Vulcan: Lessons on Reliability of Wearables through State-Aware Fuzzing

Edgardo Barsallo Yi
ebarsall@purdue.edu
Purdue University
West Lafayette, IN

Heng Zhang
Purdue University
West Lafayette, IN
zhan2614@purdue.edu

Amiya K. Maji
Purdue University
West Lafayette, IN
amaji@purdue.edu

Kefan Xu
Purdue University
West Lafayette, IN
xu1405@purdue.edu

Saurabh Bagchi
Purdue University
West Lafayette, IN
sbagchi@purdue.edu

ABSTRACT

As we look to use Wear OS (formerly known as Android Wear) devices for fitness and health monitoring, it is important to evaluate the reliability of its ecosystem. The goal of this paper is to understand the reliability weak spots in Wear OS ecosystem. We develop a state-aware fuzzing tool, Vulcan, without any elevated privileges, to uncover these weak spots by fuzzing Wear OS apps. We evaluate the outcomes due to these weak spots by fuzzing 100 popular apps downloaded from Google Play Store. The outcomes include causing specific apps to crash, causing the running app to become unresponsive, and causing the device to reboot. We finally propose a proof-of-concept mitigation solution to address the system reboot issue.

CCS CONCEPTS

• **Computer systems organization** → *Reliability*; • **Software and its engineering** → *Software testing and debugging*.

ACM Reference Format:
Edgardo Barsallo Yi, Heng Zhang, Amiya K. Maji, Kefan Xu, and Saurabh Bagchi. 2020. Vulcan: Lessons on Reliability of Wearables through State-Aware Fuzzing. In *The 18th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys '20), June 15–19, 2020, Toronto, ON, Canada.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3386901.3388916

1 INTRODUCTION

Google's Wear OS has been one of the dominant OSes for wearable devices, which include smartwatches, smart glasses, and fitness trackers. Wear OS is based on Android with similar features such as kernel, programming model, app life cycle, development framework, etc. However, unlike Android apps on the mobile devices, wearable apps have somewhat different characteristics, as does Wear OS relative to Android. The fundamental driver of the differences is the limited display area and the difficulty of executing interactive work (such as typing) on a wearable device. As a result, wearable apps tend to have more number of Services (which run in the background) relative to Activities (which run in the foreground), have fewer GUI components, and have tethering to a counterpart app on the mobile device [32]. Moreover, wearable devices are often fitted with a variety of sensors (e.g., heart rate monitor, pulse oximeter, and even electrocardiogram or ECG sensor) each with its own device driver software. Device drivers have been found to be reliability weak spots in the server world [29, 41]. Further, wearable apps are generally more dependent on these sensors, have unique interaction patterns, both intra-device and inter-device (between the smartwatch and the phone it is paired with), and are aware of the physical context. These lead to distinctive software and software-hardware interaction patterns in wearable systems. As we are poised to use wearable devices for critical apps, such as clinical-grade health monitoring, it is important to understand the vulnerabilities in their software architecture and how best to mitigate them[1].

Although there have been several previous works focusing on the reliability of the Android ecosystem [13, 25–27, 36, 37], there are only few that study Wear OS [9, 32, 33, 45, 46]. However, none of them consider the effects of the inter-device communication (between the mobile device and the wearable device) and the effect of sensor activities on the reliability of the wearable apps — both of these are dominant software patterns in wearable systems. The closest prior work is QGJ [9], which presented a black-box fuzzing tool and defined a set of fuzzing campaigns to test the robustness of wearable apps. However, QGJ is agnostic with respect to the state of a wearable app and generates a large number of inputs that are invalid and silently discarded by Wear OS. Further, for those failures that it can trigger, the triggering mechanism of QGJ is much less efficient relative to our solution. Apart from QGJ, Monkey [5] and APE [25] are two tools for testing Android apps by generating different UI events, which simulate how a user would possibly interact with the apps. However, we show that UI fuzzers by themselves are not effective in uncovering vulnerabilities in

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
MobiSys '20, June 15–19, 2020, Toronto, ON, Canada
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-7954-0/20/06…$15.00
https://doi.org/10.1145/3386901.3388916

[1]We use the term "vulnerability" not only in the security sense, but any bug that may naturally lead to failures or that can be exploited through malicious actions.

wearable systems because the UI interactions are fundamentally less complex here.

Our Solution: Vulcan

In this paper, we present Vulcan[2], a state-aware fuzzer for wearable apps. Its workflow is shown in Figure 1 and includes the offline training and model building and the online fuzzing components. ***Vulcan brings three innovations, which work together to improve the fuzzing effectiveness.*** *First*, Vulcan separately triggers the intra-device and inter-device interactions, thus leveraging the insight that the system is more vulnerable when wearable-smartphone synchronization is taking place (Table 4 shows the communication fuzzing results). *Second*, we define the notion of vulnerable states where bugs are more likely to be manifested. Vulcan steers the app to the vulnerable states and fuzzes the app at those states, which significantly increases the fault activation rate. This leverages the insight that failures are often correlated with higher degree of concurrency (Tables 3 and 4 show the results of fuzzing apps at different degrees of concurrency, where the concurrency is correspondent to the degrees of sensor activation defined in Table 2). *Third*, Vulcan parses the Android Intent specification and creates a directed set of Intents which are injected to the wearable apps to effectively trigger bugs (Section 3.4 shows the details of the parser). Specifically, the created Intents are based on what action-data pairs are valid according to the specification, e.g., `ACTION_DIAL` can take either empty data or the URI of a phone number. Summarizing, Vulcan needs neither the source codes of the wearable apps (or their mobile counterpart app) nor root privilege on the wearable device. Rather, it improves the efficiency in triggering bugs in wearable apps and causes device reboots through inputs generated from a *user-level* app.

To evaluate Vulcan, we conduct a detailed study of the top 100 free apps from the Google Play Store — in terms of share, the top categories are, in order: Health & Fitness, Tools, and Productivity. Using Vulcan, we inject over 1M Intents between inter-device and intra-device interactions. We uncover some foundational reliability weak spots in the design of Wear OS and several vulnerabilities that are common across multiple apps. We categorize the failure events into three classes—app not responding, app crash, and the most catastrophic failure, system reboot, which refers to the case where our injection is able to cause the wearable device to reboot, thus pointing to a vulnerability in the system software stack. Compared to the three prior tools, **Monkey** [5], **APE** [25], and **QGJ** [9], Vulcan is the only one that can trigger system reboots deterministically and Vulcan runs without elevated system-level privileges. Further, for app crashes that both QGJ and Vulcan can trigger, Vulcan triggers them more efficiently, at a rate 5.5X compared to QGJ, when normalized by the number of injected Intents (Table 5). Vulcan renders the insights to the Wear OS developers to avoid system reboots from a user level app and can be used by app developers to test the reliability of their apps.

We delve into the details to analyze the system reboots in Section 6. We find that these reboots are due to the Watchdog monitor of Wear OS over-aggressively terminating the Android System Server, a critical system level process, under various conditions

[2]Vulcan is the Roman God of fire and metalworking, the blacksmith of the Gods. The aspiration is for our tool to forge more reliable wearable systems.

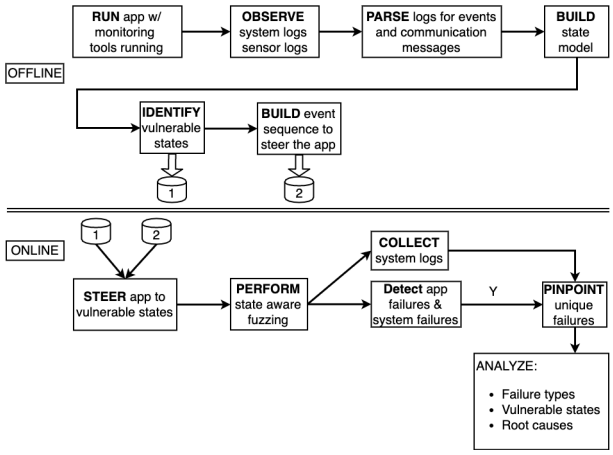


Figure 1: Workflow of Vulcan. This shows the offline phase whereby, through simulated UI events, the tool learns the finite state machine model of the app. In the online phase, the tool steers the app to the vulnerable states and injects messages or Intents in a directed manner, which leads to a higher fault activation rate and even system reboots. In online phase, the collected logs are also analyzed to deduplicate the failure cases and identify the manifestations and root causes.

of resource contention. In our experiments, we found 18 system reboots from 13 apps, while none of the prior tools uncover any. We show the system reboots can be deterministically triggered and show that they are correlated with high degree of concurrency in events such as sensor activity or inter-device communications. We also design and develop a Proof-Of-Concept (POC) solution to show how system reboots can be avoided. We evaluate the solution's effectiveness and its usability, the latter through a small user study. In the user study, only 6.7% of the users felt that the solution significantly affected their usability of a third-party calendar app on the wearable. Overall, our state-aware tool is more effective in triggering app crashes and is the only one to trigger system reboots, compared to state-agnostic fuzzers such as QGJ as well as UI fuzzers such as Monkey and APE.

