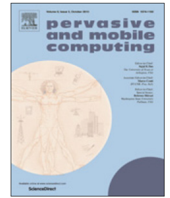
Pervasive and Mobile Computing 56 (2019) 106–131

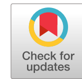
Contents lists available at ScienceDirect

Pervasive and Mobile Computing

journal homepage: www.elsevier.com/locate/pmc



CamTest: A laboratory testbed for camera-based mobile sensing applications



Chu Luo [a,*], Zewen Xu [b], Ruining Dong [a], Jorge Goncalves [a], Eduardo Velloso [a], Vassilis Kostakos [a]

[a] *School of Computing and Information Systems, The University of Melbourne, Australia*
[b] *SavvyPlus Consulting, Australia*

A R T I C L E   I N F O

Article history:
Received 5 July 2018
Received in revised form 22 February 2019
Accepted 1 April 2019
Available online 4 April 2019

MSC:
00-01
99-00

Keywords:
Mobile devices
Smartphones
Software testing
Multimedia

A B S T R A C T

Testing camera-based sensing applications on mobile devices is challenging and time-consuming. In this paper, we present CamTest, a novel laboratory testbed for camera-based mobile sensing applications. It enables researchers to conduct systematic, affordable and efficient tests on their applications in laboratory conditions. Instead of real-world tests that directly use the camera, CamTest delivers the frames of a video clip to the image processing module of the targeted application. Meanwhile, CamTest tracks the image processing output, errors and image processing speed of the targeted application at runtime. After testing, CamTest can visualise the recorded information simultaneously with video playback so that testers can understand how the sensing context affects the behaviours of the application. Through computational efficiency measurements, and a user study with 14 developers, we show that CamTest can effectively facilitate and simplify the testing of camera-based mobile sensing applications.

© 2019 Elsevier B.V. All rights reserved.

1. Introduction

Manufacturers of mobile devices now embed cameras into modern smartphones and tablets. Beyond taking digital photos and recording video, an increasing number of mobile applications use computer vision and machine learning techniques to provide cognitive services assisting people in daily life. Typical scenarios of camera-based mobile sensing applications include car driving monitoring (e.g., CarSafe [1]), object recognition (e.g., FoodCam [2]) and biometric authentication (e.g., face recognition [3]).

Although researchers and developers have proposed a large number of techniques to develop camera-based mobile sensing applications, it is challenging to systematically test these applications. Unlike UI (user interface) testing relying on human input, testing camera-based mobile sensing applications requires relevant video content. Because these applications dynamically adapt to the images captured by the camera, testers, without suitable testing tools, have to conduct tests in intended scenarios. For example, to test a car driving monitoring application in the real world, testers have to drive a car with the device on the actual road. Through the development process of an application, testers may need to repeat such tests for multiple iterations (e.g., in regression testing). Moreover, testers have to perform tests on a variety of different mobile devices for testing compatibility and performance. Hence, it is time-consuming, laborious and expensive to conduct in-the-wild tests. More crucially, it is hard to reliably replicate the testing conditions.

∗ Corresponding author.
E-mail addresses: chul3@student.unimelb.edu.au (C. Luo), vassilis.kostakos@unimelb.edu.au (V. Kostakos).

https://doi.org/10.1016/j.pmcj.2019.04.001
1574-1192/© 2019 Elsevier B.V. All rights reserved.

To avoid such painful in-the-wild tests, we propose CamTest, a novel laboratory testbed for camera-based mobile sensing applications. It enables systematic, affordable and replicable tests on these applications in laboratory settings. Instead of in-the-wild tests directly using camera, CamTest emulates the camera recording mechanism by delivering the frames of a video clip to the image processing module of the targeted application. Thus, the replayed frames of the video drive the targeted application to run as per usual. Meanwhile, CamTest tracks the image processing output, errors and image processing speed of the targeted application at runtime. In addition, CamTest provides mechanisms that allow testers to annotate the video file and analyse testing results during the video playback, so that testers can understand how the sensor context affects behaviours of the application.

Overall, our work makes the following contributions:

1. We formalise the goals for testing camera-based mobile sensing applications. We provide insights for developers and researchers who conduct tests or design similar testing tools for these applications.
2. We present CamTest as a testing infrastructure to support systematic, affordable and replicable testing of camera-based mobile sensing applications in laboratory settings.
3. We quantify the computational efficiency of CamTest regarding CPU, memory and battery usage. Experimental results show that CamTest has significantly lower overhead than other approaches.
4. We evaluate CamTest in a case study and user study with 14 developers. We show that CamTest can effectively facilitate and simplify the testing process.

2. Related work

With advanced machine learning techniques in image processing, researchers and software practitioners have designed a significant number of camera-based mobile sensing applications. However, testing these applications is a challenging task. Testing methods and tools for context-aware systems can hardly keep pace with the development of context-aware computing technology [4]. Despite a plethora of data collection middlewares and context management frameworks, it remains challenging to conduct low-cost and efficient testing in laboratory settings for mobile context-aware applications [5,6]. Especially, for testing camera-based mobile sensing applications in laboratory settings, there exists no approach in the state of the art.

Hence, in this section, we first provide an overview of camera-based mobile sensing applications. Then, to understand the need for testing camera-based mobile sensing applications, we summarise relevant work aiming to test other mobile context-aware applications.

2.1. Camera-based mobile sensing applications

Among the wide variety of camera-based mobile sensing applications, there are three typical categories: car driving monitoring, object recognition and biometric authentication.

2.1.1. Car driving monitoring

Due to dangerous road conditions and driving behaviours, the safety of car driving is a paramount issue in modern society. Many camera-based mobile sensing applications aim to alleviate this problem.

CarSafe [1] is an Android-based smartphone application which can detect dangerous traffic conditions and driving styles. With algorithms processing real-time camera video stream, it monitors the driver using the front camera, and simultaneously tracks road situations using the rear camera.

Besides camera, Bergasa et al. [7] integrate microphone, GPS, accelerometer and gyroscope into an iOS smartphone application to monitor the car driving conditions. The rear camera tracks lane marking using Kalman filtering. To detect whether a lane change is on purpose, the microphone captures the supposed clicking sound from the car indicator. GPS, accelerometer and gyroscope provide inference reflecting the driver's behaviours.

2.1.2. Object recognition

Object recognition using camera is an important technique to make mobile devices more useful in everyday life.

Kawano and Yanai [2] present a food recognition system called FoodCam using real-time computer vision on a smartphone camera. The recognition relies on the phone processor only, without sending images to a remote server. When recognising food, the phone user draws a box to locate the relevant image part on the screen. FoodCam segments the region and extracts image features for a linear SVM classifier to process. Experimental results show that FoodCam can generate top-5 candidates with 79.2% accuracy among a food dataset of 100 categories.

To help tourists find scenic routes, in this case the best route to see blossoming cherries, Morishita et al. [8] designed a camera-based sensing system called SakuraSensor. Serving as a participatory sensing system, SakuraSensor allows car drivers to voluntarily share information of adjacent blossoming cherries using the cameras of car-mounted smartphones. Based on an image processing algorithm tracking image pixels from blossoming cherries, SakuraSensor can correctly detect blossoming cherries with 74% precision and 84% recall.

More importantly, camera-based object recognition can significantly support visually impaired people with a smartphone. For instance, Grijalva et al. [9] propose a denomination recognition system that helps visually impaired people

identify United States currency of notes using smartphone camera. Relying on Principal Component Analysis (PCA) and an image recognition method called Eigenfaces, this system can classify notes with an accuracy of 99.158% in indoor environment, at a processing rate over 7 frames per second.

2.1.3. Biometric authentication

Biometric authentication is increasingly popular on regular mobile devices. Many techniques in this field become workable options to unlock devices.

Face recognition is a widely used technique on mobile devices, also becoming a standard way of biometric authentication. For example, Shen et al. [3] propose a smartphone-based face recognition method. This method aims to achieve high recognition accuracy by optimising the projection matrix of sparse representation classification, which is a classic algorithm for face recognition. On a set of different face expressions (e.g., neutral, happy, sad and so on), this method can achieve 90% overall accuracy.

On a smaller scale, Raja et al. [10] design a camera-based iris recognition application on smartphones. They develop a new segmentation scheme to extract iris images from backgrounds. With a new feature extraction technique using deep sparse filtering, the application verifies iris image using sparse representation classification. In a study on an available public database, experimental results show that this application can identify iris with 98% mean accuracy.

Raghavendra et al. [11] present a scaling-robust fingerprint recognition system using smartphone camera, unlike expensive phone models with specialised fingerprint sensors. Given a fingerprint image, this system first segments finger projection using Mean Shift Segmentation (MSS) algorithm. Then it performs the scaling to capture the fingerprint region using a novel extraction algorithm. Lastly, it compares the captured image and intended fingerprint using minutiae features. In complex backgrounds, experimental results show that this system can achieve 96% accuracy.

2.2. Testing mobile context-aware applications

Testing smartphone applications in the real world is time-consuming and expensive [5]. To enable affordable and efficient testing of mobile context-aware applications, researchers and software practitioners have proposed several approaches for laboratory settings, mainly in three categories: context generation and simulation; context record and replay; non-functional testing.

2.2.1. Context generation and simulation

Besides real-world tests, the generation and simulation of context in a laboratory can also drive mobile context-aware applications to run.

Amalfitano et al. [12] highlight that the basic events driving mobile context-aware applications are contextual events and UI events. They proposed several techniques to generate events that exercise applications:

- **manual technique**: Testers can manipulate events as test cases that applications take as input;
- **mutation-based technique**: Testers can use some fundamental event-patterns to generate more events. This can increase the coverage of testing;
- **exploration-based technique**: Testers can write a script to dynamically trigger events at runtime. They can conduct either a systematic (e.g., brute force) or random exploration in all the states of the application. Regarding simulation, they chose Android as the experimental platform. They overwrote various functions in the kernel of Android to release generated events.

To improve the efficiency of testing, Tonjes et al. [13] proposed a semi-automated approach for context generation and simulation. They designed a model to describe mobile context-aware applications using 4 elements: **states**, **events**, **actions** and **transitions**. Based on such a model, they presented an automation technique to generate and execute test cases (i.e., specific values of contextual data) to exercise the application model without human efforts. To avoid long time execution on an extremely large number of test cases, they used an optimisation algorithm to minimise test cases that lead to similar application actions.

Similarly, based on the popular UML (Unified Modelling Language) diagrams, Griebe and Gruhn [14] present another method of model-based testing for mobile context-aware applications. Their method enhances UML Activity Diagrams with contextual parameters and test data. Then, their method transforms these diagrams into Petri Net representation which generates relevant test cases by calculating the reachability tree. Finally, the test cases are converted into the platform-specific code to run on the actual application.

2.2.2. Context replay and record-and-replay

Context replay tools can read data from a data source and deliver the data to mobile context-aware applications. For instance, KnowMe [15] is a Java-based tool that replays contextual data collected by the mobile sensing middleware AWARE [16]. Before a test, testers should connect the tool to an online databases of AWARE [16] to obtain test data. Thus, it supports more than 20 types of contextual data, including hardware sensors (e.g., GPS and accelerometer) and software/OS information (e.g., application launch events). To quickly examine the functions of a mobile application, it

also allows testers to increase the replay speed. However, it is only available on PCs, meaning that testers can only test a PC-runnable code from the mobile application.

To enable context replay on mobile device for directly testing mobile context-aware application, Luo et al. [5] designed an Android-based context replay tool called TestAWARE. It has two components: a mobile client and code library. The mobile client can conduct context replay by manual controls from human. It supports all the data types provided by AWARE [16]. It also supports raw audio data for the testing of microphone-based mobile applications. The code library provides white-box testing functions which can be imported by mobile applications to detect bugs and evaluate performance.

Record-and-replay tools, beyond context replay, have the record feature to collect contextual data for replay. This bring more convenience for the preparation of testing.

For example, RERAN [17] is a record-and-replay tool aiming at Android smartphone applications. It can capture system-level data streams including GUI events (e.g., zoom-in control of a picture reader) and sensor data (e.g., accelerometer, proximity, light sensor and magnetometer). It achieves microsecond accuracy in timestamps of a data replay. However, it cannot collect GPS events because Android encapsulates GPS as a stand-alone service. RERAN requires privileged control (i.e., root access) to replay data on the device. Although the configuration of rooting is highly complex, the advantage is that testers can run replay without changing application source code.

