CSED601
Dependable Computing
Lecture 16

Jong Kim

Dept. of CSE

POSTECH

SW Reliability Paper

- “How Reliable is my Wearable: A Fuzz Testing based Study” DSN 18
- Related Issues
  - Fault injection
  - Fuzzing

Software Fault Injection

Kalynnda Berens

Science Applications International Corporation
NASA Glenn Research Center

What is Software Fault Injection?

- A testing technique that aids in understanding how software behaves when stressed in unusual ways.
- A *product*-based assurance technique.
- Variations in the technique allow it to be applied to many types of software and for different purposes.

How does SFI work?

- *Legal* permutations or faults are input at interfaces (external and/or internal).
- Outputs show whether the injected fault propagates through the software.
- Requires *instrumentation* (software code) to observe the propagation process.

Uses for Software Fault Injection

- Finding defects in software
- Robustness Testing
- COTS Validation/Determining failure modes
- Safety Verification
- Security Assessment
- Software Testability Analysis

SFI Examples

- Operating System Validation
  - Ballista (CM) – Linux and VxWorks robustness
  - WindowsNT
- Network Security
  - NCSA httpd server
- Safety
  - Advanced Automatic Train Control system
  - Magneto Stereoaxis System

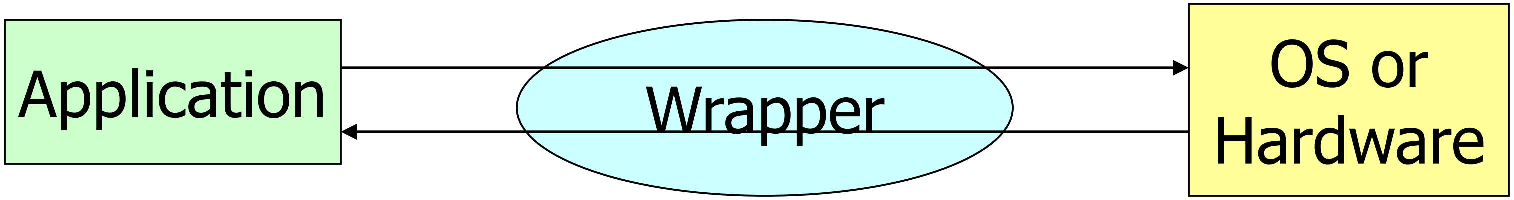
SFI can be used with or without source code

SFI without Source Code

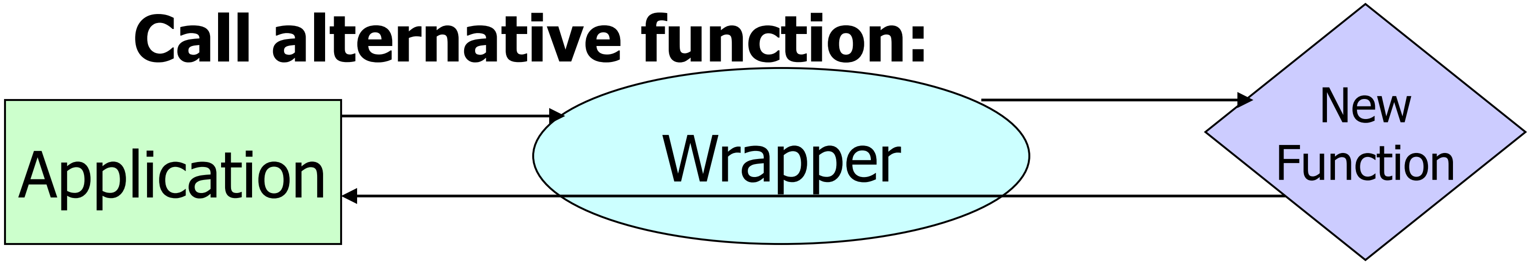
- Create software wrapper for COTS functions and other interfaces
- “Trick” OS to call wrapper functions first
- Software under test usually run in debug mode
- Wrapper can be used
  - Pass through for baselining response
  - Call alternative function
  - Call original function but change result