Our results reveal several interesting root causes for the failures—abundance of improper exception handling, presence of legacy codes in Wear OS copied from Android, and error propagation across devices. We find that the distribution of exceptions that cause app crashes is different from that in Android [36]. While `NullPointerException` is still a dominant culprit, its relative incidence has become less while `IllegalArgumentException` and `IllegalStateException` have become numerous in wearable apps.

Our key contributions in this paper are:

(1) **State-aware fuzzing:** We present a state-aware fuzzing tool for wearable apps. Our tool, Vulcan, can infer the state-model of a wearable app without source code access, guide it to specific states, and then run targeted fuzzing campaigns. Our tool can be downloaded from [1] and used with the latest version of Wear OS.

(2) **Higher failure rate:** We show that stateful fuzzing increases the fault activation rate compared to a state-agnostic approach and compare it to three state-of-the-art tools, QGJ,

Monkey, and APE. We find several novel and critical failure cases through these experiments and suggest software architecture improvements to make the wearable ecosystem more reliable.

(3) **System reboot:** We demonstrate that it is possible to trigger system reboots *deterministically* through Intent injection, *without* elevated system-level privileges. We show that the Wear OS vulnerability is correlated with high degree of concurrency in sensor activity and inter-device communication between mobile and wearable devices. Our POC solution shows an approach to preventing these system reboots without needing Wear OS framework changes.

The rest of the paper is organized as follows. After presenting an overview of Wear OS and testing approaches on Android in Section 2, we discuss the design of the fuzzing tool in Section 3. Then, in Section 4 we introduce the relevant aspects of our implementation and present the detailed experiments and results in Section 5. Next, we delve into the system reboots in Wear OS and propose a solution to avoid them in Section 6. We discuss lessons and threats to validity in Section 7. Then we discuss related work and conclude the paper.

2 BACKGROUND AND MOTIVATION

2.1 Wear OS

Wear OS apps, compared to their mobile counterparts, have a minimalist UI design, more focus on services (which are background processes), and have a sensor-rich application context. Although Wear OS supports standalone apps, some wearable apps are tightly coupled with their mobile counterparts. Those wearable apps frequently communicate with their companion apps in the mobile device to share data or to exchange IPC messages. In our evaluation of the top 100 apps, we find approximately 40 either need or are significantly improved with a mobile companion app.

Interaction in wearable apps can happen either intra-device or inter-device, typically between the mobile and the wearable devices. **The intra-device interaction we will refer to as "Intent" and the inter-device interaction as "Communication".**

Intra-device interaction typically happens through the binder IPC framework [13] and takes the form of Intent messages. Intent is an abstraction of operations in the Android ecosystem. In an Intent message, the developer can specify many attributes such as a specific action (e.g., `ACTION_CALL`) and data (e.g., the phone number of the callee). For example, the main activity of a hiking app can periodically send Intent messages to a background service to request the GPS location to build up the hiking map of the user.

For communication, two types of messages may be exchanged: *message passing (asynchronous)* or *data synchronization (synchronous)*. These are commonly used respectively to send commands or small data items, or synchronize data among devices. The inter-device communication is also used for delegation of actions from the wearable to the mobile. An example is shown in Figure 2 where a WiFi connection is being set up on the wearable, but part of the interaction (the password entry) happens on the mobile.

2.2 Existing Testing Approaches

Most existing research on wearable and, in general, Android application testing follow a GUI-centric approach [2, 7, 14, 25, 30, 47].

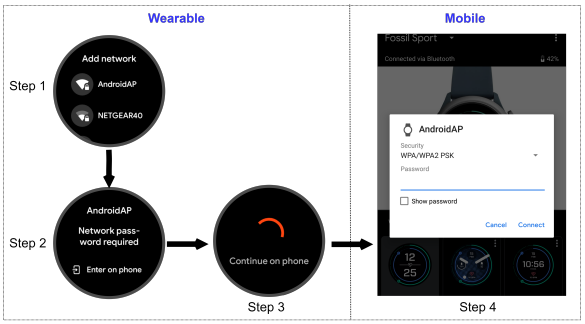


Figure 2: Example of the inter-device communication between a wearable and a mobile device. The setup of a WiFi connection starts in the wearable, but then the process continues on the mobile due to the difficulty of typing the password on the wearable device.

These tools focus on uncovering reliability issues related to the UI, by simulating user events, such as clicks, touches, or gestures. The performance metric for most of these tools is improved code coverage, which is achieved by building a model of the UI components in the target app. Compared to Android, Wear OS apps have limited display area, and hence, micro transactions are predominant [16]. We observe that although GUI testing is important for improving user experience, it overlooks some important characteristics of wearables: *inter-device communication* and *sensor awareness*. To the best of our knowledge, Vulcan is the only testing tool for wearable apps which focuses on these types of interactions.

In terms of research on inter-component communication in Android and Wear OS, earlier tools either perform static analysis of the source codes [13], or perform black-box fuzzing of target apps [9, 37]. While black-box fuzzing is simple to implement, due to the enormous input space for communication messages, its efficiency is often limited. In contrast, Vulcan uses a state-aware fuzzing approach to discover more bugs in less time. A key motivation behind Vulcan comes from the fact that behaviors of wearable apps are often state (or context) dependent, where state can either be the state of the app (e.g., operation being performed) or the state of the device (e.g., physical location or movement of the wearable device). In this paper, we evaluate the robustness of wearable apps by fuzzing both inter-device communication and intra-device Intents, in a state-aware manner.

3 DESIGN

In this paper, we present the design and implementation of Vulcan, a state-aware fuzzing tool for evaluating robustness of wearable applications. We wanted to make Vulcan state-aware to make it more efficient in detecting bugs (i.e., detect more bugs with fewer test inputs). This inherently translates to less time overhead for testing. Moreover, we wanted to target the apps in their *vulnerable* states, i.e., states where they are involved in many concurrent activities such as sensor access and high degree of communication. This is based on prior insight from server-class platforms and distributed systems that failures are correlated with high degree of concurrency [10, 33]. Our overall design and implementation of Vulcan are constrained by the following objectives, which are important for wide adoption of our tool.

(1) Make Vulcan as unintrusive as possible. We want to keep the wearable device and the wearable app (i.e., system under test) unmodified.
(2) We want to define a state model that is widely applicable to most wearable apps. This improves generalizability of Vulcan across apps.
(3) Vulcan should neither require any elevated privilege, nor require source code of the target apps.
(4) Automate as many stages of Vulcan (Figure 1), so that it can be run with little manual effort.

In the subsequent sections, we elaborate on the design of Vulcan and show how each of these objectives is met.

3.1 Overview

In order to fuzz wearable apps with the awareness of their states, Vulcan needs to build a state model for each app (offline phase in Figure 1). At first, a UI testing tool is used to explore different possible UI interactions that a user might have with the target app during normal use. In the process, Vulcan collects logs for parsing later to build the state model. We adopt the widely used Droidbot UI testing tool and run it until a time threshold. Then we terminate the training process to parse the collected logs and find out the intra-device interaction events (through Intents) and the inter-device interaction events (through communication messages). We track how the app behaves on receipt of each of these events and use this information to build the state model of the app (Fig. 3). We also record these events as a way for Vulcan to steer the app to specific states. In Vulcan, we fuzz the app in the states where it is involved in a large number of tasks, implying a high degree of concurrency. This design feature allows us to increase the fault activation rate and thus to reduce the testing time, relative to state-agnostic fuzzers (as experimentally validated in Section 5.4). We analyze the logs and deduplicate the stack traces to identify the unique app crashes.

3.2 State Model

During our initial fuzzing experiments with wearable apps, we found that several apps crashed while accessing sensors. Moreover, sensors often determine the *context* of a wearable app (e.g., location or accelerometer readings). We therefore decided to use sensor activation as a feature to define the state of the wearable. Another unique characteristic of wearable apps is that they are often tightly coupled with their mobile counterpart apps. This means that a wearable app needs a mobile app for interactive or power-hungry operations. After multiple sensor data items are collected at the wearable, they are sent to the mobile, or control commands are sent from the mobile app to the wearable app. This background data or control communication may lead to new bugs not yet seen in the mobile app. Therefore, in our model, a combination of sensor activation and communication activity together determine the state of the app.

As an example, consider a fitness app that helps people keep track of their workout (Figure 3). These apps commonly include two paired apps, installed on the mobile and the wearable device. Before starting the workout, a person uses the mobile to indicate to the app that she is going to start (`START_TRIP`). The same happens once the workout is done; the person uses the mobile app to stop tracking the workout session (`STOP_TRIP`).

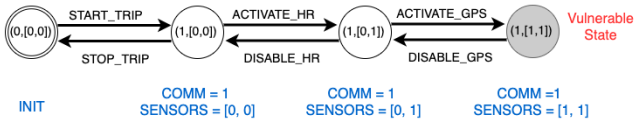


Figure 3: Simplified state transition diagram for a fitness app that has a wearable and a mobile device component. There are two sensors and inter-device communication leading to a state vector of three variables, <*COMM*, *GPS*, *HR*>.