Another example is MobiPlay [18] which is a remote platform. Developers can use MobiPlay to test Android applications by replaying datasets, including GUI events and sensor data. It has two components: a server and client. During testing, MobiPlay launches the targeted application and replays data on its remote server. The MobiPlay client runs on mobile devices, serving as the interface to accept controls by human. Despite the convenience of black-box tests on the server, installing specialised servers is a laborious and expensive task.

2.2.3. Non-functional testing

Since regular mobile devices have limited battery capacity and computing resources, non-functional properties play a critical role in mobile sensing applications. Hence, researchers and software practitioners have proposed several tools to test non-functional properties of mobile applications.

For instance, Chu et al. [19] present a non-functional testing tool called Kobe. It addresses latency and energy concerns of mobile context-aware applications with minimal loss of recognition accuracy. To balance the trade-offs, Kobe can change the sampling rate of sensors or offload heavy computation to a cloud server. It can also adapt to the dynamic environments of mobile devices, such as connectivity changes, different user habits/preferences, and computing load of devices. To reduce the unnecessarily repetitious sensor usage from different applications, it provides a SQL-like interface to synchronise sensor values across applications.

Moreover, FOREPOST [20] aims to automatically detect performance bottlenecks in black-box testing. It can learn and execute rules that contain classes of input data causing intensive computations. By identifying traces having longer execution time, it gives testers advice to find related function calls and optimise them.

Although these specialised tools can examine non-functional properties, testers can only use them after the targeted application is completely built. To save time in development, researchers and software practitioners have integrated several tools into IDE (integrated development environment) to conduct non-functional testing at the design or implementation stage.

For example, PADA [21] is an IDE-based power estimation tool to assess the power consumption of mobile sensing applications by analysing the source code during implementation. Based on the techniques of sensor-related API analysis, it allows developers to measure power use in different environments, rather than running applications on actual devices in the real world.

3. Design goals of testing camera-based mobile sensing applications

Prior to describing our testbed, in this section we summarise the requirements, necessity, design goals and deliberations of testing camera-based mobile sensing application.

3.1. Motivation: Testing an example camera-based smartphone application

Suppose we are developing a camera-based application for smartphones. Fig. 1 shows the mechanism of general camera-based sensing applications, where:

1. The camera continuously provides image data streams in real-time.
2. The application processes image data streams using computer vision techniques.
3. The application generates output after processing video frames.

During testing, we can easily examine whether the camera can correctly collect video frames for our application. Generally, camera-based mobile sensing applications have common requirements for testing. According to prior research in testing mobile context-aware applications, the testing of the video processing components providing context-awareness involves several necessary tasks:

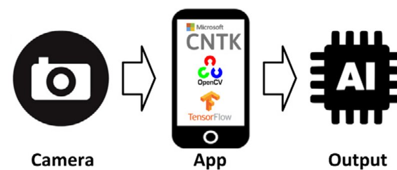


Fig. 1. Mechanism of general camera-based mobile sensing applications.

1. Functional testing: validating the correctness of video processing modules. Correctness is a critical requirement for mobile applications [22]. For mobile sensing applications based on other hardware sensors (e.g., accelerometer), there exists a large number of approaches and tools for developers to validate the functional correctness, such as [5,17,18].
2. Non-functional testing: measuring the non-functional properties. Previous research [5,19,21,23] has identified several key non-functional properties for mobile context-aware applications: recognition accuracy, latency, computational cost and energy-efficiency.
3. Regression testing: confirming that the previous properties of software still remain after a modification to the software. Several studies [22,24] have pointed out that efficient regression tests play a critical role in software quality, because many developers adopt incremental development methods such as delivering nightly and weekly builds. To test these nightly or weekly builds, using automated testing techniques to accelerate testing processes is important [22].
4. Compatibility testing: ensuring that the software is able to operate on the actual devices. Since mobile devices run various versions of operating systems with different configurations, it is necessary to check whether the software is compatible to different operating environments [25] . Developers have to run tests using actual devices, because PC-based tools and emulators cannot yet provide realistic environments to conduct such tests [25].

3.2. Shortcomings of existing approaches and tools

The development tools for mobile applications provide basic features for testing. For example, Android Studio and Xcode both allow developers to conduct some functional and non-functional tests, including executing test cases specified by scripts (e.g., using the tool Monkey [26] on Android Studio), triggering runtime assertions, and measuring computational cost. However, these development tools cannot generate and replay contextual data to test context-aware applications [18]. Previous studies [5,18] highlighted the importance of replaying contextual data in reproducing context-related bugs. Hence, researchers proposed several context-replay tools for testing mobile context-aware applications, as summarised in [5] . Although these tools can replay system events (e.g., screen on/off), regular sensor data (e.g., accelerometer and GPS) and audio samples recorded by microphone, mobile developers lack a testing tool or technique which can replay video data on mobile devices to test camera-based mobile sensing applications. Developing such a video-replay testing tool was also stated as future work in a recent article [5].

Computer vision researchers have proposed several tools and techniques to support the testing of their applications, including large-scale databases of video content [27] and low-level APIs of objective quality metrics (e.g., Peak Signal-to-Noise Ratio) [28,29]. However, none of these approaches focus on a complete video-replay testing tool for testing functional and non-functional properties of camera-based mobile sensing applications.

Technically, using a combination of different existing tools and techniques on both desktops and mobile devices is a possibly workable way to complete the testing tasks. On a desktop, developers can customise a video player that sends video frames into the desktop version, if available, of video processing modules to examine for correctness. Then on mobile devices, developers can run the mobile version of their applications to test non-functional properties. Although this kind of testing may be workable, managing both desktop and mobile version of application is laborious and time-consuming. Unnecessary repeats of tests on multiple tools for testing different properties are also inefficient. We also note that mobile developers may frequently conduct regression testing on their nightly or weekly builds in iterative and incremental development [22].

Consequently, to overcome these shortcomings, a holistic testing tool that supports efficient testing of functional and non-functional properties of camera-based mobile sensing applications is needed.

3.3. Design goals and deliberations of the intended testbed

Driven by the testing tasks and identified shortcomings of prior work, the design goals of the intended testing tool can be specified. For each testing task, Table 1 summarises how our design goals attempt to overcome the shortcomings of previous work.

Replay video. First, since prior work cannot replay video as test cases on mobile devices, our primary design goal is to replay video frame by frame to exercise the targeted application, as shown in Fig. 2. By replaying a video we automate

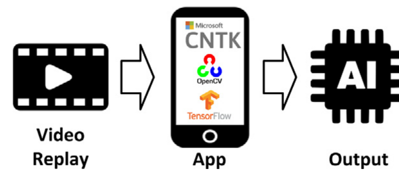


Fig. 2. The testing process with video replay for camera-based sensing applications.

Table 1
Summary of the design goals corresponding to the shortcomings of prior work for each testing task.

Task	Shortcomings of prior work	Design goals
Functional testing	Cannot replay video on mobile devices	Replay video as test cases
Functional testing	Cannot track video-related bugs	Log runtime bugs
Non-functional testing	Cannot track actual performance	Measure performance on devices
Regression testing	Time-consuming in using multiple tools	Combine all features on a testbed
Compatibility testing	PC-based tools cannot verify compatibility	Test applications on devices
Miscellaneous	Time-consuming in analysing results	Combine analyses on a testbed

the execution of test cases, and also facilitate the reproduction of bugs. In addition, testers can use standard video files (e.g., downloaded from an online library) to simplify the test case acquisition.

Mobile application developers, especially testers, may not have sufficient knowledge of video decoding to correctly customise video replay on mobile devices using low-level APIs. Hence, we decided to encapsulate the video replay feature inside an importable library. To initialise a video replay, the function has to receive several necessary parameters as input, such as the video file name and path. It is common to expect the parameters in human-written code when testers conduct tests. However, if testers want to change video files, they have to rewrite the parameters and recompile the whole application. Thus, to minimise the effort during repetitious tests, we decide to develop a mobile application with a GUI for editing the settings of tests at run-time. In this way, testers can quickly conduct different kinds of tests (e.g., using different video files) with a few clicks on the mobile device, rather than modifying and rebuilding the tested application.

Log runtime bugs. The second goal is to record runtime output from the targeted application, including bug reports and assertions at runtime. If an application enters an erroneous state or crashes, testers need to check the bug report afterwards. For camera-based mobile sensing applications, errors can be caused by video processing modules, as well as other modules (e.g., storage on mobile devices). Moreover, to check low-level details, testers often add assertions to compare the actual result with the expected result at runtime. Due to the opaqueness of video processing, it is necessary for our testing tool to label whether bugs or assertions are caused by video processing modules. This can simplify the process of bug fixes in source code. Thus, we decided to provide a code library that can be imported by the targeted application to record output and assertions at runtime.

Measure performance on devices. The third goal is to track the performance of applications on the actual mobile device at runtime. Due to the hardware differences between regular PCs and mobile devices, testers have to measure the performance on actual mobile devices. For camera-based mobile sensing applications, relevant performance metrics are recognition accuracy (measured by classification or regression results), and recognition latency (measured by processed frames per second).

Combine all features on a testbed. The fourth design goal is to combine all features on a testbed. Testers have to conduct multiple testing tasks for different properties. If testers rely on a number of different tools to conduct different tests, it is time-consuming to repeat tests (e.g., during regression testing). Note that the modern software industry largely adopts iterative and incremental development which requires developers and testers to deliver nightly or weekly builds. Hence, we decided to provide all the features for functional and non-functional testing on a single testbed.

Test applications on devices. The fifth goal is to test applications on actual mobile devices. PC-based testing tools or emulators cannot ensure compatibility of all packages used in mobile applications. For real device testing, we decide to provide the testing features of our testbed as mobile-based libraries and a mobile client which runs on mobile devices.

Combine all analyses on a testbed. The sixth design goal is to conflate analyses of testing results on our testbed. By avoiding analyses using multiple testing tools or interfaces, an integrated analyser can accelerate the process for testers to conduct different tests. Considering that most mobile devices have small screen sizes, we decide to provide the analysis feature on desktops, as a part of our testbed. For the measurement of visual recognition, we also provide the video annotation feature on desktop to help testers provide ground-truth labels.

Platform selection for the testbed. To implement the testbed, a design deliberation is whether a feature module should run on mobile devices or regular PCs. As a result, for each concern of testing, we provide a comparison of capability between mobile devices and regular PCs in Table 2:

1. **Annotating video content:** Before a test, testers have to provide ground-truth labels for the video content. For example, if the targeted application performs object recognition, testers should specify what objects appear in the

Table 2
Comparison of capabilities between mobile devices vs. regular PCs in testing camera-based mobile sensing applications.

	Mobile device	PC
Annotating video content	Inconvenient	Convenient
Testing functional properties	Achievable	Possibly achievable
Testing non-functional properties	Achievable	Not achievable
Analysing results	Inconvenient	Convenient
Compatibility of prebuilt libraries	Mobile prebuilt libraries	Cross-platform libraries

video, with correct timestamps. Unlike mobile devices with only several keys and small touch-screens, PCs have keyboards and mice as input methods for labelling and video controls. Thus, using PCs to annotate video content is more convenient.