SFI wrapper operations

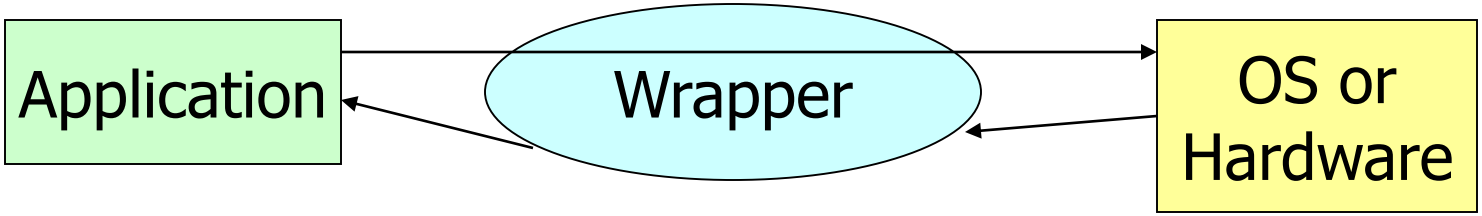
Pass through wrapper:



Call alternative function:



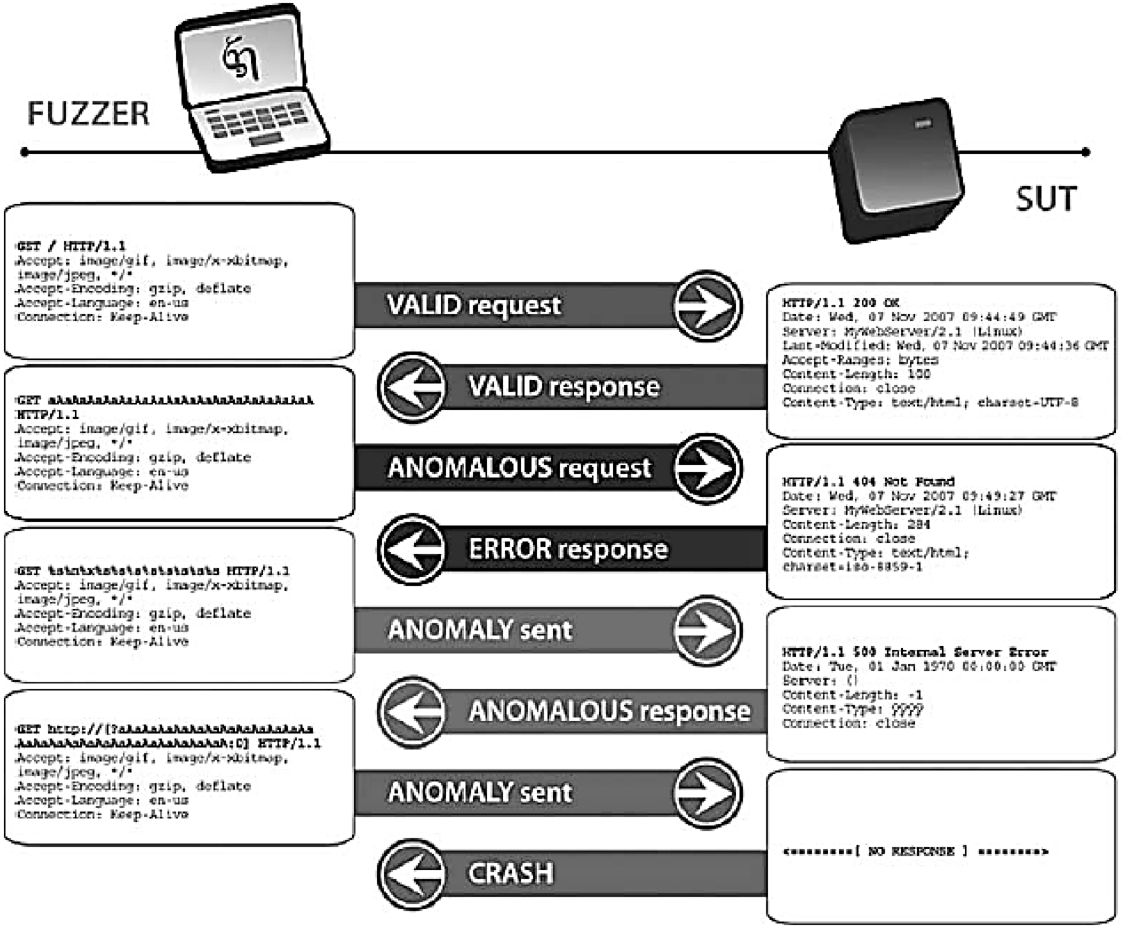
Call original function but change result:



