheng, Lanbo Zhang, and Fan Jiang. Droidfuzzer: Fuzzing the android apps with intent-filter tag. In *Proceedings of International Conference on Advances in Mobile Computing & Multimedia*, pages 68–74, 2013.

[71] Xueling Zhang, Xiaoyin Wang, Rocky Slavin, and Jianwei Niu. Condysta: Context-aware dynamic supplement to static taint analysis. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 796–812, 2021.

[72] Yueqian Zhang, Xiapu Luo, and Haoyang Yin. Dexhunter: toward extracting hidden code from packed android applications. In *Computer Security–ESORICS 2015: 20th European Symposium on Research in Computer Security, Vienna, Austria, September 21-25, 2015, Proceedings, Part II 20*, pages 293–311. Springer, 2015.

[73] Hao Zhou, Shuohan Wu, Xiapu Luo, Ting Wang, Yajin Zhou, Chao Zhang, and Haipeng Cai. Ncscope: hardware-assisted analyzer for native code in android apps. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 629–641, 2022.

[74] Sebastian Zimmeck, Jie S Li, Hyungtae Kim, Steven M Bellovin, and Tony Jebara. A privacy analysis of cross-device tracking. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 1391–1408, 2017.