2. **Testing functional properties:** As camera-based mobile sensing applications are designed for mobile devices, testers can directly install their applications on actual devices to conduct functional testing. In contrast, although PC-based tools or emulators may also be able to run these applications, they have several limitations in functional testing. First, PC-based tools or emulators cannot examine the correctness of some functions, such as vibration effects and functions based on cellular/Bluetooth/USB/headphone connection. Second, PC-based tools or emulators cannot take multi-touch as input, which may be related to the controls of functions. Third, PC-based tools or emulators cannot simulate the state change of SD cards. If the SD card is used by the targeted application, removal of the SD card during usage may trigger some actions of the application on a real device. Besides, there can be other issues due to the limited capability of PC-based tools or emulators, such as compatibility of pre-built libraries (will be detailed later in this section).
3. **Testing non-functional properties:** Due to the hardware difference between PCs and mobile devices, accurate testing of most non-functional properties can only be achieved on actual mobile devices, such as processing speed and computational cost. Computing resources of regular PCs are generally more powerful than those of mobile devices. For example, PC CPUs are significantly faster than mobile CPUs [30]. Moreover, most PC CPUs are based on the x86/x86-64 architecture (used by Intel and AMD). Most mobile CPUs are based on the ARM architecture (used by Apple, Samsung, etc.). Prior work pointed out that CPUs of the two architectures have unique advantages due to their different features [31].
4. **Analysing results:** After a test, testers have to conduct post-task analysis to understand the results and find problems. Similar to annotating video content, displaying or rewinding the captured results with the large screens, keyboards and mice on PCs is more convenient compared to mobile devices. Also, since the source code of mobile applications is written on PC-based tools, testers can quickly locate a piece of code corresponding to the results analysed on PCs.
5. **Compatibility of prebuilt libraries:** Many developers develop their applications with third-party prebuilt libraries. These prebuilt libraries are architecture-specific (e.g., for x86 or ARM) [32]. Thus, a mobile application with mobile (ARM-oriented) prebuilt libraries cannot run on regular PCs with the x86 architecture. To make this application run on a PC, developers must rebuild it with x86-oriented prebuilt libraries (i.e., the prebuilt libraries are cross-platform). However, not every prebuilt library is cross-platform for both mobile devices (ARM) and PCs (x86). Software library owners may only distribute the prebuilt libraries without publishing the source code (e.g., for commercial or scientific confidentiality). For example, the Tensorflow mobile demo [33] provides the prebuilt libraries of object recognition for ARM only. Without its x86-oriented version, developers cannot build an application that can run on PCs.

Based on our comparison results, it is unwise to try to implement all features of the testing tool entirely on either mobile devices or PCs. Thus, we decided to implement the features for testing functional and non-functional properties of applications on mobile devices, the features for annotating video and result analysis on PCs.

With the above design goals, our testbed aims to support researchers to conduct a complete testing process on their camera-based mobile sensing applications, from test data preparation to the final analysis. Note that the process is entirely in laboratory settings, to avoid the high cost and low reproducibility of real-world tests. Driven by these goals, the components of our testbed are detailed in the next section.

4. CamTest

Fig. 3 shows the architecture of CamTest. It consists of 4 components: mobile client, replay service, video annotator and analyser. The mobile client and replay service operate on mobile devices where the targeted application is installed. They conduct video replaying and log the output of the targeted application at runtime. We implemented the mobile client and replay service using Android. Since major mobile platforms, such as Android (from version 4.1, using MediaCodec API [34]), iOS (from version 6.0, using VideoToolbox framework [35]), and Windows Phone (using Windows.Media in Universal Windows Platform [36]), all support hardware-accelerated video play, this design of inner-application video replay is suitable for most of these platforms. The video annotator and analyser operate on a PC where testers input labels into video files and analyse test results.

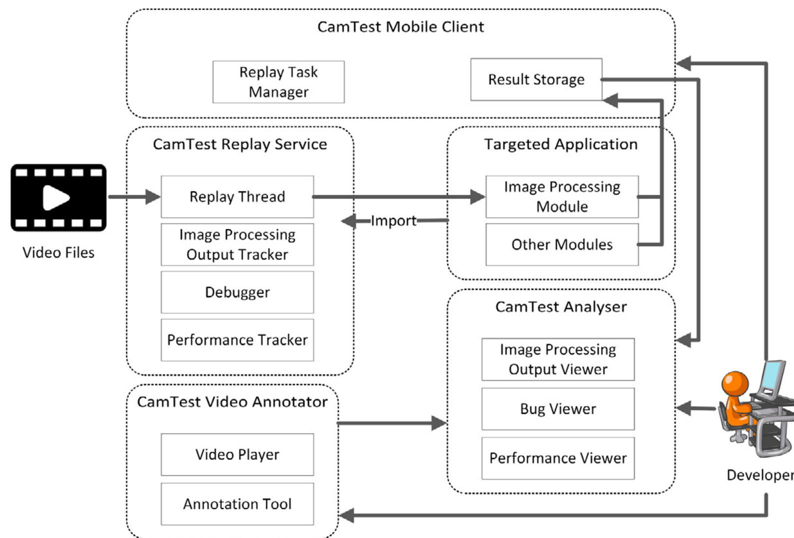


Fig. 3. Architecture of CamTest. Targeted applications, video files and developers interact with components of CamTest.

Table 3
Mapping from design challenges to corresponding solutions.

Design goal	Challenge	Solution
Replaying video	Inter-app replay may not work	Offer replay in an importable library
Replaying video	Computational cost may be high	Use hardware acceleration (GPU)
Replaying video	Deliver frames for apps to receive	Use the container of camera output
Replaying video	Difficult to change replay settings	Manage them in the mobile client
Logging app output Tracking app performance	Difficult to save runtime output	Add a shared storage in mobile client
Analysing test results	Difficult to analyse results on mobile client storage	Add the feature of exporting results from mobile client to PC analyser

4.1. Design challenges and solutions

Before discussing the technical details of CamTest, we first describe the design challenges and corresponding solutions to achieve the identified design goals. In Table 3 we present, for a certain design goal, how each design challenge is mapped to our solutions in CamTest.

Inter-app replay or intra-app replay. To replay video, a design challenge is to determine where to implement it: inter-application (i.e., an external application replays video in its process and delivers the frames to the targeted application) vs. intra-application (i.e., within the process of the targeted application, a thread replays video and sends the frames to the video processing module being tested). On major mobile platforms, the support for inter-application communications is limited for video images. iOS 7.0+ even has a very strict sandbox policy that does not allow any inter-process communication. On Android, the targeted app has to be granted permissions to receive inter-application communications, which may affect the test results of permission-related functions. Considering the availability and permission-related issues of inter-application communications, we adopted the intra-application approach.

Computational cost of replaying video. Another design challenge is the computational efficiency of replaying video. Replaying video is a computationally intensive task, especially on mobile devices. On Android and other major platforms, there exist two approaches for multimedia-based applications: software and hardware acceleration. Software-based approaches rely on CPU, such as the audio/video codec project on Android [37] using FFmpeg [38]. In contrast, solutions based on hardware acceleration perform graphics-related tasks inside the GPU. Android provides hardware acceleration support since version 3.0 [39]. Considering that the video replaying thread has to run together with the video processing module being tested, we decide to implement the video replaying feature using hardware acceleration to minimise the computational cost.

Delivery of video frames to apps. After the video replaying thread decodes image frames from a video file, a design challenge is to deliver these frames to the video processing module. If the video processing module cannot receive these frames in a suitable way, it may not work to drive responsive functions. In practice, camera-based mobile sensing applications receive image frames directly from the camera. Then, these applications may output some results together with the captured image frames on the screen. Thus, the testing tool has to identify and use the container that is intended

for the frames directly from camera, rather than the modified frames with visual effects generated by the targeted application. We will detail our solution later based on the options provided by Android (see [40]).

Change of replay settings. To support repeated testing with different video files, a design challenge is the change of replay settings. Since replaying video is enabled by an importable library, testers have to modify the parameters of the code to change the replay settings. Because of the change in source code, this requires the rebuild of the entire application. Considering that writing these parameters by hand is laborious and rebuilding the entire application is time-consuming, we decided to simplify the change of replay settings by implementing a GUI for configuring replays on the mobile client. Testers can change the replay settings without modifying the code and rebuilding the entire application being tested. Once a replay is about to start, the video replay thread reads the settings from the storage of mobile client.

Saving app output at runtime. To examine the app's properties, a design challenge is the collection of the runtime output from the application. When the targeted application generates output during testing, the mobile device has to record such output in device storage. Because the replay service is imported as a library of the targeted application, implementing the output storage in the library requires several critical permissions from the targeted application, such as access to the internal or external storage of mobile devices. Unfortunately, the addition of these permissions may affect the security/privacy level and the behaviour of the targeted application (e.g., an application may decide to launch some different actions when it has these permissions). Also, from the targeted application, it is difficult to retrieve the recorded results after tests, because the library cannot have an interface in the targeted application. Since the mobile client has to run together with the targeted application during testing, we decided to implement the output storage in the mobile client. The benefits are twofold. First, the targeted application only needs a non-general-purpose permission (i.e., effective for CamTest only) to access the storage of the mobile client, which does not affect its security/privacy level and behaviour. Second, we can implement an interface for testers to manage testing results from the mobile client.

Analysing test results from saved output. Analysing test results from captured output is also a design challenge. Since the output is saved by the mobile client on the mobile device, one design choice is to implement the analysis feature in the mobile client. Although some previous tools did so, this choice is not suitable for video-related analysis. Compared to PCs with monitors, keyboards and mice, the controls for video play are not convenient on mobile devices due to the screen size and input methods. Hence, we decide to implement the analysis feature on PC. However, we have to design a mechanism to export the results from the mobile client to the PC analyser. Since the supported storage is SQLite database on Android, our mechanism convert SQL entries into comma separated value (*.CSV) files for the PC analyser. Then, testers can analyse the output with the video playback on PC.

4.2. Mobile client

The mobile client manages the replay tasks. It also records the output and performance information from the targeted application at runtime. To perform detailed analysis, testers can export test results from the mobile client to a PC. Testers can change the settings of replay tasks using the GUI of our mobile client, without modifying code and rebuilding the entire application being tested.

```
1  <permission
2          android:name=``io.camtest.READ_DATA''
3          android:description=``@string/app_name''
4          android:label=``Read_Replay_Settings''
5          android:protectionLevel=``normal''>
6  </permission>
```

Listing 1: Permission for reading replay settings

```
7  <provider
8          android:name=``.providers.Replay_Provider''
9          android:authorities=``${applicationId}.provider.replay''
10         android:exported=``true''
11                 android:readPermission=``io.camtest.READ_DATA''  />
```

Listing 2: The storage provider mapped to the permission

Fig. 4 shows the interface of the mobile client. Before a test, testers should specify the information of the intended replay, including the replay name, targeted application, video file, orientation of the video, and the replay mode (Fig. 4a). Testers can choose any video file on the device storage to perform a replay (Fig. 4b). In the replay list (Fig. 4c), testers can schedule a replay task and export the test results of completed replay tasks.

To read the replay settings and send the output of the targeted application at runtime, the replay service must access the storage of the mobile client. Hence, the mobile client has two customised permissions (i.e., read and write) of storage access. For example, Listing 1 shows the definition of the permission to read the replay settings. Listing 2 shows the storage provider mapped to this permission.

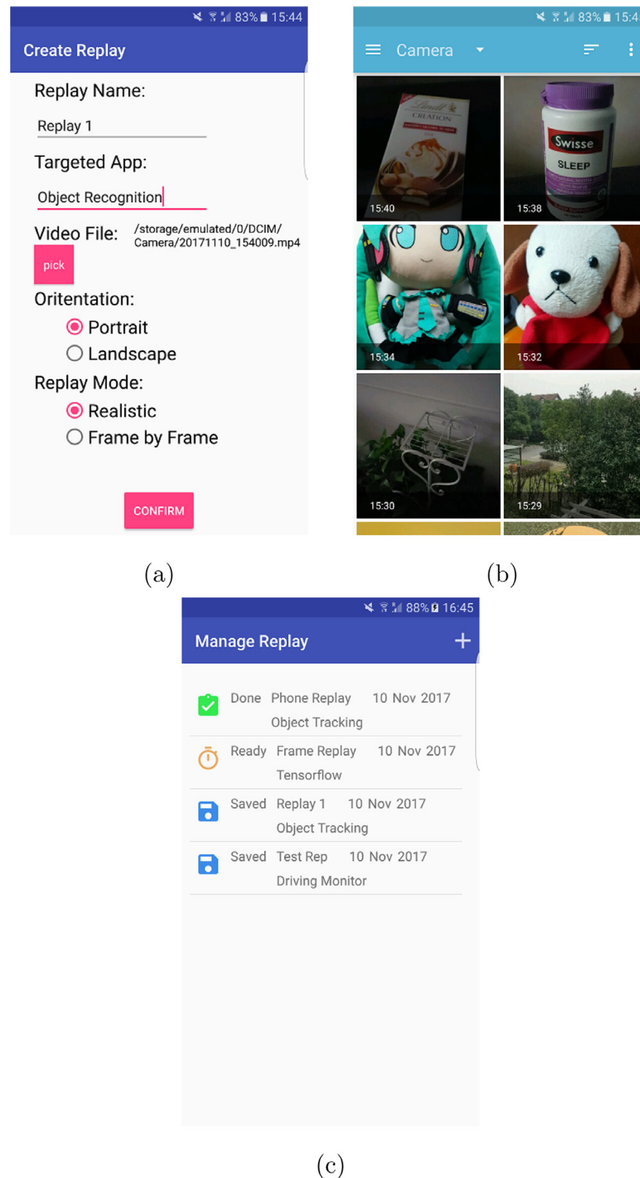


Fig. 4. Interface of the CamTest mobile client.