Fuzz testing

- Fuzz testing is a negative software testing method that feeds malformed and unexpected input data to a program, device, or system with the purpose of finding security-related defects, or any critical flaws leading to denial of service, degradation of service, or other undesired behavior (A. Takanen et al, Fuzzing for Software Security Testing and Quality Assurance, 2008)
- Programs and frameworks that are used to create fuzz tests or perform fuzz testing are commonly called **fuzzers.**

Fuzzing process



```
GET / HTTP/1.1
Accept: image/gif, image/x-xbitmap,
image/jpeg, */*
Accept-Encoding: gzip, deflate
Accept-Language: en-us
Connection: Keep-Alive
```

```
HTTP/1.1 200 OK
Date: Wed, 07 Nov 2007 09:44:49 GMT
Server: MyWebServer/2.1 (Linux)
Last-Modified: Wed, 07 Nov 2007 09:44:36 GMT
Accept-Ranges: bytes
Content-Length: 100
Connection: close
Content-Type: text/html; charset-UTF-8
```

```
GET aAaAaAaAaAaAaAaAaAaAaAaAaAaAaAaA
HTTP/1.1
Accept: image/gif, image/x-xbitmap,
image/jpeg, */*
Accept-Encoding: gzip, deflate
Accept-Language: en-us
Connection: Keep-Alive
```

```
HTTP/1.1 404 Not Found
Date: Wed, 07 Nov 2007 09:49:27 GMT
Server: MyWebServer/2.1 (Linux)
Content-Length: 284
Connection: close
Content-Type: text/html;
charset=iso-8859-1
```

```
GET %s%n%x%s%s%s%s%s%s%s%s%s HTTP/1.1
Accept: image/gif, image/x-xbitmap,
image/jpeg, */*
Accept-Encoding: gzip, deflate
Accept-Language: en-us
Connection: Keep-Alive
```

```
HTTP/1.1 500 Internal Server Error
Date: Tue, 01 Jan 1970 00:00:00 GMT
Server: ()
Content-Length: -1
Content-Type: ????
Connection: close
```

```
GET http://[?aAaAaAaAaAaAaAaAaAaAaAaAaA
AaAaAaAaAaAaAaAaAaAaAaAaAaAaA:0] HTTP/1.1
Accept: image/gif, image/x-xbitmap,
image/jpeg, */*
Accept-Encoding: gzip, deflate
Accept-Language: en-us
Connection: Keep-Alive
```

```
<=========[ NO RESPONSE ] ========>
```

Fuzzing approaches

- **Generic:** crude, random corruption of valid data without any regard to the data format.
- **Pattern-based:** modify random data to conform to particular patterns. For example, byte values alternating between a value in the ASCII range and zero to “look like” Unicode.
- **Intelligent:** uses semi-valid data (that may pass a parser/sanity checker’s initial line of defense); requires understanding the underlying data format. For example, fuzzing the compression ratio for image formats, or fuzz PDF header or cross-reference table values.
- **Large Volume:** fuzz tests at large scale. The Microsoft Security Development Lifecycle methodology recommends a minimum of 100,000 data fuzzed files.
- **Exploit variant:** vary a known exploitative input to take advantage of the same attack vector with a different input; good for evaluating the quality of a security patch.