The state of the app is defined by a vector <*COMM*, *GPS*, *HR*>, where `COMM` is the data synchronization activity with values 0 (stopped) or 1 (started), and *GPS* and *HR* are the GPS and heart rate sensors that can have values 0 (deactivated) or 1 (activated). Initially, both sensors and *COMM* are stopped and the app has state <0, 0, 0>. After receiving the `START_TRIP` message, the app goes to state <1, 0, 0>. At this point the user may decide to start monitoring her heart rate and activate the HR sensor (state <1, 0, 1>). If she starts tracking her movement using GPS, the state changes to (<1, 1, 1>). After some time, she may choose to stop either of the sensors. Finally, the app goes back to state <0, 0, 0> after receiving `STOP_TRIP`.

In general, if there are N sensors in the device, then the state of an app can be represented by a tuple <*COMM*, $S_1, S_2, .., S_N$>. Even though wearable devices can have in excess of 20 sensors leading to many possible states in theory, we found that, in practice, any given app uses only a few sensors. Therefore, we restrict the dimensionality of our state definition to include only the sensors used in the app. Further, apps also often activate sensors in groups, e.g., an app with 3 sensors can go from state <0, 0, 0, 0> to <1, 1, 1, 1> without traversing through other combinations of the sensors leading to further reduction of the state space. We found that this relatively coarse definition of states reduces the state-space size but is sufficient for our state-aware fuzzing, which targets *vulnerable* states, and leads to more fault activations.

3.3 Target States for Fuzzing

It is well known in the area of server reliability that higher degree of concurrency can lead to higher failure rates in software [10, 34]. During preliminary experiments, we also observed the negative effect of concurrent activities on the overall reliability of the wearable device. We, therefore, use the degree of concurrency of an app as an indicator of its *vulnerability*[3]. The underlying hypothesis is that by fuzzing the app in its vulnerable states, we can discover more bugs in a shorter time. We show empirical validation of this hypothesis in Section 5.4.

In Vulcan, higher concurrency relates to sensor activity and handling of communication messages at the wearable, each of which runs in a separate thread. Therefore, we define the ***vulnerable states*** of a given app as the states with a certain percentage or greater degree of concurrency than the maximum observed during the training runs. Vulcan steers the app to these states and then fuzzes communication messages or Intents in those states. Our approach has the potential shortcoming that bugs that are not related to concurrency will be missed. One way to mitigate this

[3]We use the terms *vulnerability* and *unreliability* interchangeably in this paper.

problem would be to broaden the definition of vulnerable states to include other factors, such as, the use of deprecated API calls.

3.4 Fuzzing Strategies

Vulcan applies two different fuzzing strategies depending on the state of the app. If the app is in a state that has ongoing communication, i.e., $COMM = 1$, then both intra-device interaction, i.e., Intent fuzzing, and inter-device interaction, i.e., Communication fuzzing, are performed. In contrast, if no communication is ongoing, i.e., $COMM = 0$, only intra-device fuzzing is performed. This is due to the fact that when $COMM = 0$, communication messages from mobile to wearable are silently ignored. Below, we detail each of the two types of fuzzing and the method to automate the process.

3.4.1 Intent Injection. While fuzzing intra-device interaction, we generate *explicit* Intents targeting *Activities* and *Services* within an app. We modify the `Action`, `Data`, and `Extra` fields of Intents based on three pre-defined strategies: *semi-valid* (use combination of known `Action` and `Data` values, but which are incorrect according to the actual usage), *random* (use random strings as `Action` or `Data`), and *empty* (keep one of the fields blank). Vulcan generates *semi-valid* injections following the Android API specification [23]. The Intents are composed using valid combinations of `Action`, `Data`, and `Extra` fields. For instance, using a semi-valid strategy, the fuzzer will generate an `Intent{component=WearAct, act=action.RUN}`, when the app is expecting: `Intent{component= WearAct, act=fitness.TRACK}`.

3.4.2 Communication Fuzzing. We fuzz the communication messages between the mobile and the wearable based on two strategies: *random* (use random message) or *empty* (use *null* value as the message). For example, for a message like `[/getOffDismissed, SomeMessage]`, the fuzzer replaces the message with `[/getOffDis missed, null]` using the empty fuzzing strategy. We did not add semi-valid fuzzing strategy here because we are unable to read the structures of inter-device communication messages due to the lack of source code and the lack of any standard format for inter-device messages. To the best of our knowledge, no other Android testing tool fuzzes inter-device communication messages. We found that communication fuzzing led to some novel failures, including error propagation from a wearable app to a mobile app, and a higher rate of failures, as we discuss in Section 5.4.

3.4.3 Automated Intent Specification Generation. During our initial fuzzing experiments, we randomly generated Intents but we found that a majority of randomly generated Intents were discarded by the Android Runtime, since the target apps only subscribed to specific Intents. To maximize the effectiveness of our semi-valid Intent injection, we wanted to generate the Intents following the Android specification [23]. We found that the Android documentation specifies the valid fields (`Data`, `Action`, `Category`, `Extra`) for each Intent and the possible valid combinations of those fields. According to the purpose of the Intent, the valid type for each of its fields is also specified. For example, an Intent could specify the action of `ACTION_PACKAGE_REPLACED` to broadcast an updated app package. The Intent will have the data field as the name of the package and an extra field called `EXTRA_UID` that indicates a unique integer identification number associated with the updated package. This motivates us to intelligently generate Intents by combining only valid fields instead of randomized values for the fields (e.g., as used in QGJ).

We automate the Intent specification extraction process by designing a text analysis tool. This tool takes the Android specification HTML page [23] as the input and iterates through this page as well as any other possible hyper-linked pages to compose all possible Intents. Two core text extraction techniques are used to extract information on the HTML pages: lexical matching and pattern matching. Lexical matching is used to extract some specific keywords like "Constant Value".

The text analysis tool produces a structured JSON file describing the Intent specifications and this file is used by Vulcan to randomly inject Intents that follow the Android specification. We evaluate the correctness of this automated tool by manually reviewing the generated specifications. The manual process was performed independently by three graduate students to minimize mistakes. We found that the text analysis tool achieved a reasonably high accuracy of 93.5%. We measured accuracy through a conservative calculation whereby if the tool got any of the fields wrong in an Intent, it was taken to have made an error on that Intent. This automation gives Vulcan the capability to dynamically adapt to changes in the Android Intent specification in future API releases.

3.5 Vulcan Components

Vulcan consists of an ensemble of tools that performs various functions at different stages of the offline training or the online testing processes. The overall architecture of our testing tool and the interaction between each of the components is shown in Figure 4. Since Vulcan targets both interaction within the wearable device (inter-process interaction via Intents) and across devices (mobile and wearable), its components are spread across the two devices.

3.5.1 Offline Training. Training runs for each target app are done using the Droidbot UI testing tool [30], which uses the Android Debug Bridge (ADB) to send UI events to the wearable device. Training is initiated with a script on the host computer. It is run for a pre-defined time interval (2 hours for each app), by which time we estimate (and empirically validate for a sampled set of apps) that a majority of the app's states are explored. Further, if during the online operation new states are discovered, we add them into the state model.

Instrumentation. The instrumentation module captures and logs inter-device communication messages that are sent from the mobile to the wearable. These are later used to generate mutated communication messages during the fuzzing phase. This module resides on the mobile device and intercepts the communication to alter messages *before* it is signed by the Android runtime. Such an instrumentation infrastructure does not exist yet on the wearable side. If we were to alter the messages from the wearable to the mobile in transit, or at the mobile end, these messages would fail the Android check and be silently dropped. This is the reason why we cannot mutate messages from the wearable to the mobile device.

Monitor. The Monitor captures log traces from various sources such as `logcat` [24] and `dumpsys` [22]. Through this it evaluates the values of the state variables, such as, for each sensor, whether it is active or not.

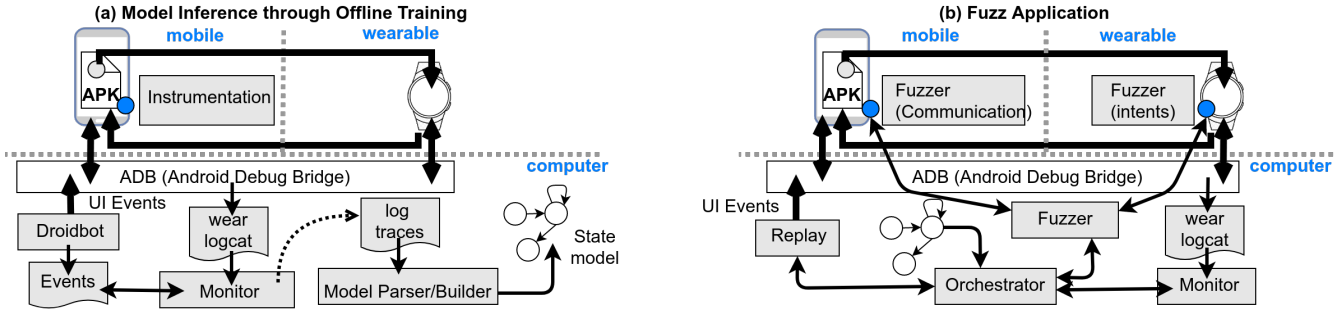


Figure 4: Vulcan architecture. The diagram shows the location for various components and their interaction in each of the two stages: (a) The tools collect information from the log traces to build a model that represents the app; (b) The model is used to guide fuzzing campaigns targeting the wearable app.