Developers of external applications, such as the targeted application, must add these permissions into the application source code (i.e., the AndroidManifest.xml file which defines the functionality and requirements of an application on Android) to use the data storage of the mobile client. For example, for reading replay settings, they must add $< uses - permission\ \ android : name = "io.camtest.READ_DATA" / >$. For recording output of the targeted application, the permission definition and usage are similar.

4.3. Replay service

The replay service provides four mechanisms: replaying video, capturing the output of the targeted application, recording bugs/assertions, and measuring the performance. The replay service encapsulates these features in an importable library which the targeted application should import as a component to use these mechanisms.

4.3.1. Replay thread

The replay thread can decode and deliver video frames to an image container. Running in a background thread, the replay thread operates concurrently with the video processing module and other modules of the targeted application in

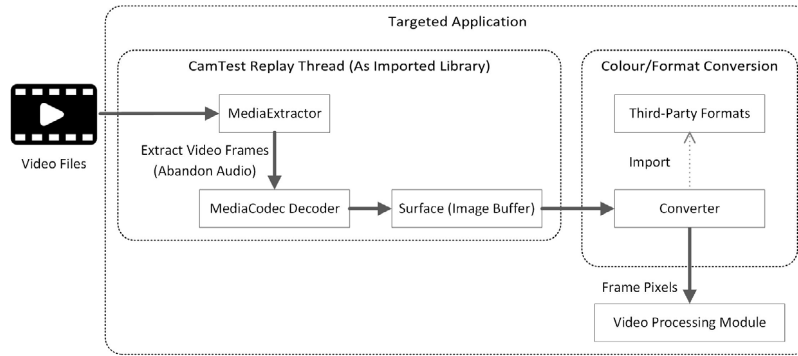


Fig. 5. The video replaying process of the replay service.

the main thread. Since most modern mobile devices have multiple cores, the replay thread does not affect the performance and functionality of the targeted application.

However, despite the multi-thread design, replaying video is a computationally intensive task for typical mobile devices today. Unlike previous work entirely relying on CPU (e.g., [37]), we implement the video replaying feature using hardware acceleration [39] to minimise the computational cost.

Among all the multimedia functions available on Android, only two functions *android.media.MediaCodec* and *android.media.MediaPlayer* support video decoding with hardware acceleration. Compared to *MediaPlayer*, *MediaCodec* provides more flexible and low-level functions that can facilitate the testing of the targeted application. First, using *MediaCodec* together with *android.media.MediaExtractor* can ignore the audio stream of video files during decoding, reducing unnecessary computation. Conversely, *MediaCodec* dynamically decides when it should decode next frame. If the decoder is running concurrently with the video processing module, the video processing module may choose to skip some frames during its recognition due to the limited processing speed. For careful examination of the application, testers may want the decoder to wait for the video processing module in recognition (i.e., testing the application in a frame-by-frame way). In addition, *MediaExtractor* and *MediaCodec* can be used to decode audio streams from video files, together with video decoding. Unlike video decoding, audio decoding is not operated on GPU and may impose heavy load on CPU. With *MediaExtractor* and *MediaCodec*, a testing tool can replay context data for mobile applications that process both audio and video data. However, mechanisms are needed to synchronise the delivery of the two streams after decoding, since audio and video decoding may not be equally fast.

Fig. 5 shows the mechanism of video replaying by the CamTest replay thread which is imported as a component inside the targeted application. First, the replay thread uses *MediaExtractor* to read a video file which is specified in the storage of the mobile client. Unlike normal video decoders (e.g., *android.media.MediaPlayer*), we use *MediaExtractor* to extract video frames only and ignore audio streams. This can save both memory and CPU resources. Next, *MediaExtractor* sends extracted video frames to the hardware-accelerated (i.e., GPU-based) decoder that relies on *MediaCodec*. For applications processing video in the standard Android image format *YUV_420_888*, *android.media.ImageReader* can be used to pass each decoded frame the video processing modules. However, some applications, such as AR (augmented reality) applications, process video data in third-party image formats, because Android does not have corresponding native APIs for advanced graphical features. OpenCV [29] and Vuforia [41] are two popular frameworks of such applications. To support third-party image formats, a conversion mechanism after video decoding is designed. With imported libraries describing third-party image formats, buffered images in *YUV_420_888* can be transformed with regard to required colour spaces and storage formats (e.g., Mat objects of OpenCV in RGB colour space). Finally, the video processing module can analyse the captured frame pixels in the intended format.

4.3.2. Testing-oriented decoding of video frames

Based on *MediaExtractor* and *MediaCodec*, we have developed an algorithm for testing-oriented video decoding (i.e., the video processing module can receive and analyse these decoded frames) on the replay thread. Its pseudocode is shown in Algorithm . It supports two modes of replay: Realistic and Frame-by-Frame:

- The Realistic mode replays video frames at the real-world speed. Common smartphone cameras can capture video files at 30 frames per second. This rate is much higher than the video processing rate of applications, meaning that applications may ignore most of the frames (as they do in reality). Testers can use this mode for any type of testing.
- The Frame-by-Frame mode replays video frames as fast as the processing rate of the targeted application. The targeted application can read each frame without any loss. However, since common applications have low processing speed, the replay speed may be much slower than the captured speed. Testers can use this mode to carefully check the behaviours of the target applications. Also, the process speed of applications is not stable over the whole period

of runtime, causing nondeterministic performance (i.e., even if testers repeat tests with the same video replay in the realistic mode, the frames processed by the application may be different in each test). With such nondeterministic performance, it is difficult for testers to reproduce bugs in the realistic mode. Hence, to overcome this challenge, the Frame-by-Frame mode forces the replay thread to ensure that the video processing module analyses every frame regardless of its processing rate.

Compared to conventional video players, the decoding process of the CamTest replay thread is designed with three testing-oriented features:

1. Ignoring audio data: Video files usually contain both video and audio frames. The video processing module does not analyse audio data. Hence, during replay, the decoding process identifies and extracts video frames from the specified video file using *MediaExtractor*, ignoring audio data to reduce computational costs.
2. Interaction with the video processing module: The video processing module can interact with the replay thread at runtime to control the progress of the decoding process. This interaction is to support the Frame-by-Frame mode. As shown in Algorithm , the decoding process does not immediately decode and render the next frame after processing the current frame in the Frame-by-Frame mode. The video processing module should analyse the current frame first. After the analysis, the video processing module should inform the replay thread of the readiness (by calling the public function *getNext()* of the replay thread to change the *nextFrame* variable which controls the progress of decoding).
3. Providing an API for the acquisition of frame timestamps: Since the behaviours of the targeted application are related to the video content, it is necessary to record runtime testing results (e.g., bug report, assertion and performance events from image process output tracker, debugger and performance tracker) together with the frame timestamps of a video clip. Hence, the replay thread provides a public function *getFrameTime()* as the API for the targeted application to describe the accurate timing of a testing result. This function returns the timestamp of the currently decoded video frame in millisecond (i.e., the variable *Timestamp* in Algorithm).

Since CamTest replay service is implemented as an importable library, testers must add several API calls of this library into the source code of the targeted application to conduct video replay for testing. In Listing 3 (See Appendix), we show exemplary API calls for initialising the video replay in the source code of an application. During an ongoing replay, the video processing module can capture and analyse the rendered frames on the UI. To achieve this, testers also need to use several API calls from the source code of the video processing module. In Listing 4 (See Appendix), we list the exemplary API calls of delivering the decoded video frames to the video processing module in the source code of an application.

4.4. Testing functionality of CamTest replay service

Besides the feature of video replay, there are three kinds of testing functionality encapsulated in the library of the CamTest Replay Service: image process output tracker, debugger and performance tracker.

- Image process output tracker can record machine learning results and confidence values. It supports multiple concurrent entries of output on one frame. The output tracker can also record visual output generated by the video processing module.
- Debugger captures runtime assertions and exceptions. An assertion compares the expected value and the actual value of a specific variable. Exceptions may be generated from the operating system and the targeted application itself (i.e., testers can customise exceptions), such as "null pointer" and "divide by 0".
- Performance tracker measures how fast the targeted application processes one frame. The measurement is in milliseconds.

By calling the APIs of the CamTest library from the source code of the targeted application, testers can use the three kinds of testing functionality to examine the functional and non-functional properties of the targeted application. In Listing 5 (See Appendix), we list the exemplary API calls of using testing functionality in the source code of an application.

Besides text output, CamTest can also record visual output generated by the video processing module of camera-based mobile sensing applications for some complex tasks, such as object segmentation, object tracking and scene understanding. The recorded visual output can be exported as files which can be parsed to display the corresponding visual output on the video analyser, so that testers can conduct further analysis. In Listing 6 (See Appendix), we list the exemplary API calls of recording visual output generated by the video processing module of a camera-based mobile sensing application.

4.5. Video annotator and analyser

The video annotator and analyser are integrated as a Microsoft Foundation Class (MFC) PC application (Fig. 6). We implemented it using two popular open-source multimedia libraries: FFmpeg [38] and Simple DirectMedia Layer (SDL) [42]. With these two libraries, our implementation supports regular PCs even if they do not contain dedicated graphics

ALGORITHM 1: Decode and Display Image Frames to Replay a Video File

```
input  : Video File File and Replay Mode Mode and UI Graphics Renderer SurfaceTexture SurfaceTexture
output: Completion Signal Completion_Signal
// Start Replay
Timestamp ← 0;
Create a MediaExtractor object to open File;
Select only the video stream of File for MediaExtractor;
Create a Surface object as the image output buffer for SurfaceTexture;
Create a MediaCodec object with connection to Surface;
Create a Boolean variable nextFrame for the video processing module to report readiness in Frame-by-Frame mode;
nextFrame ← True;
while File not finished AND nextFrame is True do
    Current_Frame ← GetCurrentVideoFrame(MediaExtractor);
    Decode Current_Frame using MediaCodec;
    Timestamp ← GetFrameTime(Current_Frame);
    if Mode is Realistic then
        Deliver Current_Frame to Surface;
        // The video processing module analyses the frame on another thread without changing nextFrame
        Next_Frame ← GetNextVideoFrame(MediaExtractor);
        Timestamp_Next ← GetFrameTime(Next_Frame);
        Sleep (Timestamp_Next − Timestamp);
    else
        nextFrame ← False;
        Deliver Current_Frame to Surface;
        // The video processing module analyses the frame on another thread
        // If the video processing module is ready again, set nextFrame as True from that thread
    end
    MediaExtractor goes to next frame;
end
Stop and Release MediaCodec, MediaExtractor;
return Completion_Signal;
```

cards. Before a test, testers can use this tool to play the video file and provide ground-truth labels as annotations of the video content. On the right side of the interface, testers can add labels to either one frame or a set of consecutive frames.

After a test, the video analyser can read the test results from the mobile client. Then, testers can perform post-task analysis on the application output, errors and performance. The data entries of application output, errors and performance have timestamps corresponding to the related frames of the video file. When testers are playing the video file, ground-truth labels and the entries of application output, errors (including failed assertions and exceptions captured at runtime) and performance (i.e., processing speed of the video processing module) are displayed in the windows on the right side simultaneously according to the progress of video playing. Also, the screen can display the visual output generated by the targeted mobile application, together with the video frames. Hence, testers are able to analyse the states which the targeted application has entered at runtime.

Fig. 6 shows the examples of visual output supported by CamTest: dots, lines, rectangles and circles. Once a video frame is about to display, the video analyser renders the visual effects which should be visible at this moment (according to their duration and timestamps). Finally, the visual effects are displayed together with the video frame during the video play.