Android Wear Fuzzing – DSN 18

Introduction

- Wearable devices
  - Types: smart glasses, smart watches, hearables, fitness and health trackers, smart jewelry, and smart clothing
  - The number of connected wearable devices worldwide:
    - estimated to be 325 million at the end of 2016
    - expected to grow to over 830 million in 2020
- Motivation
  - Study on vulnerabilities
  - No study on failure mechanisms and propagation
  - New challenges exist
    - Limited display area, limited computing power, limited volatile and non-volatile memory, non-conventional shape of the devices, abundance of sensor data, complex communication patterns of the apps, and limited battery size

Apps on Android Wear (AW)

- Fact on AW
  - Shares much of the codebase with Android OS
  - Follows Androids conventional programming paradigm:
    - written in Java,
    - compiled ahead-of-time,
    - executed atop the managed Android Runtime
- Major differences between traditional Android apps and Wear apps.
  - Due to the smaller display area and rich sensor data in AW
  - The user activities supported is much more limited
  - Wear apps
    - run in the background
    - communicate with users using notifications
    - typically controlled by a companion app on the smartphone

Reliability of AW

- Through three aspects
  - Exception Types:
    - What are the key differences between traditional Android apps and Wear apps in terms of exception handling?
    - Are the relative proportions of manifestations of uncaught exceptions (such as, app crash or system reboot) similar?
  - Failures across Applications:
    - What are the differences between failure manifestations across application types?
    - Are health/fitness apps more or less robust than other apps?
  - Robustness:
    - How well do Wear apps handle unexpected interactive user inputs?
    - Can a user-level process crash the system?

Research Approach

- Tool: Qui-Gon Jinn (QGJ)
  - Two components
    - QGJ-Master: An Android app for sending inter-component communication messages (intents) to targeted applications
    - QGJ-UI: A tool based on Android Monkey fuzzer - which can send malformed UI events to the wearable device
  - Method
    - Systematically inject a large number of mutated, synthetic intents or mutated UI events to a selection of 45 popular Wear apps.
    - Then analyze the system logs to find how well these applications handled the mutated intents or events

Key Findings

- Distribution of exception types does indeed differ between Android and Wear apps.
- Across application types, built-in apps showed more failures compared to third-party apps.
- Several times during the experiments, the Wear device rebooted in response to malformed intents – due to escalation of multiple errors.
- AW apps offer significant scope for improvement of input validation.

Android

- Android programming model
  - Based on passing intent messages for communication within or across applications
  - An intent can been seen as a passive data structure with an abstract description of an operation to be performed.
  - Intent itself represents a message including the operation to be performed and the data needed to perform that operation.
  - Example:

    {ACTION_EDIT, content://contacts/people/1}

Android

- Android application components relevant to this study
  - Activity: the entry point for interacting with the user. One app can invoke an activity in another app, if permissions are granted.
  - Service: a component that runs in the background to perform long-running operations or to perform work for remote processes. A service does not provide a user interface.
  - Broadcast receiver: a component that enables the system to deliver events to the app outside of a regular user flow, allowing the app to respond to system-wide broadcast announcements.