Model Builder. It parses the collected logs from the Monitor and the traces of UI events from Droidbot and determines the state model of the wearable app. If multiple UI events occur sequentially without causing any change to the state variables, then Vulcan creates one transition event as the concatenation of the multiple UI events. This design helps in part to contain the explosion of the state space. The sequence of UI events are later used to steer the target app to a vulnerable state.

3.5.2 Online Fuzzing. During online testing, fuzzing happens in an automated manner managed by the Orchestrator, given the state model and the replay traces.

Orchestrator. The Orchestrator is the core of Vulcan and it determines what actions will be performed next. Given a target app and its state model, the orchestrator can run the app in one of two modes–replay mode and fuzzing mode. In the replay mode, the orchestrator guides an app to a target state. This is used to steer the wearable app to a vulnerable state as described earlier. In the fuzzing mode, it directs the Fuzzer to inject inter or intra-device interaction messages accordingly, as described in Section 3.4.

Replay Module. The replay module is responsible for guiding the target app to a given state. For this, it uses the state model of the app and the sequence of UI events needed to drive the app to a target state, both of which have been determined offline and are provided by the Model Builder. This ordered set of UI events are injected into the mobile device using ADB [21]. In case of a crash on the wearable app during fuzzing, the replay module also steers the app to the last valid state before the failure.

Fuzzer. The Fuzzer in Vulcan has two submodules, one for each of the fuzzing strategies described in Section 3.4. Intra-device interaction fuzzing (Intents) is done by a service component in the wearable device, whereas, inter-device communication fuzzing is done using the instrumentation module in the mobile device as described above. Depending on the state of the wearable (i.e., is communication in progress between the mobile and the wearable), one or both of the fuzzing strategies may be applied. In either of these campaigns, if the app state changes while fuzzing, the control goes back to the *Orchestrator* to decide the next fuzzing strategy to execute.

Monitor. As in the offline mode, the Monitor keeps track of current state of the wearable app during the fuzz testing. If during fuzzing, it discovers a new state (which was not observed during the training phase), it updates the state model with this new state and records the events that caused that transition. The new state is not immediately fuzzed during the current iteration, but is considered in future runs if it meets the definition of a vulnerable state.

4 IMPLEMENTATION

We implemented Vulcan using Java (Intent fuzzer), Python (scripts for training and execution), and Javascript (Communication fuzzer). Vulcan does not require access to source code of a target app and can work with binaries. This is crucial for adoption of any tool and is different from many tools in this space, such as ACTEve [3], JPF-Android [42], AW UIAutomator [45]. All of our instrumentation and fuzzing are done without any modifications to the wearable app itself.

Data Collection and State Model Building. The fact that Vulcan is designed to fuzz apps without access to the source code posed a challenge to infer the *state model*. The events relevant to our approach to create the state model are interaction between devices and hardware sensor activation and these are observed using the `adb logcat`, `dumpsys`, and the communication traces collected from the callback functions (instrumentation). The communication between a wearable device and a mobile device is visible using the traces from `Wearable Data Layer` APIs in the Google Play services framework. We collected the log traces of inter-device interactions by enabling logging in Android's `WearableListenerService`. The sensor events are readily available by invoking the `dumpsys` service, through the `adb`. This service provides the current status of the sensor and which process is interacting with it, if it is activated,

Dynamic Binary Instrumentation. In order to fuzz inter-device communication messages, we alter the messages at source (i.e., on the mobile side). This was achieved by dynamically instrumenting the binary of the mobile app and registering a callback for all its Data Layer API calls. The instrumentation module was implemented using the Frida dynamic binary instrumentation framework [4].

Fuzzing Components. In terms of deployment, Vulcan is deployed across both the mobile and the wearable device as shown in Fig. 4. Our inter-device interaction fuzzer is implemented as a Javascript code that is invoked by Frida when the mobile app communicates with the wearable. Our intra-device interaction (Intent) fuzzer was implemented as a separate app. It uses the Android framework specification for generating semi-valid and empty Intents. Currently, these specifications are automatically extracted from Android documentation and are saved as JSON files. We also developed another app for selectively activating and using sensors on the wearable

device, called the Manipulator app. The purpose of this app is to stress the wearable device itself by creating contention for various sensors (Section 5.1).

5 EXPERIMENTS

5.1 Experiment Setup

Sample Set. We evaluated our fuzz tool on the top 100 wearable apps available in the Google Play Store in the Android Wear section. The three predominant categories that these apps fall under are (in order) Health & Fitness, Tools, and Productivity. The criteria for ranking are as per Google's definition of popularity (a combination of factors such as number of installs, number of uninstalls, user ratings, growth trends etc.). We downloaded each app from the Google Play Store and installed on the wearable devices. If the app has a companion app, we installed the companion app on the mobile device. Table 1 shows the detailed information on the selected apps, which combine for a total of 861 activities and 604 services.

Hardware. For the experiments we used 4 Google Nexus 6P smartphones with Android 8.1 paired with 4 different smartwatches (two of Huawei Watch 2 and two of Fossil Gen 4). All the smartwatches have installed Wear OS 2.8 (released in July 2019). For the 100 apps that we tested, we evenly distributed 25 apps to each smartwatch. During the experiments, the mobile was connected to a computer via USB to collect the log traces using `adb`, while the smartwatch was connected to `adb` over Bluetooth.

Setup. None of our evaluations required any modification to the target apps. For our baseline experiments, we ran Monkey and APE for 2 hours on each app. In the case of QGJ, we ran all the four fuzz campaigns presented in [9] targeting all the *Activities* and *Services* components of each app. Finally, for evaluating Vulcan we first ran each app for two hours with Droidbot for state-model construction. After identifying the state models, we fuzz each application in its corresponding *vulnerable states* using the fuzzing strategies described in Sec. 3.4. If the device rebooted, we continued the fuzzing in the remaining *vulnerable* states after the reboot.

5.2 Failure Categories

From our experiments, we found failures to have three types of manifestations, which are all user-visible failures. **ANR:** This condition is triggered when the foreground app stops responding, leading to the manifestation that the system appears frozen. The system generates a log entry containing the message `ANR` (App Not Responding). **App Crash:** This corresponds to an app crash. The system generates a log entry with the message `FATAL EXCEPTION`. **System Reboot:** This is the most severe of the failures and triggers a soft reboot of the Android runtime causing the device to reboot. The system generates a log entry with the message `android.os.DeadSystem Exception`. These manifestations are equivalent to the Catastrophic, Restart, and Abort failures proposed in [28] for the classification of the robustness of server OSes. Another interesting class of errors, silent data corruption, is kept outside the scope of this paper as it needs expert users writing specific data validity checks. We capture the stack trace following a failure message to identify unique failures, by looking for matching stack traces (after removing non-deterministic fields like `pid`). For our results, we only count unique failures after deduplication.

Table 1: Overview of selected apps for the evaluation grouped by category. These are the top 100 apps on the Google Play Store between Mar and Aug 2019.

Category	# Apps	# Act.	# Serv.
Books and references	1	3	1
Communication	7	149	173
Entertainment	7	14	0
Finance	5	69	30
Food and drink	1	7	6
Game arcade	1	2	0
Game puzzle	1	2	2
Health & Fitness	16	160	85
Lifestyle	2	18	1
Maps and navigation	5	24	16
Medical	1	3	1
Music and audio	7	58	57
News and magazines	2	7	7
Personalization	5	62	29
Photography	1	6	4
Productivity	10	82	65
Shopping	2	24	14
Sports	5	25	16
Tools	11	95	51
Travel and local	4	16	11
Weather	6	35	35
	100	861	604

5.3 State-aware Injection Campaigns

One of the primary objectives of Vulcan is to evaluate the effectiveness of a state-aware fuzzing strategy in comparison with state-agnostic fuzzing. To this end, we ran our experiments with fuzzing strategies described in Section 3.5. Based on the state of an app, our fuzzer either alters intra-device interaction messages (Intents) or inter-device interaction messages (control commands or data synchronization messages from the mobile to the wearable). Besides state-aware fuzzing strategies, we also want to evaluate the impact of degree of concurrency on application reliability. We hypothesize that altering global state of the device (how many and which sensors are activated) can lead to different failure manifestations. One approach to evaluate this could be to stress the sensors from within the target app, e.g., querying the same sensors but at a higher rate. However, doing so would require alteration of the target app, and our usage model does not allow this. Therefore, we decided to stress the apps, and thus the device, by activating sensors through an external app, introduced earlier as the "Manipulator app". The Manipulator app either activates the same set of sensors as those in an app (thereby causing contention for those sensors) or it activates the complementary set of sensors (increasing load on the `SensorManager`, the base class that lets apps access a device's sensors). Thus, we ran Vulcan under three scenarios as shown in Table 2. While Expt. I evaluates Vulcan against a state-agnostic fuzzer like Monkey or QGJ, Experiments II and III show the impact of concurrency on the app reliability.