Fig. 7 shows the mechanism of displaying recorded visual output during the video playback on the CamTest video analyser. First, the *libavformat* library of FFmpeg extracts the audio and video streams from the video file. Then, the *libavcodec* library of FFmpeg decodes the audio and video frames using the corresponding decoders. The decoded audio and video data are sent to the audio and video buffer. Before playing the audio and video content, the CamTest video analyser has to display recorded visual output which should be visible at this moment. During the initialisation of a video playback, the output parser extracts recorded visual output from the file exported from the CamTest mobile client. Next, the extracted visual effects, which can be a number of dots, lines, rectangles and/or circles, are sent to the corresponding renderer. Then, the colour converter transforms the colour format of the visual effects from RGB (used by Android Canvas) to YUV420P (used by SDL). After colour format conversion, the pixel editor embeds the visual effects into the video frame saved in the video buffer. Finally, the SDL library plays audio and video content from their buffers to the computer hardware. Since FFmpeg and SDL support both CPU and GPU to perform video processing for PC platforms, the CamTest video analyser can operate on low-cost PCs without a dedicated graphics card.

5. Evaluation

Due to the wide-ranging functionality of CamTest, we decided to conduct a multi-pronged evaluation:

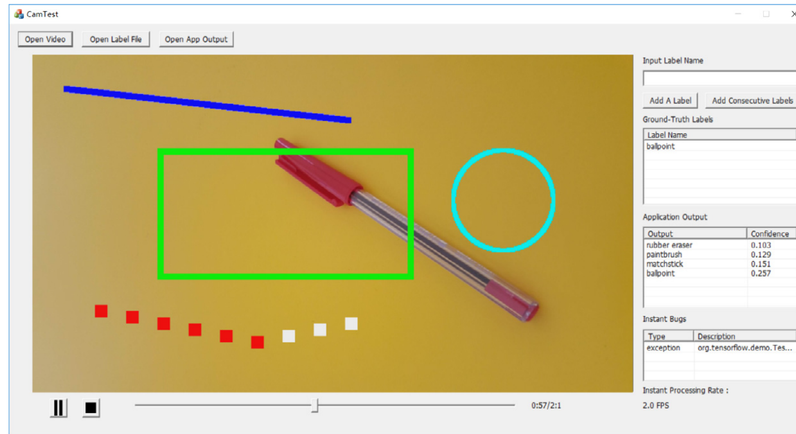


Fig. 6. Examples of visual output supported by CamTest: dots, lines, rectangles and circles.

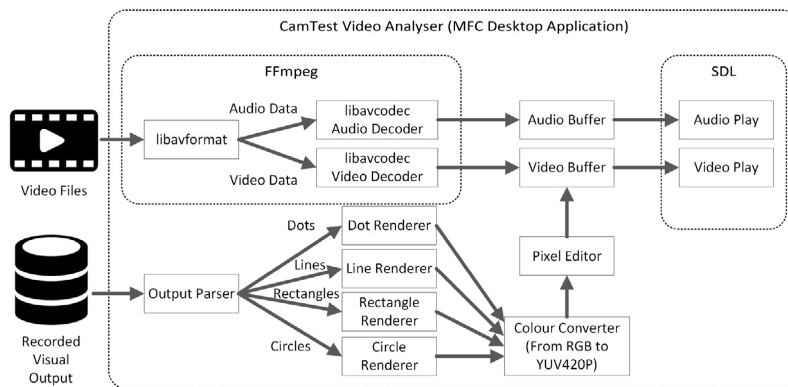


Fig. 7. The mechanism of displaying recorded visual output during the video playback on the CamTest video analyser.

1. a **case study** with mobile devices and a real-world object recognition application. This study first evaluates the video replay speed of CamTest replay service on different mobile devices. Then it shows the workflow of all the CamTest components in testing a real-world object recognition application.
2. a **computational efficiency experiment** on actual mobile handsets. This experiment compares the CamTest replay service with other approaches in terms of CPU, memory and battery usage.
3. a **user study** with 14 professional and experienced mobile application developers. This study investigates the usefulness and usability of CamTest. Also, this study collects the suggestions from these developers comparing CamTest to other testing tools that they have used.

5.1. Case study

To demonstrate the usefulness and performance of CamTest we consider a case study. We first measure the maximal replay speed of the replay service on different devices. Then, we test a real-world object recognition application using CamTest.

Since smartphone cameras can record video at 30 frames per second, it is necessary for CamTest to achieve the same rate to support realistic testing. Due to the device heterogeneity, we measured the maximal replay speed by running CamTest with an identical video file (size: 1920×1080, rate: 30 frames per second) on 6 off-the-shelf smartphones: Motorola Moto G4, HuaWei GR5, Samsung J5 $\times$ 2, Samsung S6 Edge, Samsung S7. Fig. 8 shows the results on each phone. All the phones were able to achieve a rate more than 30 frames per second, however, with a huge difference: the best performance 193.12 on S7 and worst 57.87 on G4. The results on identical phone models were very close: 68.57 and 69.28 on 2 instances of J5.

Selected application. There exist a small number of open-source camera-based mobile sensing applications on the Internet that we can use for testing. We selected an object recognition application: the TensorFlow Demo for Android [33]. The purpose of the application is to recognise the objects captured by camera in real time. Fig. 9 gives an example of the

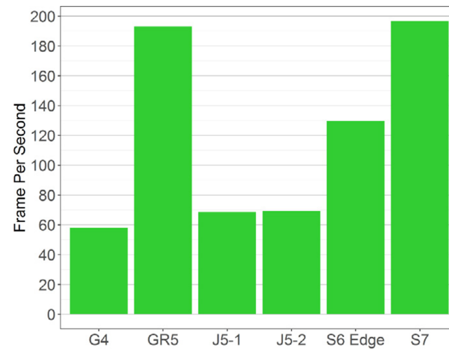


Fig. 8. The maximal replay speed on different smartphones.

Fig. 9. The interface of TensorFlow Demo for Android.

interface of TensorFlow Demo for Android. The recognition results and confidence values are displayed within the interface at runtime.

Test process. The test aims to evaluate both functional (bugs) and non-functional (machine learning output and performance) properties of the application. First, we recorded a video clip with default settings of the camera using a smartphone (size: 1920 $\times$ 1080, rate: 30 frames per second). The video contains everyday objects, including a mouse, ballpoint pen, and toilet roll. We manually annotated the video with labels using our video annotator. We added some customised assertions and exceptions to its source code, to simulate the debugging efforts. Then, we configured the replay task (set to realistic mode) on the mobile client and launched the TensorFlow application on the real device. Finally, we exported these results from the mobile client.

Results playback. In this stage, the tester should perform a post-hoc analysis of test results. Of course, it is possible to calculate the accuracy of TensorFlow by comparing all ground truth labels to all outputs of the TensorFlow application. This would give a single measurement of accuracy, which could be used to benchmark performance as the application source code is updated. However, this crude approach is not enough to gain a granular understanding of the application's behaviour. For example, perhaps a mouse was visible in the video for 10 s, but the application only detected 8 s of that; perhaps the application has a delay in detecting certain types of objects. These are low-level performance issues that can actually be interrogated with our tool. Hence, the video analyser displays the video content with all the runtime states of the application so that the tester can view the whole progress of the application execution.

To investigate in detail TensorFlow's behaviour, we can load the exported results (i.e. the timestamped labels generated by TensorFlow), and our video, into our video analyser shown in Fig. 10. As the video analyser plays back the file, it displays the ground-truth labels (provided by the tester), application output (i.e., timestamped object recognition results and confidence values), runtime errors (i.e., failed assertions and caught exceptions), and processing speed in frames per second. The displayed information updates during playback. This information can be used to inspect the behaviour of TensorFlow with frame-level accuracy.

Finally, since typical camera-based mobile sensing applications have a processing rate much lower than real-world camera recording rate (e.g., 30 frames per second in most cases), these applications process a subset of the frames and ignore the rest. Also, the process rate of applications is not stable over the whole period of runtime, causing nondeterministic performance. To demonstrate this problem, we repeated the test twice using the TensorFlow Demo for Android on Samsung S6 Edge. Table 4 shows the first 10 frames processed by the targeted application in each of the two tests. Although the device and video file were identical, the targeted application operates differently, and actually

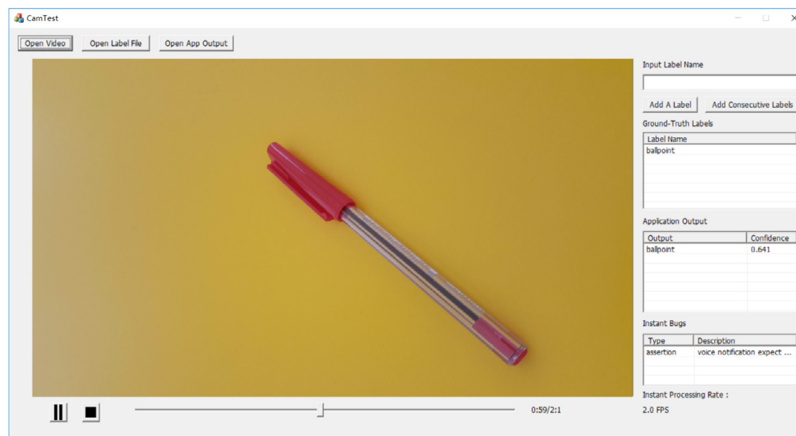


Fig. 10. Analysis of results on the CamTest video analyser.

Table 4
First 10 frames processed by the target applications. Two independent tests were conducted.

Test	Frame number
1	37, 45, 53, 60, 67, 75, 82, 89, 97, 104
2	36, 50, 61, 70, 79, 88, 97, 106, 115, 124

processed different frames of the video in each of the two tests. This means that even if one test finds a bug, the next test cannot effectively reproduce it using the realistic replay. Hence, researchers should replay the video in Frame-by-Frame mode to ensure that the targeted application processes every frame of the video. Then, the processed frames are deterministic.

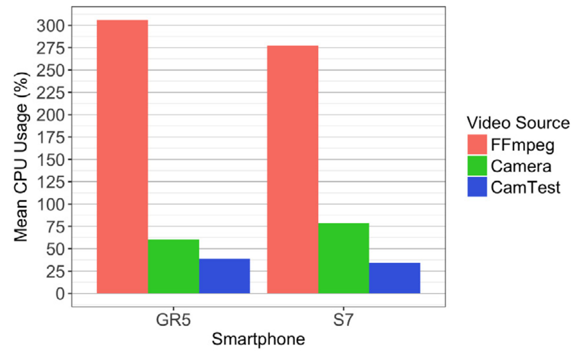
5.2. Computational efficiency

Since multimedia-related computation is generally intensive on mobile devices, we conducted an experiment to evaluate the computational efficiency of video replaying by CamTest compared to other approaches. To support the testing of camera-based mobile sensing application, testers need a way to deliver the video frames to the application, such as the CamTest replay service. The common alternatives to CamTest replay service are FFmpeg video decoder, or using camera to collect real-time video stream (i.e., in-the-wild testing). Thus, the experiment compared these three approaches: CamTest replay service (in realistic mode), FFmpeg video decoder, and camera. We used two smartphones with the same operating system Android 7.0: HuaWei GR5 and Samsung S7. They were disconnected from WiFi, Bluetooth and cellular networks. Their power saving modes were off and had maximal screen brightness.

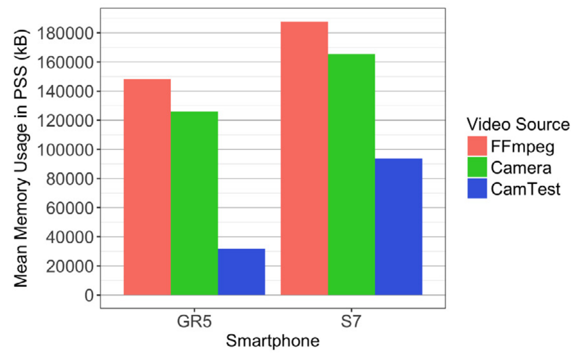
In the experiment, CamTest and FFmpeg performed video decoding and rendering on the same video file. When using camera, the two smartphones were capturing and displaying images from the camera using Android API *android.hardware.camera2* without saving the video to storage. We used the *dumpsys* tool from Android Debug Bridge (ADB) (i.e., the debugging feature available on Android phones) to measure CPU, memory and battery usage. Specifically, for CPU and memory usage, the measurement was the average use by the application process in 5 min. The phones were disconnected from USB (i.e., not charging).