Android Wear (AW)

- Fact:
  - the version of Android OS designed for smartwatches and other wearable devices.
  - unlike Android, not completely open source
  - based on Linux, and follows the same programming paradigm as Android OS, with some significant differences
    - require minimal human interaction (micro transactions).
    - user interface (UI) is designed to be the least attention seeking to the user, by showing minimal information and centered on notifications, watch faces, native applications and voice commands.
    - more services driven in contrast to Android applications
    - take advantage of context-awareness by sensing information from hardware and software sensors equipped on the devices

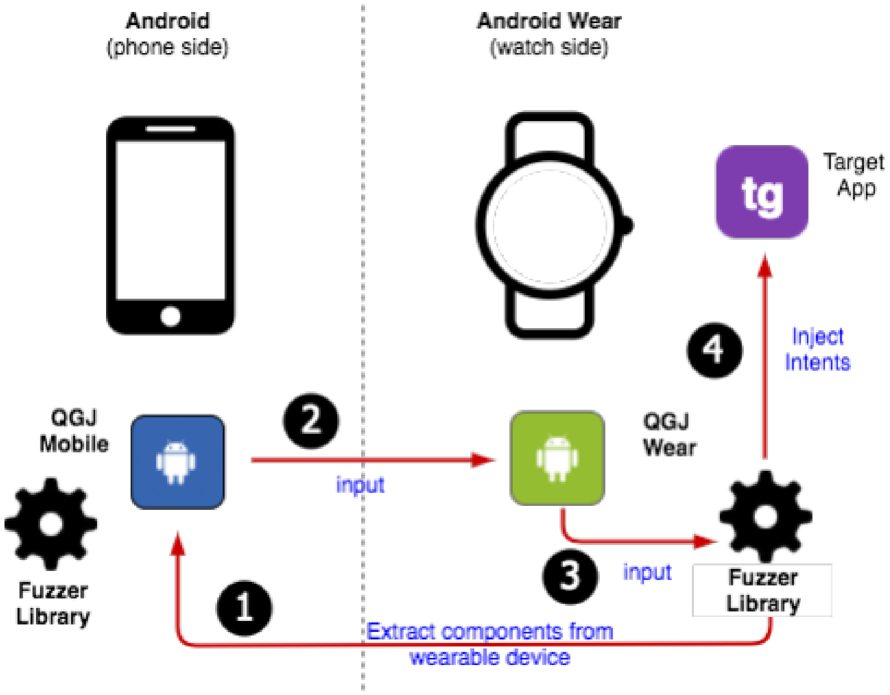
Tool Design

- Fact
  - Followed the architecture of Jar Jar Binks (JJB)
  - JJB is an Android robustness testing tool
    - exploits IPC on Android
    - can fuzz a single component or a group of components registered in a device
    - supports fuzz injection of Activities, Services, and Broadcast Receiver components
  - QGJ extends the JJB capabilities to support Android Wear
  - needs to be installed on two paired devices
  - inject randomly generated intents to both paired devices.
  - consists of three main components: a mobile application (QGJ Mobile), a wear application (QGJ Wear), and a fuzzer library

Three Components of QGJ

- QGJ Mobile: Android application
  - runs on the mobile and offers a UI to interact with the fuzzer
  - The user can choose the component type to fuzz and target device
- QGJ Wear. Android Wear application
  - executes on the wearable. It communicates with the mobile app using the Android Wear MessageAPI and DataAPI.
- Fuzzer Library. Java library
  - contains the main functions needed to inject intents on the target device.

Operational Workflow of QGJ



(1) The QGJ Mobile app retrieves automatically all the components (Activities, Services or BroadcasterReceiver) from the installed app on the Android wearable.

(2) Next, from the Android device, we choose a target application and the fuzzing campaign to use. Then the Android phone communicates with the wearable using the AW Message API.