Some of the apps in our sample set did not use any sensor, therefore, Expt III is not applicable to these apps. Due to this, we split the results into two categories: (a) apps that use sensors for which all three experiments are run and (b) apps that do not use sensors

Table 2: Variation of sensor activations across experiments.

Expt.	Device State
I	Not modified outside target app
II	Activate complementary sensors (by the Manipulator app)
III	Activate same sensors as in target app (by the Manipulator app)

for which only experiments I and II are run. We found that despite not using any sensors, these apps still showed a larger number of ANRs and reboots for Expt II compared to Expt I.

Baselines

We compare the results of Vulcan against three state-of-the-art tools for testing Android or Wear OS: QGJ [9], Android Monkey tool [5], and the recently released APE [25], the first two are state-agnostic while the last is state-aware. QGJ, a black box fuzz testing tool, injects malformed Intents to wearable apps based on four campaigns which vary the Intent's input (e.g., *Action*, *Data* or *Extras*). Monkey is a widely used UI fuzzing tool that generates pseudo-random UI events and injects them to the device or an emulator. We assigned an equal probability to each of the possible events in Monkey. Finally, APE, built on top of Monkey, is a GUI injection tool that does dynamic refinement of the statically determined state model to increase the effectiveness of Monkey's testing strategies.

5.4 State-Aware Injection Results

Figure 5 shows the results obtained from our experiment. In general, we found that our state-aware fuzzing with Vulcan generated more crashes than the state-agnostic fuzzers. Moreover, Vulcan is the only one that is able to trigger system reboots in the wearable device. However, Monkey generated more ANRs compared to Vulcan, since it sends events at a higher rate than Vulcan. Vulcan pauses injections after a set and invokes the Garbage Collector to prevent overloading the system. To see the effect of altering the global state of the device, according to Experiments I-III, we look at Tables 3 (for all apps, using sensors and not using sensors) and 4 (only for the apps that use sensors). As we can observe, ANR and system reboots increase when the sensors on the device are stressed. We omit Expt III from Table 3 since it has fewer apps than other experiments and cannot be compared directly. Table 4 depicts the distribution of failure manifestations for all three experiments.

We also found differences in failure manifestations across Experiments I, II, and III, which supports our hypothesis that fuzzing in the vulnerable states (higher concurrency) can trigger new bugs on the wearable app. Although the number of crashes remain the same across the experiments (Table 4), the number of ANRs is much higher for Experiments II and III. This is expected considering the Manipulator app is stressing the `SensorService` on the wearable device. Similarly, number of reboots is higher in Expt II and III compared to Expt I (Tables 3 and 4). This indicates that sensor activation (even through an external app) has a negative effect on overall reliability of the system. Vulcan can automatically guide a wearable app to a vulnerable state and then fuzz at that state, thus increasing the rate at which bugs can be discovered.

Comparison across tools. Figure 6 shows a comparison of unique and overlapping crashes triggered by each of the tools. It can be seen that QGJ and Vulcan has a large degree of overlap though there

Table 3: Failure manifestations for all apps. This indicates that state-aware fuzzing leads to more ANR, crashes, and reboots. For Vulcan, the values are presented in parenthesis as (Intent fuzzing, communication fuzzing).

State	#ANR	#Crashes	#Reboots
Vulcan (Expt. I)	12	44 (39, 5)	3 (3, 0)
Vulcan (Expt. II)	20	45 (40, 5)	12 (12, 0)
QGJ	12	38	0
Monkey	57	17	0
APE	20	15	0

Table 4: Failure manifestations for apps that use sensors. This indicates that state-aware approach can trigger more failures than state-agnostic approaches like QGJ or Monkey. For Vulcan all the failures correspond to the Intent fuzzing strategy.

State	#ANR	#Crashes	#Reboots
Vulcan (Expt. I)	1	10	2
Vulcan (Expt. II)	12	10	3
Vulcan (Expt. III)	9	10	3
QGJ	2	8	0
Monkey	18	5	0
APE	10	0	0

are 8 crashes that were not triggered by QGJ. On the contrary, Monkey and APE have very little overlap among the crashes because they generate UI events very differently and they have no overlap with the Intent fuzzing tools, QGJ and Vulcan. This highlights the importance of using complementary tools for testing Wear OS apps, namely, UI, Intent, and Communication fuzzing tools.

Exception types. To see the distribution of exception types that resulted in the crashes across the experiments, we look at Figure 7. It can be seen that `NullPointerException` dominates as the leading cause of failures, followed by `IllegalArgumentException` and `IllegalState Exception`. This ordering is quite consistent across the four fuzzing-based tools. Previous work [36, 37] has shown that input validation bugs (null pointer and illegal argument exceptions) have been a common category in Android apps over the years. Our results indicate that similar situation persists in the current generation of wearable apps. Most of these crashes can be avoided by proper exception handling code in the apps. IDEs such as Android studio can implement better exception handling. If a system service (e.g., Google Fit in our case) throws an exception, the tools should check absence of exception handling codes and throw a compile time error.

From Tables 3 and 4, we see Monkey triggers a high number of ANRs. We performed an experiment to understand the reason behind this. When Monkey is run with default settings, there is no delay between UI events and this causes an overload of the app. But when we ran Monkey with a delay of 500ms between UI events to make it comparable to that of inter-Intent delay of Vulcan, we observed that the ANRs come down significantly. For example, for the 7 most ANR-prone apps, the number of ANRs with

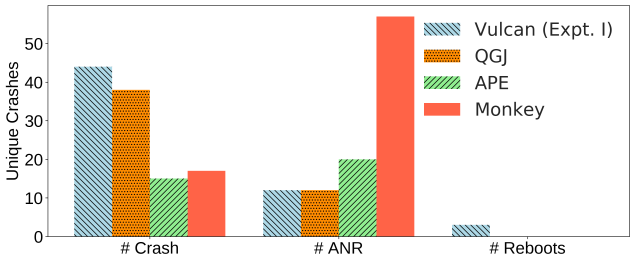


Figure 5: Distribution of failure manifestation for all the tools. Vulcan leads to more crashes and reboots than others.

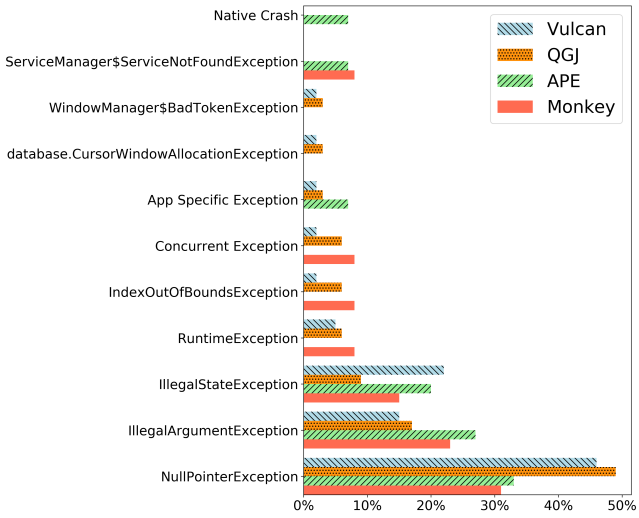


Figure 7: Distribution of Exception Types across runs that resulted in a crash. The NullPointerException dominates across all tools.

the default execution and with the delay went down from 16 to 4 (the number of crashes also incidentally went down, from 12 to 8, while no system reboots were triggered). We also observed that ANRs were rather non-deterministic, unlike crashes and system reboots, and this is understandable because these correspond to individual apps' response to overload conditions, which tend to be variable themselves.

5.5 QGJ vs Vulcan

Efficiency. Here we take a look at the efficiency of triggering the crashes for QGJ versus Vulcan (Expt. I). Efficiency is measured by how many Intents have to be injected to cause a single failure. We replicate the four fuzzing campaigns of QGJ described in [9]. The results are shown in Table 5.

QGJ is able to trigger comparable numbers of unique crashes through Intent fuzzing as Vulcan (Table 3 Exp I). However, its efficiency in triggering the crashes is much lower. Table 5 shows how Vulcan is 5.5X more efficient than QGJ in inducing unique crashes. The efficiency difference comes from the fact that QGJ creates fuzzed Intents without awareness of the validity of the

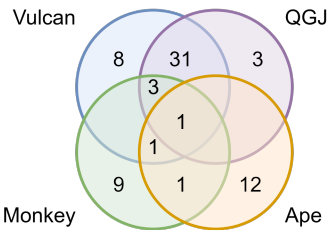


Figure 6: Unique crashes. Comparison of unique crashes across tools after normalizing stack traces.

Table 5: Efficiency of triggering failures between QGJ and Vulcan using Intent fuzzing.

Tool	QGJ	Vulcan
# Intents injected	3,390,010	631,049
# crashes	7,220	1,990
# unique crashes	38	39
# of unique crashes per 100K injected Intents	1.12	6.18 (5.52X)

combination of Intent fields defined in the Android Intent specifications [23]. It tests all combinations of `Action`, `Data`, and Extra fields in an Intent, therefore, a significant number of Intents sent by QGJ are rejected by the Wear OS or triggered duplicated failures. Vulcan, on the other hand, parses and follows the specification to create specific Intents so that far fewer Intents are rejected.