CPU usage. Fig. 11a shows the average CPU usage of the three approaches on the two handsets. Due to its CPU-based decoding mechanism, FFmpeg decoder had the highest CPU usage (305.8% on GR5, 277.5% on S7), beyond the capability of a single CPU core (GR5 and S7 have 8 cores). Comparatively, the approach of using camera to collect real-time video frames achieved a more tolerable level of CPU usage (60.6% on GR5, 78.6% on S7). Our CamTest replay service had the lowest CPU usage (39.2% on GR5, 34.4% on S7).

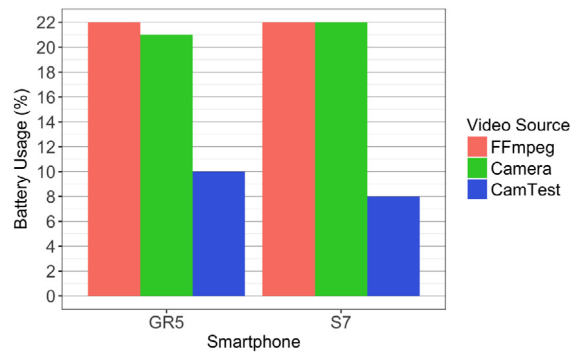
Memory usage. Fig. 11b shows the mean memory usage in proportional set size (PSS). Since a process may share memory space with other processes (e.g., these processes are using the same system service), PSS is a widely used measurement to reflect the amount of private memory, as well as the proportion of shared memory used by a process. FFmpeg decoder had the highest memory usage (148,300.6 kB on GR5, 187,680.8 kB on S7). The approach of using camera also used large memory space (125,760.2 kB on GR5, 165,669.2 kB on S7). In contrast, CamTest replay service achieved the lowest memory usage (31,928.0 kB on GR5, 93682.6 kB on S7). Specifically, the GR5 has 3 GB memory, while the S7 has 4 GB. As the memory management strategy on Android performs differently on different hardware conditions [43] , the same approach required much larger memory space on S7 than on GR5.



(a) Mean CPU usage (%)



(b) Mean memory usage in PSS (kB)



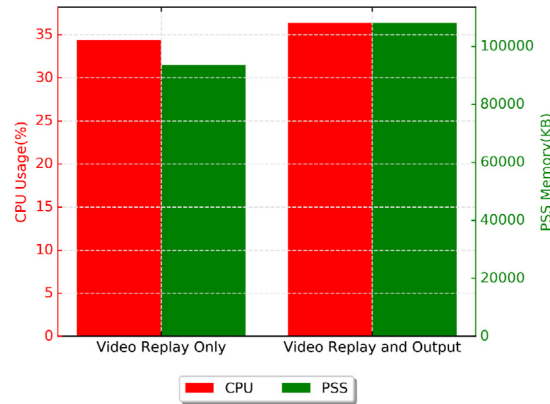
(c) Battery usage (%) of one-hour continuous executions

Fig. 11. Results of the computational efficiency experiment.

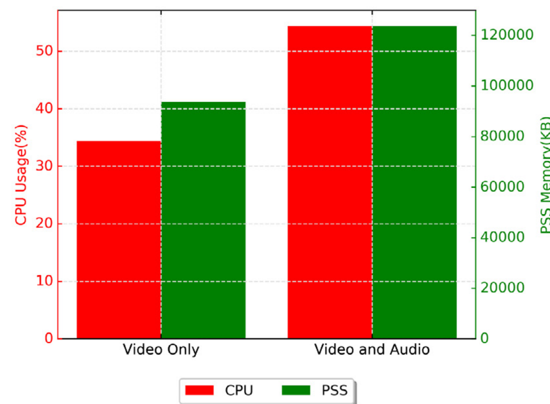
Battery usage. Fig. 11c depicts the battery usage of one-hour continuous execution. FFmpeg decoder was the most power-hungry: 22% of overall battery level on both GR5 and S7. Also, the approach of using camera had a similar rate of battery usage: 21% on GR5, 22% on S7. By contrast, CamTest replay service was significantly more energy-efficient: 10% on GR5, 8% on S7.

In addition, we conducted two experiments to quantify the overhead of recording output (e.g., machine learning predictions, debugging assertions) and replaying audio streams during video replay with CamTest on S7. During the replay of a video with 30 frames per second, CamTest was configured to generate all types of output (before recording runtime exceptions, it generated customised exceptions first) for each frame. We measured the additional CPU and memory usage during the replay.

Fig. 12a shows the extra overhead of recording output during video replay, using the video replay without output as a contrast. For recording output, the CPU usage increased only 2% and PSS memory increased only 14 499 kB. Fig. 12b



(a) Overhead of recording output during video replay



(b) Overhead of replaying audio with video

Fig. 12. Results of experiments on overhead of recording output and replaying audio (on S7).

shows the extra overhead of replaying audio together with video. The impact of including audio into replay is significant: 20% more CPU usage and 30051 kB more PSS memory usage. As the audio streams cannot be accelerated by hardware (e.g., GPU), CPU received all the cpu workload of processing audio data.

5.3. User study

To further evaluate CamTest, we conducted a user study with 14 professional mobile application developers (3 females, 11 males, mean age 27.6, mean work experience 3.3 years). The sample size is sufficient for our purposes, since widely accepted practices have 12 to 16 participants [44]. The 14 participants have at least 1-year work experience and at least hold a bachelor's degree in computer science or software engineering. Every participant was compensated with 30 Australian dollars.

We conducted the study with each participant individually. We first briefed each participant about CamTest and the background information of camera-based mobile sensing applications. Next, using a real smartphone connected to our PC, we showed a demonstration on how to use all the features of CamTest. Then, participants were asked to use such features for any purpose that they like. They could use these features with the camera-based sensing application that was used for the demonstration: the TensorFlow Demo for Android [33] which is designed to recognise objects. During the testing period, we gave the documentation and example code of CamTest to participants. Also, they were allowed to ask us for help if necessary.

After testing was completed, we asked each participant to answer a questionnaire with 7 questions about the usage and features of CamTest. Regarding each question, they were instructed to choose a score from 5 (strongly agree), 4 (agree), 3 (neutral), 2 (disagree), 1 (strongly disagree). Then, they were asked for a detailed explanation of scores based on their

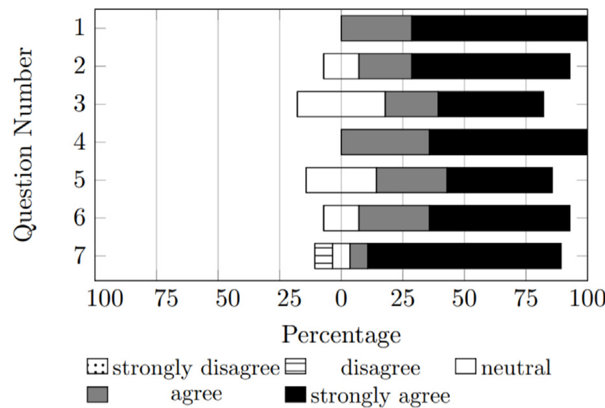


Fig. 13. The responses of participants.

professional experience, so that they could not have a bias from external settings. Finally, they were asked to give some suggestions, if any, by comparing CamTest with other testing tools that they had used. The study lasted for 1.5 to 2 h per participant. The questions and responses are described as below. Also, the scores for each question are shown in Fig. 13.

Q1: Is the CamTest video replay useful for testing the application? (mean score: 4.71, median score: 5, standard deviation: 0.45) This question was to investigate the usefulness of the video replay feature provided by CamTest (related to the mobile client and replay service). The scores indicate that all participants perceived the usefulness of the video replay feature. Most participants stated that replaying videos as test cases can ensure the reproducibility of bugs and reduce testing cost. In terms of the reproducibility, P10 stated: "*Software companies commonly ask testers to report a bug together with the way to reproduce it. With the video replay of CamTest, testers can just provide a video file to reproduce the bug*". Regarding testing cost, P3 said: "*Software companies run smoke testing and endurance testing to check whether applications can correctly operate for significantly long time. The cost on such tests is very high. Video replay is good way to reduce cost*".

Q2: Is the Frame-by-Frame mode a useful feature? (mean score: 4.50, median score: 5, standard deviation: 0.73) This question was to confirm the usefulness of the Frame-by-Frame mode in video replay. The scores show that the majority of participants considered it as a useful feature. P4 stated: "*Examining every frame can carefully test the application, although the testing process takes longer time*". In terms of test coverage, P10 stated: "*Testers can run both realistic and Frame-by-Frame mode to increase the coverage of testing. With higher test coverage, more bugs can be found before deployment. Fixing a bug after deployment is costly*". However, the high standard deviation indicates disagreements among them. Regarding the relation between frames and bugs, P2 stated: "*This mode does not help testers when the bugs are caused by a specific subset of all frames*".

Q3: Is the CamTest video replay easy to use? (mean score: 4.07, median score: 4, standard deviation: 0.88) This question investigated the usability of the CamTest video replay. The scores indicate that most participants considered it convenient. P5 stated: "*Since the video replay feature works as a library, the efforts of using it are minor. Testers can easily configure tests using the mobile client*". P6 stated: "*The required code is easy to write. Most testers have enough coding skills to use it*". P12 stated: "*The work flow of using this feature is simple. I understood it after seeing the demonstration*". Meanwhile, the high standard deviation reflects the wide variety of opinions. P2 stated: "*Not all testers can write code*". P14 stated: "*When adding the required code, even testers with coding skills may make mistakes*". Regarding large-scale software testing, P7 stated: "*Configuring a large number of tests using the mobile client is still time-consuming*".

Q4: Is the test functionality of CamTest useful for testing the application? (mean score: 4.64, median score: 5, standard deviation: 0.48) This question investigated the usefulness of the test functionality in CamTest replay service which is encapsulated as a code library. The scores reveal the endorsement among all participants. P4 stated: "*Testers can record runtime states of the application with video frame timestamps. Other tools cannot link the testing results with video frames*". P12 stated: "*The APIs of testing functionality can automatically record the required results. It is good for long-term tests so that testers do not have to stare at the output console of a testing tool*". Regarding the completeness of testing, P5 suggested: "*CamTest only records what the tester specifies. It may be good to also use other tools that can examine some low-level aspects neglected by testers*".

Q5: Is the test functionality of CamTest easy to use? (mean score: 4.14, median score: 4, standard deviation: 0.83) This question was to examine the usability of the test functionality in CamTest. The scores show that most participants were satisfied with its usability. P3 stated: "*It is simple. One line of code can record a kind of information*". P7 stated: "*The APIs are easy to use, as they have a style similar to Android APIs*". However, as the high standard deviation shows, some participants did not consider it convenient. P8 stated: "*I must check each parameter of an API by reading documentation*". P10 stated: "*It requires considerable effort to do all kinds of tests with this*". P9 stated: "*It would be better if the tool can recommend me all the parameters. Many IDEs have this feature*".

Q6: Is the video annotator/analyser of CamTest useful for testing the application? (mean score: 4.43, median score: 5, standard deviation: 0.73) This question was to investigate the usefulness of the video annotator/analyser (integrated as a PC-based tool) in CamTest. The scores indicate that most participants considered it as a useful part of CamTest, apart from the parts designed for the mobile platform. P3 stated: "*This tool simultaneously shows the video scenario and runtime states of the application. It is easier to find problems compared to other tools only having a log viewer*". Regarding exceptions, P7 stated: "*The stack trace in the exception description is very important. Testers can quickly locate a bug in the source code*". In contrast, the high standard deviation reveals the diversity of opinions. Regarding the flexibility of application logs, P2 stated: "*Testers may want more information than that on the interface. Testers sometimes customise application logs*". In terms of the exception coverage, P11 stated: "*Common testing tools check some exceptions without commands of testers. If testers do not specify these exceptions during runtime recording, CamTest will not show them*".

Q7: Is the video annotator/analyser of CamTest easy to use? (mean score: 4.57, median score: 5, standard deviation: 0.90) This question was to check the usability of the video annotator/analyser in CamTest. The scores indicate that the majority of participants were satisfied with its usability. Regarding the controls of opening video and testing files, P3 stated: "*It is easy and quick for testers to open and play the video and results*". In terms of understanding output on the UI, P14 stated: "*Testers having some machine learning experience should quickly understand the displayed output*". Also, P4, P7, P10 and P12 agreed that testers with a little experience in mobile development and machine learning can understand and use it without training. However, the high standard deviation indicates disagreement. P2 stated: "*Testers need to read the documentation to use it*". P5 and P7 argued that a training session is necessary for testers to understand each item on the UI. Regarding the exception display, P12 suggested: "*It would be better if exceptions are previewed in the progress bar*".