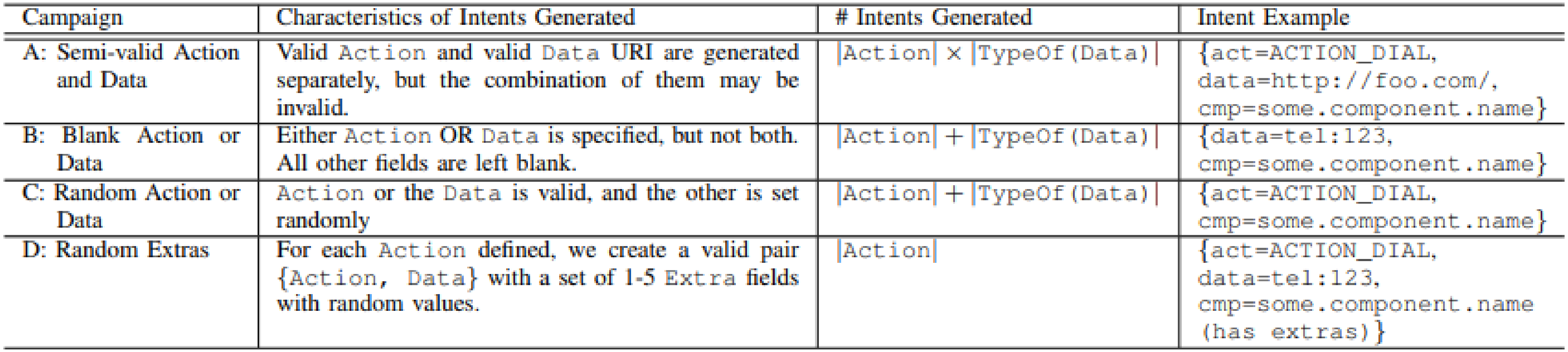
(3) When the AW device receives the message, the wearable app forwards the input (the target component and FIC) to the Fuzzer library to initiate the intent injection.

(4) The fuzzer library, which is part of the QGJ wear application, triggers the fuzzing on the chosen target app component.

Fuzz Intent Campaign

- Strategy
  - Fuzz generated, the number of intent generated, a sample
  - 100 actions and 12 types of data

TABLE I
FUZZ INTENT CAMPAIGNS



Campaign	Characteristics of Intents Generated	# Intents Generated	Intent Example
A: Semi-valid Action and Data	Valid `Action` and valid `Data` URI are generated separately, but the combination of them may be invalid.	|`Action`| × |`TypeOf(Data)`|	{`act=ACTION_DIAL`, `data=http://foo.com/`, `cmp=some.component.name`}
B: Blank Action or Data	Either `Action` OR `Data` is specified, but not both. All other fields are left blank.	|`Action`| + |`TypeOf(Data)`|	{`data=tel:123`, `cmp=some.component.name`}
C: Random Action or Data	`Action` or the `Data` is valid, and the other is set randomly	|`Action`| + |`TypeOf(Data)`|	{`act=ACTION_DIAL`, `cmp=some.component.name`}
D: Random Extras	For each `Action` defined, we create a valid pair {`Action, Data`} with a set of 1-5 `Extra` fields with random values.	|`Action`|	{`act=ACTION_DIAL`, `data=tel:123`, `cmp=some.component.name (has extras)`}

Target Apps and Error Manifestations

- Target apps:
  - Health/Fitness and non-Helath/Fitness
  - Builtin or Third-party apps

- Error manifestations
  - System reboot
  - Crash
  - Hang or unresponsive
  - No effect

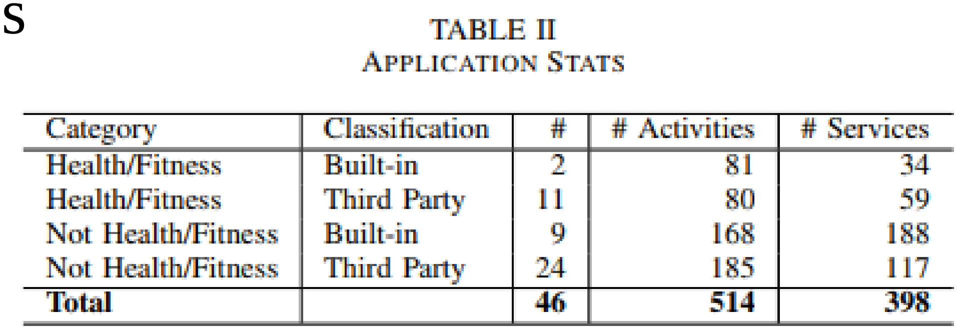


TABLE II
APPLICATION STATS

Category	Classification	#	# Activities	# Services
Health/Fitness	Built-in	2	81	34
Health/Fitness	Third Party	11	80	59
Not Health/Fitness	Built-in	9	168	188
Not Health/Fitness	Third Party	24	185	117
Total		**46**	**514**	**398**