Failure types. Vulcan fuzzes both inter-device and intra-device communications, whereas, QGJ only fuzzes intra-device communications. Due to our state-aware fuzzing, Vulcan is able to identify when there is an ongoing synchronization between the wearable and the mobile device and fuzz the inter-device messages in those states. Vulcan was able to identify 5 failures related to inter-device communications. These failures are mostly due to `IllegalStateException` errors. As we can notice in Figure 7, Vulcan trigerred twice as many `IllegalStateException` failures as QGJ. Some of these failures even propagated to the paired mobile, crashing both the wearable app and the mobile companion app (we provide a root cause analysis in Sec. 7.1).

Deterministic system reboots. In our experiments, QGJ did not trigger any system reboots. By contrast, Vulcan was able to trigger 18 system reboots across 13 apps. This can be explained by the fact that QGJ is a state-agnostic tool, while Vulcan can steer a wearable app towards a vulnerable state before applying the fuzz campaigns. Moreover, from our results, we identified that apps with high concurrency often trigger system reboots with Vulcan. Hence, we decided to further validate this claim by repeating the experiments for the sensor-rich applications from the 13 in total for which we observed system reboots. We focus here on the 4 apps that had the richest use of sensors. Naturally, these achieve a higher degree of concurrency due to the use of sensors. We followed the same procedure as Section 5.4 and for each app we repeated the experiment 5 times. As expected, Vulcan triggered system reboots, but now the system reboots happen deterministically for each trial,

while for the entire set of applications, the reboot trigger happened approximately half the time.

6 SYSTEM REBOOTS

In this section, we show the cases where Vulcan causes the wearable device to reboot and delve into the root causes of these system reboots. We found 18 system reboots across 13 apps, out of the 100 evaluated apps. As part of responsible disclosure, we have shared the failure details with the OS vendor. System reboots are identified by the exception `DeadSystemException` in the Logcat logs, which essentially means that the Android System Server is dead. Android System Server is the middleware to support user level apps such as a browser. It creates most of the core components in Android such as Activity Manager, Package Manager, etc. If the System Server is down, user apps will not function so the whole system has to reboot to restore the normal operations.

6.1 Root Cause Analysis

In the system reboots that Vulcan triggered, the System Server is killed by the Android Watchdog. The Watchdog is a protection mechanism to prevent the wearable from becoming unresponsive; it monitors the system processes in a `while(true)` loop with a fixed delay between iterations. The Watchdog as well as all the components shown in Figure 8 run as threads within the System Server process. If the Watchdog finds a component that is hung for more than 60 seconds (the default timeout value), it will call `Process.killProcess(Process.myPid())` to send a SIGKILL signal. Because the monitored components share the same process id with the Watchdog as well as the System Server, this essentially means killing the System Server, which causes the device to reboot.

By delving into the logs and stack traces of those cases, we categorize those 18 reboots into two groups. 8 reboots are due to blocked lock and 10 other reboots are due to busy waiting. Blocked lock means that a thread is trying to acquire a lock but is blocked because the lock is held by another thread. Busy waiting means a thread is in the waiting status for some other thread to finish, such as a UI thread (`android.ui`) or an I/O thread (`android.io`) for read or write operation.

We give two examples to explain each of the two categories. For **Blocked Lock**, Figure 9 shows the stacktrace where the `Activity Manager` is waiting for the lock (0x05804bd6 in line 266) that is held by thread 82. This wait passes the 60 seconds default timeout value. As a thread created by the System Server, `ActivityManager` shares the same process id with the System Server. There is no way for the Watchdog to just kill the `ActivityManager` but has to kill the System Server process thereby rebooting the system. For **Busy Waiting**, Figure 10 shows an example. Here the `ActivityManager` calls `__epoll_pwait` in line 513, which is a Linux function to wait for an I/O event. This wait expires the 60 second threshold in the Watchdog, so the System Server is also killed thereby rebooting the device.

6.2 Mitigation of System Reboots

Keeping all the core system services as threads in the System Server makes for a simple design. However, it comes with the price of unnecessary reboots of the whole system because of the death of a single service. With increasing number of system services, the reboots will become more frequent and it will also require longer to reinitialize the System Server process.

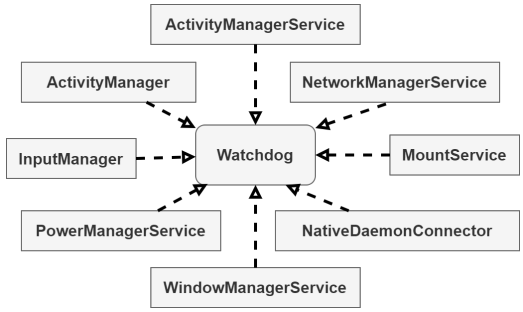


Figure 8: Various system service threads monitored by the Watchdog. The system will reboot if any thread is killed by the Watchdog.

Our solution strategy is to use an Intent buffer to alleviate such resource starvation and thereby, preventing system reboots. In the proposed solution, all the Intents sent from one app to another are stored in the buffer. We then have a **Fetcher process** fetch a single Intent periodically and forward the Intent to the destination(s) after a pre-set delay. When the buffer is full, the system rejects any incoming Intent to keep the device operational thus preventing resource starvation from a burst of Intents. This approach will prevent an attacker app (such as one injecting Intents at in Vulcan) from overwhelming Wear OS.

Since Wear OS is closed source, we cannot modify the framework so we implement a Proof-Of-Concept (POC) solution. We implement an Intent buffer as a middle layer for Intent handing on the target app itself. We empirically found that for a sensor-rich application like the Cardiogram app, a delay of 2.5 seconds between Intents eliminated *all* system reboots. Note that this delay is a pessimistic estimate (assuming an already busy device) and we can incrementally raise the delay from a low value (few milliseconds) after every Intent delivery. Moreover, (trusted) system Intents or signed Intents can bypass our buffer and can be delivered immediately. Such a strategy will have negligible delay for system Intents and a relatively high delay for untrusted user-level Intents.

Next, we discuss a small user study to quantify the effect on usability due to the delay introduced by the Fetcher (we used the default value of 2.5 seconds). First, we developed a (trivial) wearable app, which communicates with a third-party calendar app [6] using Intents. Our app can interact with the calendar app using two approaches: either using our POC solution or the standard mechanism. Then, we recruited 15 random people from our department to use our app to schedule events on the smartwatch. Each person repeated the process three times for each approach, run as a blind study. Finally, we asked the users to evaluate their experience with the app. As Table 6 shows, most of the users experienced no difference (60.0%) or little difference (33.3%) while using the app with our POC solution versus the standard mechanism. In contrast, only (6.7%) noted a significant difference between the two approaches. This shows that an appropriately tuned delay for our Intent buffer solution can potentially mitigate system reboot attacks, while not affecting usability of wearable apps. A full validation will require

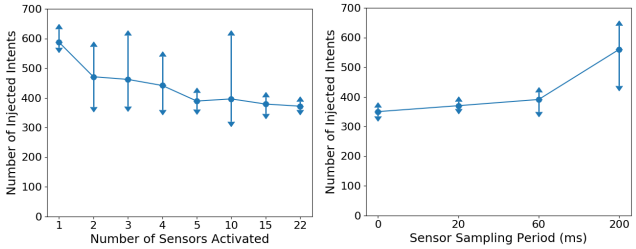
```
259 "ActivityManager" prio=5 tid=11 Blocked
260   | group="main" sCount=1 dsCount=0 flags=1 obj=0x12f01178 self=0x
9cd59600
261   | sysTid=685 nice=-2 cgrp=default sched=0/0 handle=0x91a35970
262   | state=S schedstat=( 0 0 0 ) utm=381 stm=336 core=2 HZ=100
263   | stack=0x91933000-0x91935000 stackSize=1038KB
264   | held mutexes=
265   at com.android.server.am.TaskChangeNotificationController.forAll
RemoteListeners(TaskChangeNotificationController.java:245)
266   - waiting to lock <0x05804bd6> (a com.android.server.am.Activity
ManagerService) held by thread 82
```

Figure 9: Example stacktrace of the blocked lock system reboots. `ActivityManager` is waiting for the lock at line 266.

```
506 "ActivityManager:kill" prio=5 tid=13 Native
507   | group="main" sCount=1 dsCount=0 flags=1 obj=0x13180f08 self=0x9
eeb8a00
508   | sysTid=16672 nice=10 cgrp=default sched=0/0 handle=0x99984970
509   | state=S schedstat=( 171525879 6433427327 603 ) utm=5 stm=12
core=1 HZ=100
510   | stack=0x99882000-0x99884000 stackSize=1038KB
511   | held mutexes=
512   kernel: (couldn't read /proc/self/task/16672/stack)
513   native: #00 pc 000492b4  /system/lib/libc.so (__epoll_pwait+20)
514   native: #01 pc 0001b3c5  /system/lib/libc.so (epoll_pwait+60)
```

Figure 10: Example stacktrace of the busy waiting system reboots. `ActivityManager` calls a I/O wait function at line 513.

a larger user study with a greater and a diverse set of wearable activities.