Suggestions from comparison between CamTest and other tools: All participants reported that the video-oriented features of CamTest can help testers test camera-based mobile sensing applications more or less. And they pointed out that existing testing tools cannot provide similar help. However, some of them suggested that some features of other testing tools can inspire the testing of camera-based mobile sensing applications.

1. **Android studio.** Comparing CamTest to Android Studio, P4 stated: "*CamTest records exceptions specified by testers. Testers must know and define all required exceptions beforehand. On Android Studio, although redundant, all system exceptions are captured and shown. Testers can apply a filter to search exceptions*".
2. **Gprof.** Regarding performance testing, P1 compared CamTest to Gprof: "*CamTest only measures the main machine learning algorithm. Linux has Gprof to measure the call count and execution time of functions, including its children functions. However, CamTest can measure the performance change in the whole replay period. Gprof does not connect the performance with video moments*".
3. **Bugzilla.** In terms of bug tracking, P11 found Bugzilla has a better mechanism than CamTest: "*CamTest shows bugs in the video analyser. However, CamTest does not track a bug in the whole process of testing and repair. Bugzilla can help testers transfer a bug to responsible developers. It also records the repair state of a bug, such as fixed or to be fixed in next version*".
4. **Firebase test lab.** Regarding large-scale testing, P12 compared CamTest to Firebase Test Lab: "*CamTest can only run tests on a real device owned by the tester. On Firebase Test Lab, testers can concurrently deploy their applications to a large number of different devices on the cloud. Then they can write test scripts to automate tests. After tests, testers can check bugs and logs on every single device. It is cheaper and faster to find bugs that only happen on certain hardware or OS. However, Firebase Test Lab cannot simulate video content to test camera-based applications*".
5. **IBM rational test workbench.** Also regarding large-scale testing, P13 compared CamTest to IBM Rational Test Workbench: "*Testers must copy a video file to a device for CamTest to replay. The distribution of video files takes long time for testing on many devices. IBM Rational Test Workbench can share the test cases from one device to other devices, although it only supports the sharing of UI test scripts*".

6. Discussion

Based on the empirical evidence of our implementation and evaluation, we discuss the lessons, implications and limitations of using CamTest to test camera-based mobile sensing applications in laboratory settings.

6.1. Performance

To test camera-based mobile sensing applications, it is important for our tool to replay video with a sufficiently fast rate which is no less than the camera recording rate (e.g., 30 frames per second in most cases). Hence, we measured the replay speed on actual physical devices. Our study results show that our approach can achieve this rate on a variety of off-the-shelf smartphones (Fig. 8).

At this rate, researchers can conduct realistic tests on their applications using replayed video images, rather than actual camera images. During real-world speed tests, Algorithm can ensure that the replay frame rate is exactly the same as the camera recording rate. Also, to test applications that process video and other sensor data, CamTest can be used together with another tool that replays other sensor data. For example, testers can import both the CamTest replay service and TestAWARE [5] library into the mobile application. When launching a replay, the speed of TestAWARE must be set to the real clock so that the video stream and other sensor data can be synchronised.

We used CamTest to conduct a complete process of testing on a real-world camera-based mobile sensing application: TensorFlow Demo for Android [33]. We found that it is straightforward to import the replay service into TensorFlow Demo. Although this approach precludes black-box testing (i.e., without changing source code), the change of the targeted application's code is minor: calling the replay service, and stopping the camera connection. To make use of the debugging functionality provided by the replay service, researchers can insert customised assertions and exceptions.

Once the testing is complete, researchers can analyse the results using the video analyser. We found that the video analyser (Fig. 10) can correctly trace the runtime states of the targeted application at every moment of the video image processing. Compared to the actual targeted application, the video analyser offers additional information for researchers to understand the runtime state, including ground-truth labels, assertions, exceptions and the processing rate in FPS (frames per second). This information is crucial in helping researchers examine both functional and non-functional properties for their application.

If the application has a processing rate lower than the camera recording rate, the application will only process some of video frames and ignore the rest. In our case study, we show that nondeterministic performance on physical devices increases the difficulty of bug reproduction. To address this issue, researchers can use Frame-by-Frame mode to replay video frames at the same rate of image processing of the targeted application. Once the replay service delivers a frame, it will suspend the next frame until the application finishes the current frame. Hence, the application can read all the frames of the video file without loss. And researchers can reproduce bugs related to the same video file in different tests.

Finally, inefficient design or implementation can affect the longevity of hardware components. Hence, regarding CPU, memory and battery usage, we compared the video replay of CamTest to other approaches (i.e., FFmpeg and using camera directly) on real devices. The experimental results show that the video replay of CamTest has significantly lower impact on CPU, memory and battery than other approaches (Fig. 11a, 11b and 11c). This finding indicates that CamTest can reduce the cost of hardware maintenance, since high temperatures caused by inefficient computation increase the risk of hardware damage [45]. Also, by consuming less energy, CamTest can reduce the energy cost in testing. Specifically, this reduction can be significant in scenarios such as smoke testing, endurance testing (i.e., tests with very long time on devices) and large-scale testing (i.e., tests with a large number of devices).

6.2. User study findings

The user study investigated the usefulness and usability of the different components in CamTest. The interview results indicate that the features of video replay, Frame-by-Frame mode, test functionality and video annotator/analyser are effective for the testing of camera-based mobile sensing applications. All participants identified the lack of tools for such testing, and considered CamTest as a necessary, useful and novel tool to bridge this gap. Multiple improvements were also suggested.

All participants felt that the video replay is a useful feature. Specifically, they highlighted that CamTest can ensure the reproducibility of bugs and reduce testing cost by replaying video files as test cases. Regarding the Frame-by-Frame mode, they confirmed its usefulness in performing careful examination on the application using every frame in the video file. They also suggested running tests with both realistic and Frame-by-Frame mode for a higher test coverage which can potentially find and fix more bugs before deployment.

In terms of the test functionality, all participants considered it as a useful feature to examine the application at runtime. They noted that connecting runtime states with the video timestamps is helpful in understanding how the application operates in different scenarios. However, they pointed out that testers have to manually specify the aspects of the examinations on the application. If testers forget to specify some kinds of runtime states, CamTest will not automatically perform these examinations. Although following user specifications can improve efficiency and reduce redundant information in testing, it affects the completeness of testing.

Regarding the video annotator/analyser, most participants confirmed its usefulness in displaying the video scenario simultaneously with runtime states of the application. They highlighted the simplicity to find problems using it, such as locating bugs in source code via the stack traces of recorded exceptions. However, one participant would like to have higher flexibility in the support of customising application logs, rather than only using those provided by CamTest. Also, one participant argued that it may be helpful to display all system-based exceptions without explicit specifications.

In terms of perceived Usability, there was some minor disagreement, but participants generally considered all the features of CamTest easy to use. Regarding the video replay feature, most participants considered it convenient to import the library and configure tests on the mobile client. They thought that the required code for launching video replay is not a challenging task for most testers who have coding skills. However, some argued that there exist testers who cannot write code, and that testers having coding skills may make mistakes. Besides, one participant pointed out that testers cannot quickly configure a large number of tests using the mobile client.

Regarding the test functionality in CamTest, the feedback was similar. Most participants were satisfied with the API style (similar to Android's APIs) and simplicity. In contrast, some participants found it hard to add parameters to the APIs without carefully reading the documentation. Inspired by some existing IDEs, one participant suggested an additional feature of automatic parameter recommendation for the APIs.

Regarding the video annotator/analyser, most participants agreed that it is easy and quick for testers to use it. They felt that testers with some experience in mobile development and machine learning can use it without training. However,

some participants contended that a training session and documentation are necessary to fully understand its usage. Moreover, one participant suggested showing the previews of exceptions in the progress bar so that testers can quickly browse all exceptions with video playback.

Finally, our study asked participants to compare CamTest and other testing tools that they had used. Some participants identified several useful features in other tools and provided suggestions to CamTest for improvement in testing camera-based mobile sensing applications. We summarise these suggestions in Section 6.4.

6.3. Implications for testing camera-based mobile sensing applications

Testing camera-based mobile sensing applications requires suitable video content as test cases to exercise the programs. Collecting such video content using the camera in real-world tests can bear high costs, especially for regression and long-term testing. Also, contexts can be highly dynamic in the real world, lowering the reproducibility of bugs. CamTest enables laboratory tests where testers can use a video file as the test case to drive the application. Testers may either use the device camera to record or download (e.g., from online repositories) relevant video files, minimising the cost. With video files, testers can always repeat tests without capturing video content using the camera in the real world. For multiple rounds of tests or long-term tests, testers can significantly save time and reduce the test budget. Moreover, since the video content is stable in a video file, laboratory tests with CamTest can ensure the reproducibility of bugs. For applications based on the standard Android image format *YUV_420_888*, CamTest can directly deliver the decoded frames which are also in the same format. Besides, for applications based on third-party image formats (e.g., Mat in OpenCV), a conversion phase from decoded frames (*YUV_420_888*) to the third-party format is needed. Hence, the simplicity of CamTest setup and computational cost vary among applications based on third-party image formats. Since major third-party libraries (e.g., OpenCV on Android) provide specialised functions to support such format conversion, the additional setup efforts are insignificant.

Additionally, it is important to connect test results to video playback. Since the output of applications relies on the video input, connecting test results with the video content is necessary for analysis. During testing, CamTest can record runtime states of the targeted application with the timestamp of the video frame having been processed. After a test, the video analyser of CamTest can show the recorded runtime states simultaneously with the video playback. Hence, based on the connection between test results and video playback, testers can understand the context affecting the functional and non-functional properties of the application.

Furthermore, it is important to be able to conduct tests on actual handsets, like with CamTest. This way, testers can verify whether their applications can successfully operate on certain hardware and operating systems. Also, real device testing can accurately reveal application performance on the intended hardware. Although some properties may also be tested on PC-based tools, migrating applications gives rise to extra cost and compatibility issues. For example, a mobile application with an ARM-oriented third-party prebuilt library cannot run on x86 hardware (most PCs). Considering the feasibility, realism and cost of testing, we believe that conducting testing on actual handsets is necessary, although PC-based tools may also be used for some specific testing tasks. Similarly, in some cloud-based mobile testing tools (e.g., Firebase), real device testing is available.

Another implication relates to computational efficiency. As multimedia-related computation is often intensive on mobile devices, computational efficiency is an important concern for the testing of camera-based mobile sensing applications. Inefficient testing tools consuming unnecessarily high computing resources increase energy cost and the risk of hardware damage. To improve the computational efficiency, CamTest ignores the audio stream of video files and processes video frames using hardware acceleration. Regarding CPU, memory and battery usage, experimental results show that CamTest has significantly lower overhead than FFmpeg and using camera. Hence, CamTest is a low-cost and environmentally friendly solution.

Finally, designing a laboratory-based testbed for camera-based mobile sensing applications involves 5 goals (as listed in Table 1): Annotating Video Content, Replaying Video Streams, Logging App Output, Tracking App Performance, and Displaying Results. To fulfil these goals, we designed the architecture of CamTest (Fig. 3). Although our case study shows that our approach is effective, there may exist better approaches to achieve the same goals. Our replay mechanism aims at white-box testing, meaning that researchers have to change the source code of the targeted application to launch tests using CamTest. If the operating system is rooted, replaying data without changing the targeted application is possible [18]. However, rooting an operating system is a laborious and technically difficult task for most researchers. Considering that our white-box approach requires insignificant change within the targeted application, we implemented CamTest which can launch tests without requiring a rooted operating system. When designing similar testing tools, developers/researchers should consider the effort in both using and setting up testing tools.