Experiment Setup

- Phone: LG Nexus Android 5.0
- Communication: Bluetooth
- Wearable: Moto 360 Android wear 2.0
- Things to note
  - the relatively high frequency of Services compared to Activities: most of each application’s workload is done by Services

Experiment Mechanism

- Experiment Mechanism
  - Choose a particular wearable application using QGJ UI, from the mobile phone, and begin the experiments.
  - The fuzzer starts injecting malformed intents according to the particular FIC.
  - All 4 campaigns are executed one after another.
  - Collected all of the log files (over 2GB) from the wearable using logcat, through the adb interface.
  - Analyzed the logs to gather information, and for each component classified the behavior of the application according to the expected scenarios.
  - For any failure or error encountered, we manually analyzed further to find their possible root cause.

Experiment Results

- Distribution of Exception Types
  - After SecurityException, the second largest share belongs to IllegalArgumentException. This type of exception is raised because of the mismatch on the data contained in an injected intent and what is expected by the component

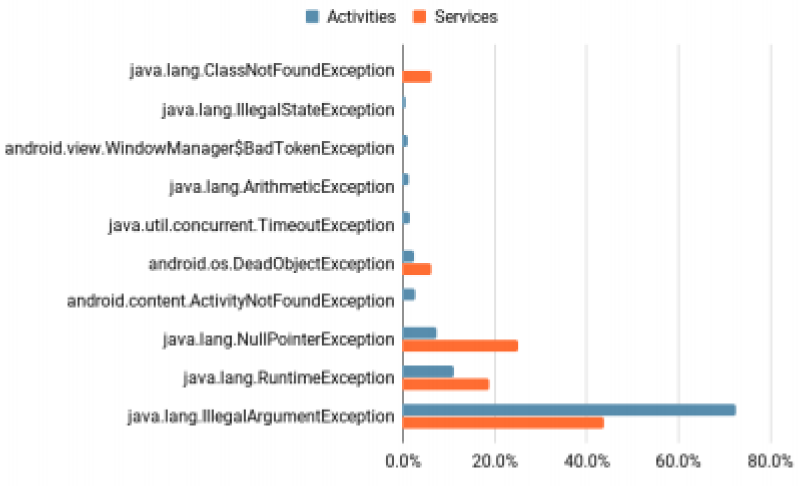
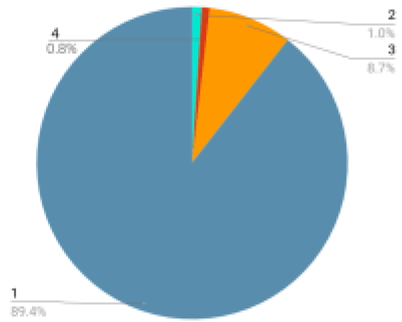