(a) Varying # of sensors activated **(b) Varying sensor sampling period**

Figure 11: Effect of load of sensors on system reboots. Fewer Intents are needed to trigger a system reboot as (a) the number of sensors being activated increases or (b) the sensor sampling period decreases. The bars represent the range from the min to the max values.

Table 6: User experience reported while using the smartwatch with our POC solution.

User Experience	% Response
No difference	60.0 %
Little difference	33.3 %
Significant difference	6.7 %

Comparison with prior research on Android system reboot. In prior work [26], it was shown that Android suffers from coarse-grained locking in its system services, whereby, seemingly innocuous apps can cause starvation of system resources by acquiring these locks in a loop. This would cause the Android device to reboot. We tried to replicate this failure mode on Wear OS and could not. On investigating, we find that their attacker app, designed for Android 5, relies on successfully receiving two Broadcast Intents: `PACKAGE_REMOVED`, and `ACTION_PACKAGE_REPLACED`. But, since Android 8.0, receiving Broadcast Intents has been severely restricted for user-level apps and these two specifically can only be received by system apps. In contrast, our Intent fuzzer is a user-level app and can cause the system reboots without system-level privileges.

The mitigation proposed in [26] is finer-grained locking in Android system services, a smarter Watchdog (that does not automatically kill a non-responsive process), and restricting the number of resources (e.g., sensors) an app can acquire. All their defenses require modification of the Android framework, whereas, our solution can be implemented on a per-app basis (an Intent buffer at the receiving app), or, at the framework level with vendor cooperation.

6.3 Effect of Load on System Reboots

Sensor activity. Our central hypothesis is that the reliability of the software system is compromised by the degree of concurrency. Here, we quantify this effect by varying the number of sensors that are activated and measuring how many Intents are needed to trigger a device reboot. For the experiment, we choose a popular Health & Fitness app, the Cardiogram app [12], and one that has been implicated in our earlier experiments with system reboots. We vary the number of sensors activated (by the Manipulator app) from 0 to the maximum available in the device, 22 sensors (15 hardware sensors and 7 software sensors), while concurrently injecting fuzzed Intents into the Cardiogram app. In each trial, we keep the sampling period of each sensor fixed at once every 60 ms. We repeated the experiment 5 times for each number of activated sensors. Importantly, device reboots were triggered deterministically in every single trial. As shown in Figure 11(a), as the number of sensors activated increases, we need fewer number of Intents to trigger a system reboot. This number drops sharply for the first 5 sensors, and then drops slowly. This result indicates that with increasing concurrency, the vulnerability of the system increases to the most catastrophic of failures, system reboots.

Sensor sampling period. In this experiment, we show the effect of changing sensor sampling periods on the ease of triggering system reboots. We inject Intents to the Cardiogram app while concurrently using our Manipulator app to sample all 22 sensors in synch, with varying periodicities. We show the result in Figure 11(b). For each sensor sampling period, we run 5 trials and in each trial, we measure the number of injected Intents until the wearable device reboots. The Android Developer document [18] specifies four different sampling periods and we use all of them in our experiments. In every single trial, the device eventually rebooted (approximately in 20–30 minutes), emphasizing the deterministic nature of this failure. The result can be explained by the fact that sensor sampling consumes system resources. Thus, the faster the sensors are sampled, the more resources they consume and therefore, it requires injecting fewer Intents to trigger system reboots.

7 LESSONS AND THREATS

7.1 Failure Case Studies

Our empirical evaluation of 100 popular Wear OS apps helped us understand several aspects of wearable app reliability. In this section, we present various failure categories that highlight some

nuances about the Wear OS ecosystem and provide motivation for future research.

Android-Wear OS Code Transfer. Among the crashes observed, we found that every time our fuzzer injected a `KEYCODE_SEARCH` event, the wearable app crashed. This is a widespread problem—among the 100 tested apps, 95 crashed when this event was injected. We found empirically that no other key event triggered a similar vulnerability. On Android, the `KEYCODE_SEARCH` event is used to launch a text search inside the app (apps can also silently ignore the event if it has not implemented this feature). However, in the case of Wear OS, when the event is injected, the apps crashed with an `IllegalStateException`. Digging deeper we found that Wear OS, when inheriting code from Android, removed the `SearchManager` service class, assuming probably that text searches are not suited for the form factor of wearables, but did not drop the corresponding Intent. This kind of behavior can be seized on by a malicious app (potentially with system-level privileges) to crash other apps. The root problem can be handled at the Wear OS level by simply discarding the search event in the base Activities class (`WearableActivity`) instead of trying to launch `SearchManager`.

Error Propagation. As noted in Section 5.5, Vulcan found several vulnerabilities related to ongoing synchronization between mobile and wearable (refer Table 3). It is noteworthy that some of these failures even propagated back to the mobile device, crashing not only the wearable app but also its mobile counterpart. For example, in our experiments, the crash of a wearable navigation app triggers a communication from the wearable to the mobile with the `Exception` object However, this causes a `NullPointerException` in the mobile app when it tries to invoke a method `getMessage`. This exception is caused by a mismatch of the serialized object sent by the wearable and that expected by the mobile. The wearable sends the exception wrapped as a `GSON` object, while the mobile device expects a `Throwable` object. This can be handled better by strongly typing the data being shared between devices and throwing a compile-time error in case of type mismatch.

Watchdog and Resource Starvation. Our results in Section 6.1 show that it is possible to reboot a wearable device without any root privilege. The primary reason for this is resource starvation triggered by high degrees of concurrent activities, which cannot be handled by the System Server. We have demonstrated how an intent-buffer can alleviate such resource starvation and can mitigate the problem of system reboots in Section 6.2.

7.2 Threats to Validity.

The experiments presented in this paper have a few shortcomings that may bias our observations. First, our study is based on two different smartwatch models each of which can have individual vendor-specific customizations. Second, our inter-device communication fuzzing is currently one-directional, for messages going from the mobile to the wearable. This is primarily to keep the wearable device unaltered and because the injection tool Frida only exists for Android. Third, even though we selected popular apps from different categories, our sample size of 100 can be considered small. However, this is a shortcoming of most fuzzing studies which are usually time consuming. Even with 100 apps, Vulcan tested more than 1,400 components and took more than two weeks for the experiments of Table 3, running on 4 devices in parallel.

8 RELATED WORK

Stateful Fuzzers. Most of the previous work on stateful fuzzing has been focused on communication protocols, such as SIP or FTP [8, 15, 17, 20]. A common approach here is to infer a finite state machine by analyzing the network traffic. Then, this model is used to guide the black-box fuzz testing of the communication messages, rather than the intra-protocol interactions.

Android. Since its release in 2008, there have been numerous studies on the reliability of Android OS. These works vary from random fuzzing testing tools based on UI events and system events [35, 44], fuzzers focused on Android IPC [11, 36, 39], model-based testing tools based on static analysis [2] or specialized approach based on concolic testing [3]. Previous works based on model representations of the apps, focused on GUI navigation [2, 14, 25, 30, 38, 40, 47], rather than a stateful approach of communication and sensors as in Vulcan.

Wear OS. Research on Wear OS focuses on the performance and the reliability of the OS itself [31–33, 46]. They found inefficiencies in the OS because of deficiencies in the design. Some work found unreliability of Wear OS apps for health monitoring based on physical context such as mobility [19, 43]. However, those design flaws were never tied to any vulnerabilities that made the Wear OS prone to system reboots. Recent studies have addressed the need for testing tools for Wear OS ecosystem [9, 25, 45]. Barsallo Yi *et al.* [9] proposed QGJ, a testing tool for Wear OS that creates four campaigns to inject faulty Intents to the wearable. These campaigns are less efficient than the strategies of Vulcan. Being state-agnostic, it is unable to trigger system reboots. To the best of our knowledge, Vulcan is the first tool that fuzz tests Wear OS using a stateful approach.

9 CONCLUSION

We presented the design and implementation of Vulcan, a state-aware fuzzing tool for wearable apps. Vulcan can automatically build a state model for an app from its logs and steer the app to specific states by replaying the traces. It then launches state-aware fuzzing on the wearable app targeting both inter-device communication and intra-device Intents at the states with high degrees of concurrent activities. State-aware fuzzing leads to more app crashes compared to stateless fuzzing. As the most worrisome result, it is possible to predictably reboot a wearable device from a user app, with no system-level or root privileges, by targeting specific states. We provide a proof-of-concept solution to mitigate the system reboots that performs rate control at an intermediate layer between the source and the destination apps. Lessons for improving the wearable ecosystem are better exception handling, type checking of inter-device communication messages, and diagnosing and terminating components that starve sensor resources. Our future work will focus on automatic identification of the minimum working example for triggering failures and improvements to the exception handling mechanism through static and dynamic techniques.