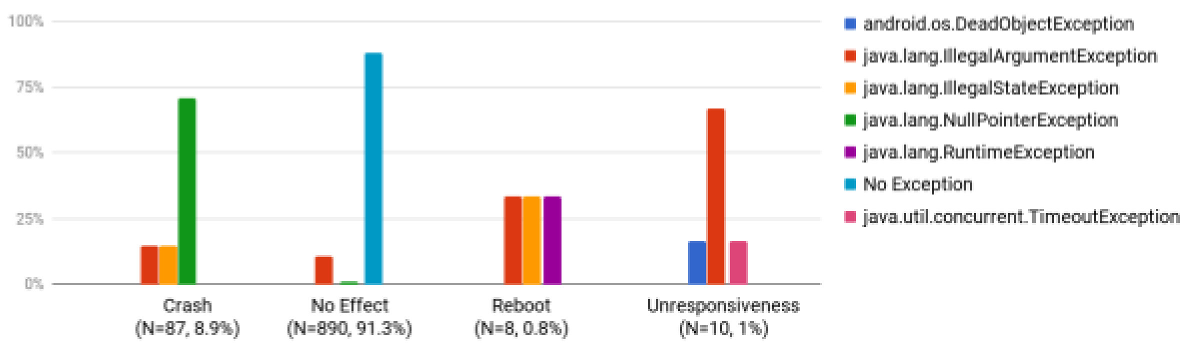
Fig. 2. Distribution of type for uncaught exceptions (without considering Security Exception) grouped by component type.

Experiment Results

- Distribution of Error Manifestation

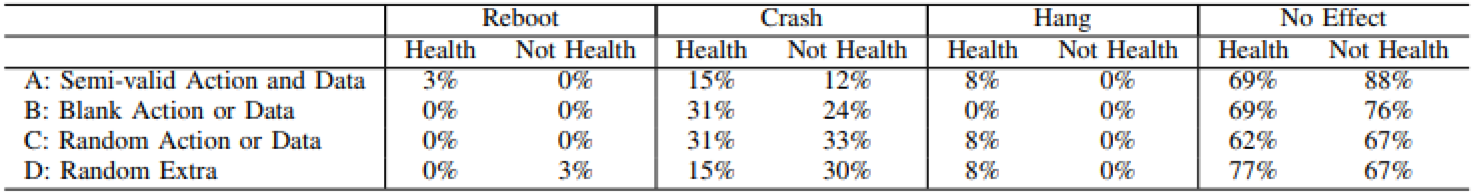


(a) Distribution of error manifestation among the application components



(b) Distribution of exceptions by manifestation

TABLE III
DISTRIBUTION OF BEHAVIORS AMONG FUZZ INTENT CAMPAIGNS



	Reboot		Crash		Hang		No Effect	
	Health	Not Health	Health	Not Health	Health	Not Health	Health	Not Health
A: Semi-valid Action and Data	3%	0%	15%	12%	8%	0%	69%	88%
B: Blank Action or Data	0%	0%	31%	24%	0%	0%	69%	76%
C: Random Action or Data	0%	0%	31%	33%	8%	0%	62%	67%
D: Random Extra	0%	3%	15%	30%	8%	0%	77%	67%

Software Engineering Techniques for improving robustness

- Techniques
  - Better tool support: In the software engineering community, there has been significant amount of work to analyze exception handing in Java applications: perform static analysis of the application codes to check how exception handling codes are linked together.
  - Consistent input validation: found that various Android components handle invalid arguments inconsistently: recommend that intent fields that can only have a limited number of valid values be more rigorously validated by the Android system.
  - Research on software aging: During our experiments, we found the Adroid Watch rebooted twice. These reboots were not due to a single malformed intent, but rather manifested at certain stages of the experiments: research on software aging and rejuvenation can help detect and potentially recover from such accumulated errors

Conclusion

- Things found
  - NullPointerException handling has improved relative to Android, there is still a disturbingly high incidence of other exceptions, such as IllegalArgumentException.
  - Built-in apps crash at a higher rate than popular third-party apps
  - A confluence of factors—software aging and cascading failures— can cause the entire device to reboot even through mutating unprivileged intents.
  - Three approaches that can improve the resilience of AW apps—better IDE support, consistent input validation, and guarding against software aging.
- Limitations
  - Used a single wearable device and thus is blind to vendor-specific customizations.
  - While most AW apps are two-part, we have ignored the inter-device interactions and focused only on the wearable components.
  - Our comparison with Android error manifestations is not fully accurate since the earlier studies were done on a different version of Android.