REFERENCES

[1] 2019. Vulcan: A Wearable App Fuzzing Tool. https://github.com/purdue-dcsl/vulcan/

[2] Domenico Amalfitano, Anna Rita Fasolino, Porfirio Tramontana, Salvatore De Carmine, and Atif M. Memon. 2012. Using GUI Ripping for Automated Testing of Android Applications. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering* (Essen, Germany) *(ASE 2012)*. ACM, New York, NY, USA, 258–261. https://doi.org/10.1145/2351676.2351717

[3] Saswat Anand, Mayur Naik, Mary Jean Harrold, and Hongseok Yang. 2012. Automated concolic testing of smartphone apps. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. ACM, 59.

[4] Ole Andre. 2018. Frida. https://www.frida.re

[5] Android. 2017. UI/Application Exerciser Monkey. https://developer.android.com/studio/test/monkey.html

[6] appfour. 2019. *Calendar for Wear OS (Android Wear)*. https://play.google.com/store/apps/details?id=com.appfour.wearcalendar

[7] Tanzirul Azim and Iulian Neamtiu. 2013. Targeted and depth-first exploration for systematic testing of android apps. In *Acm Sigplan Notices*, Vol. 48. ACM, 641–660.

[8] Greg Banks, Marco Cova, Viktoria Felmetsger, Kevin Almeroth, Richard Kemmerer, and Giovanni Vigna. 2006. SNOOZE: toward a Stateful NetwOrk prOtocol fuzZEr. In *International Conference on Information Security*. Springer, 343–358.

[9] Edgardo Barsallo Yi, Amiya K Maji, and Saurabh Bagchi. 2018. How Reliable is my Wearable: A Fuzz Testing-based Study. In *In Proceedings of the 48th IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 410–417.

[10] Francesco A Bianchi, Mauro Pezzè, and Valerio Terragni. 2017. Reproducing concurrency failures from crash stacks. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM, 705–716.

[11] Jesse Burns. 2012. Intent Fuzzer. https://www.nccgroup.trust/us/about-us/resources/intent-fuzzer

[12] Inc Cardiogram. 2019. *Cardiogram: Wear OS, Fitbit, Garmin, Android Wear*. https://play.google.com/store/apps/details?id=com.cardiogram.v1&hl=en_US

[13] Erika Chin, Adrienne Porter Felt, Kate Greenwood, and David Wagner. 2011. Analyzing inter-application communication in Android. In *Mobisys*. ACM, 239–252.

[14] Wontae Choi, George Necula, and Koushik Sen. 2013. Guided gui testing of android apps with minimal restart and approximate learning. In *Acm Sigplan Notices*, Vol. 48. ACM, 623–640.

[15] Paolo Milani Comparetti, Gilbert Wondracek, Christopher Kruegel, and Engin Kirda. 2009. Prospex: Protocol specification extraction. In *Security and Privacy, 2009 30th IEEE Symposium on*. IEEE, 110–125.

[16] E Connolly, A Faaborg, H Raffle, and B Ryskamp. 2014. Designing for wearables. *Google I/O* (2014).

[17] Joeri De Ruiter and Erik Poll. 2015. Protocol State Fuzzing of TLS Implementations.. In *USENIX Security Symposium*. 193–206.

[18] Android Developers. [n.d.]. *Sensors Overview*. https://developer.android.com/guide/topics/sensors/sensors_overview.html

[19] Cesar Garcia-Perez, Almudena Diaz-Zayas, Alvaro Rios, Pedro Merino, Kostas Katsalis, Chia-Yu Chang, Shahab Shariat, Navid Nikaein, Pilar Rodriguez, and Donal Morris. 2017. Improving the efficiency and reliability of wearable based mobile eHealth applications. *Pervasive and Mobile Computing* 40 (2017), 674–691.

[20] Hugo Gascon, Christian Wressnegger, Fabian Yamaguchi, Daniel Arp, and Konrad Rieck. 2015. PULSAR: stateful black-box fuzzing of proprietary network protocols. In *International Conference on Security and Privacy in Communication Systems*. Springer, 330–347.

[21] Google. 2017. Android Debug Bridge. https://developer.android.com/studio/command-line/adb

[22] Google. 2019. Android Developers. Dumpsys. https://developer.android.com/studio/command-line/dumpsys

[23] Google. 2019. Android Developers. Intent Specification. https://developer.android.com/reference/android/content/Intent

[24] Google. 2019. Android Developers. Logcat. https://developer.android.com/studio/command-line/logcat

[25] Tianxiao Gu, Chengnian Sun, Xiaoxing Ma, Chun Cao, Chang Xu, Yuan Yao, Qirun Zhang, Jian Lu, and Zhendong Su. 2019. Practical GUI testing of Android applications via model abstraction and refinement. In *Proceedings of the 41st International Conference on Software Engineering*. IEEE Press, 269–280.

[26] Heqing Huang, Sencun Zhu, Kai Chen, and Peng Liu. 2015. From System Services Freezing to System Server Shutdown in Android: All You Need Is a Loop in an App. In *ACM CCS*. 1236–1247.

[27] Antonio Ken Iannillo, Roberto Natella, Domenico Cotroneo, and Cristina Nita-Rotaru. 2017. Chizpurfle: A Gray-Box Android Fuzzer for Vendor Service Customizations. In *ISSRE*.

[28] Philip Koopman, John Sung, Christopher Dingman, Daniel Siewiorek, and Ted Marz. 1997. Comparing operating systems using robustness benchmarks. In *Proceedings of SRDS'97: 16th IEEE Symposium on Reliable Distributed Systems*. IEEE, 72–79.

[29] Joshua LeVasseur, Volkmar Uhlig, Jan Stoess, and Stefan Götz. 2004. Unmodified Device Driver Reuse and Improved System Dependability via Virtual Machines.. In *OSDI*, Vol. 4. 17–30.

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

[31] Renju Liu, Lintong Jiang, Ningzhe Jiang, and Felix Xiaozhu Lin. 2015. Anatomizing system activities on interactive wearable devices. In *APSys*. 1–7.

[32] Renju Liu and Felix Xiaozhu Lin. 2016. Understanding the characteristics of android wear os. In *Mobisys*. 151–164.

[33] Xing Liu, Tianyu Chen, Feng Qian, Zhixiu Guo, Felix Xiaozhu Lin, Xiaofeng Wang, and Kai Chen. 2017. Characterizing Smartwatch Usage in the Wild. In *Mobisys*. 385–398.

[34] Brandon Lucia and Luis Ceze. 2013. Cooperative empirical failure avoidance for multithreaded programs. *ACM SIGPLAN Notices* 48, 4 (2013), 39–50.

[35] Aravind Machiry, Rohan Tahiliani, and Mayur Naik. 2013. Dynodroid: An input generation system for android apps. In *FSE*. 224–234.

[36] Amiya K Maji, Fahad A Arshad, Saurabh Bagchi, and Jan S Rellermeyer. 2012. An empirical study of the robustness of inter-component communication in Android. In *DSN*. 1–12.

[37] Amiya Kumar Maji, Kangli Hao, Salmin Sultana, and Saurabh Bagchi. 2010. Characterizing failures in mobile oses: A case study with android and symbian. In *Software Reliability Engineering (ISSRE), 2010 IEEE 21st International Symposium on*. IEEE, 249–258.

[38] Ke Mao, Mark Harman, and Yue Jia. 2016. Sapienz: multi-objective automated testing for Android applications. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*. ACM, 94–105.

[39] Raimondas Sasnauskas and John Regehr. 2014. Intent fuzzer: crafting intents of death. In *WODA and PERTEA*. 1–5.

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

[41] Michael M Swift, Muthukaruppan Annamalai, Brian N Bershad, and Henry M Levy. 2006. Recovering device drivers. *ACM Transactions on Computer Systems (TOCS)* 24, 4 (2006), 333–360.

[42] Heila van der Merwe, Brink van der Merwe, and Willem Visser. 2012. Verifying android applications using Java PathFinder. *ACM SIGSOFT Software Engineering Notes* 37, 6 (2012), 1–5.

[43] Naixing Wang, Edgardo Barsallo Yi, and Saurabh Bagchi. 2017. On reliability of Android wearable health devices. *arXiv preprint arXiv:1706.09247* (2017), 1–2.

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

[45] Hailong Zhang and Atanas Rountev. 2017. Analysis and Testing of Notifications in Android Wear Applications. In *Proceedings of the 39th International Conference on Software Engineering* (Buenos Aires, Argentina) *(ICSE '17)*. IEEE Press, 347–357. https://doi.org/10.1109/ICSE.2017.39

[46] Hailong Zhang, Haowei Wu, and Atanas Rountev. 2018. Detection of energy inefficiencies in android wear watch faces. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 691–702.

[47] Haibing Zheng, Dengfeng Li, Beihai Liang, Xia Zeng, Wujie Zheng, Yuetang Deng, Wing Lam, Wei Yang, and Tao Xie. 2017. Automated Test Input Generation for Android: Towards Getting There in an Industrial Case. In *Proceedings of the 39th International Conference on Software Engineering: Software Engineering in Practice Track* (Buenos Aires, Argentina) *(ICSE-SEIP '17)*. IEEE Press, 253–262. https://doi.org/10.1109/ICSE-SEIP.2017.32