6.4. Limitations and future work

Based on our implementation and evaluation, we identified several limitations of CamTest. Regarding catching exceptions, CamTest can only record exceptions specified by testers. If testers forget to specify one kind of exception, CamTest will not record them. However, tools such as Android Studio automatically capture all system-level exceptions, allowing testers to find relevant ones using a filter. Future work can investigate how to automate the recording of runtime

system-level exceptions with the video timestamps during testing. Also, CamTest focuses only on the performance of the main machine learning algorithm. Considering that applications may have multiple functions in algorithm design, measuring the call count and execution time of each function can provide better insights for testers to understand application performance. Regarding large-scale testing with a variety of devices, CamTest cannot automatically duplicate video files and test configurations from one device to the rest. Supporting efficient and low-cost large-scale testing of camera-based mobile sensing applications is a worthwhile task in future work. Furthermore, CamTest requires testers to have a suitable physical device to conduct tests. Future work may also attempt to establish a cloud-based testing platform hosting a large number of physical devices so that testers without devices can conduct tests remotely. To test mobile applications that process both audio and video streams, future work can investigate how to replay the two streams simultaneously with speed difference between audio and video decoding. On Android, most mobile applications use only one camera and can be tested with CamTest. However, from Android 9.0, mobile applications may use two or more cameras. On these new Android devices, future work can extend CamTest to design multiple threads for video decoding and synchronised delivery in the testing of multi-camera applications.

For test result analysis, the video analyser of CamTest displays information captured by the replay service only. During testing, testers may want to customise some application-specific logs beyond CamTest. Future work can refine the video analyser by adding a suitable interface to show application-specific logs. Another issue is that CamTest does not support visual annotations, as the mixture or overlap of visual annotations and output may confuse testers. Future work can study how the video screen can simultaneously display visual annotations and output without confusion. Also, in the video analyser, it would be useful for the progress bar to display previews of exceptions so that testers can quickly browse exceptions. Future work can study how the previews can be generated and presented. Regarding bug tracking, CamTest does not allow testers to transfer bugs to responsible developers. Future work can design a suitable solution for testers to transfer and manage bugs detected in testing.

7. Conclusion

In this paper, we present CamTest, a novel laboratory-oriented testbed for camera-based mobile sensing applications. It enables developers to conduct systematic, affordable and efficient tests on their applications in laboratory settings. Instead of real-world tests, CamTest delivers the frames of a video clip, rather than camera images, to the image processing module of the targeted application. Meanwhile, CamTest records the image processing output, errors and image processing speed of the targeted application at runtime. After testing, CamTest can display the recorded information simultaneously with the video playback so that testers can understand how the context affects behaviours of the application. Through a case study, computational efficiency experiment and user study with 14 developers, we show that CamTest can effectively facilitate and simplify the testing of camera-based mobile sensing applications.

Conflict of interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Appendix. Listings of exemplar APIs

As in Listing 3, to conduct a test with CamTest, the targeted application should contain an *Activity* that implements the interface *android.view.TextureView.SurfaceTextureListener* so that *Activity* can render its video images on the UI via *TextureView*. Inside the *onCreate()* function of this *Activity*, testers should first create an instance of *TextureView* as the UI container using the corresponding element of the UI layout (line 3). Then, testers should use the function *setSurfaceTextureListener()* to connect the activity to the graphics renderer *SurfaceTexture* of the instance of *TextureView* (line 4). Next, testers should use a public static function *getCurrentVideo()* of the class *CamTest_Replay* to fetch the path of the specified video file from the storage of the mobile client (line 5). The access to this storage is provided by the *android.content.ContentProvider* of the mobile client. Such access requires the global information of the current application environment. Thus, the function *getApplicationContext()* is used to provide the global state of the targeted application. Then, testers can create the replay thread instance using the class *CamTest_ReplayThread* provided in the CamTest library (line 6). Since the decoded frames must be rendered on the UI, the initialisation of the replay thread must be in the function *onSurfaceTextureAvailable()* which means that the graphics renderer *SurfaceTexture* of *TextureView* on the UI layout is ready for frames. Testers should initialise the replay thread using three parameters (line 8): the object *surfaceTexture* (i.e., an instance of *SurfaceTexture*) which is provided by the function *onSurfaceTextureAvailable()*, the path of the video file (provided by the function *getCurrentVideo()*), and the mode of the replay task (provided by the function *getCurrentVideo()*). In addition, testers should start the video processing module on another thread from the function *onSurfaceTextureAvailable()*. Finally, in the function *onSurfaceTextureAvailable()*, testers can start replay by calling the function *start()* of the thread (line 10).

```
1   //In an Activity implementing TextureView.SurfaceTextureListener
2   //In function onCreate
3   UI_Texture = (TextureView) findViewById(R.id.textureView);
4   UI_Texture.setSurfaceTextureListener(this);
5   videoPath = CamTest_Replay.getCurrentVideo(getApplicationContext());
6   replayThread = new CamTest_ReplayThread();
7   //In function onSurfaceTextureAvailable with parameter ‘‘surfaceTexture"
8   replayThread.init(surfaceTexture, videoPath, CamTest_Replay.mode);
9   //Start the video processing module
10  replayThread.start();
```

Listing 3: Exemplary API calls of initialising the replay feature of CamTest in the source code of an application

As in Listing 4, for sending the decoded video frame to the video processing module, the interface *TextureView.Surface TextureListener* has the function *onSurfaceTextureUpdated()* which is triggered once a new frame is rendered. In this function, testers can fetch the pixels of the rendered frame using the function *getBitmap()* of the object of *TextureView* (line 3). Then, testers can pass the frame pixels to the video processing module running on another thread (line 5). In the Frame-by-Frame mode, after processing one frame, the replay thread does not decode and render the next frame unless the video processing module requests it. Such request can be done by calling the public function *getNext()* of the replay thread (line 7).

```
1   //In the Activity implementing TextureView.SurfaceTextureListener
2   //In function onSurfaceTextureUpdated
3   frame = UI_Texture.getBitmap();
4   //On another thread
5   Result rst = video_processing_module.recognise(frame);
6   //In Frame by Frame mode, it must ask replayThread to send next frame
7   replayThread.getNext();
```

Listing 4: Exemplary API calls of sending the decoded video frame to the video processing module of an application

As in Listing 5, the APIs of testing functionality are all encapsulated in the class *CamTest_Testing*. To measure the processing speed of the video processing module, testers need to record the start and end time of a recognition action by using the two functions on the video processing thread: *recordStartTime()* and *recordEndTime()* (line 3, 8 and 9). To get the timestamp of the video frame corresponding to a recognition action, testers can use the public function *getFrameTime()* in the replay thread. (line 6). *recordEndTime()* involves the access to storage. Since recoding data into the storage requires the information of the current application environment, testers should call *getApplicationContext()* to get the application state that is represented by an instance of *Context* (line 7).

After a recognition action on a frame by the video processing module, testers may record the recognition result using the function *recordPrediction()* (line 10 and 11) to examine the accuracy of the video processing algorithm. The result can be numerical or a text string. However, for the compatibility of data saving, testers must convert the result into a text string. Together with the result, a numerical value of the confidence can be saved.

To support debugging, there are two kinds of debugging functionality: runtime assertions and exceptions. To launch and record an assertion on a variable, testers can call *recordAssertion()* with the expected and actual value as parameters (line 12 and 13). If the assertion fails (i.e., the two values are different), the function will record it together with the related information and the frame timestamp into the storage. To record runtime exceptions, testers can call *recordException()* with the exception object as the parameter (line 14 and 15). The description and stack trace (i.e., the trace of method calls with line numbers in source code) of the exception are recorded. The supported exceptions can be standard exceptions directly from the operating system, or developer-defined exceptions (e.g., defining a class inheriting *java.lang.Exception* to handle an application-specific error) from the targeted application itself.

```
1   //In the Activity implementing TextureView.SurfaceTextureListener
2   //On the video processing thread
3   CamTest_Testing.recordStartTime();
4   //Application Specific Video processing API
5   Result rst = video_processing_module.recognise(frame);
6   long eventEndTime = replayThread.getFrameTime();
7   Context ctx = getApplicationContext();
8   CamTest_Testing.recordEndTime(ctx, CamTest_Replay.replayName,
9           CamTest_Replay.appName, eventEndTime);
10  CamTest_Testing.recordPrediction(ctx, CamTest_Replay.replayName,
11          CamTest_Replay.appName, rst.text, rst.confidence,eventEndTime);
12  CamTest_Testing.recordAssertion(ctx, CamTest_Replay.replayName,
13          CamTest_Replay.appName, expected, actual, info, eventEndTime);
14  CamTest_Testing.recordException(ctx, CamTest_Replay.replayName,
15          CamTest_Replay.appName, exception, eventEndTime);
```

Listing 5: Exemplary API calls of testing functionality

As in Listing 6, the design of these API calls of recording visual output adopts the drawing function (*Canvas*) in Android. Similar to the API for recording plain text output, testers need to get the timestamp of the video frame and the state of the application environment (line 5 and 6).

Once the video processing module generates visual output, testers should obtain the attributes of the visual output which can be a number of dots, lines, rectangles and circles. To record a dot, testers can call the function *recordVisualisationDot()* with the attributes of the dot as parameters, including coordinates, weight (i.e., thickness), colour in RGB format

(Android applications have to use RGB colour to display visual effects on UI), duration of appearance and the timestamp of the video (line 8, 9 and 10). Similarly, to record a line or rectangle, testers can call the function *recordVisualisationLine()* (line 12, 13 and 14) or *recordVisualisationRectangle()* (line 16, 17 and 18). The coordinates for a line should be its two nodes. For a rectangle, the coordinates should be the two nodes of one of its diagonals. To record a circle, testers need to call the function *recordVisualisationCircle()* (line 20, 21 and 22) with the centre coordinates and the radius as parameters.

```
1   //In the Activity implementing TextureView.SurfaceTextureListener
2   //On the video processing thread
3   //Application Specific Video processing API
4   Result rst = video_processing_module.recognise(frame);
5   long eventEndTime = replayThread.getFrameTime();
6   Context ctx = getApplicationContext();
7   //If the result rst generates a dot for visual output
8   CamTest_Testing.recordVisualisationDot(ctx, CamTest_Replay.replayName,
9           CamTest_Replay.appName, x, y, weight, R, G, B, duration,
10          eventEndTime);
11  //If the result rst generates a line for visual output
12  CamTest_Testing.recordVisualisationLine(ctx, CamTest_Replay.replayName,
13          CamTest_Replay.appName, x1, y1, x2, y2, weight, R, G, B,
14          duration, eventEndTime);
15  //If the result rst generates a rectangle for visual output
16  CamTest_Testing.recordVisualisationRectangle(ctx,
17          CamTest_Replay.replayName, CamTest_Replay.appName, x1, y1,
18          x2, y2, weight, R, G, B, duration, eventEndTime);
19  //If the result rst generates a circle for visual output
20  CamTest_Testing.recordVisualisationCircle(ctx,
21          CamTest_Replay.replayName, CamTest_Replay.appName, x1, y1,
22          radius, weight, R, G, B, duration, eventEndTime);
```

Listing 6: Exemplary API calls of recording visual output

References

[1] C.-W. You, N.D. Lane, F. Chen, R. Wang, Z. Chen, T.J. Bao, M. Montes-de Oca, Y. Cheng, M. Lin, L. Torresani, et al., Carsafe app: Alerting drowsy and distracted drivers using dual cameras on smartphones, in: Proceeding of the 11th Annual International Conference on Mobile Systems, Applications, and Services, ACM, 2013, pp. 13–26.

[2] Y. Kawano, K. Yanai, Foodcam: A real-time food recognition system on a smartphone, Multimedia Tools Appl. 74 (14) (2015) 5263–5287.

[3] Y. Shen, W. Hu, M. Yang, B. Wei, S. Lucey, C.T. Chou, Face recognition on smartphones via optimised sparse representation classification, in: Information Processing in Sensor Networks, IPSN-14 Proceedings of the 13th International Symposium on, IEEE, 2014, pp. 237–248.

[4] H. Lu, W. Chan, T. Tse, Testing context-aware middleware-centric programs: a data flow approach and an rfid-based experimentation, in: Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering, ACM, 2006, pp. 242–252.

[5] C. Luo, M. Kuutila, S. Klakegg, D. Ferreira, H. Flores, J. Goncalves, M. Mäntylä, V. Kostakos, Testaware: A laboratory-oriented infrastructure for mobile context-aware applications, Proc. ACM Interact. Mob. Wearable Ubiquit. Technol. 1 (3) (2017) 80.

[6] C. Luo, M. Kuutila, S. Klakegg, D. Ferreira, H. Flores, J. Goncalves, V. Kostakos, M. Mäntylä, How to validate mobile crowdsourcing design? leveraging data integration in prototype testing, in: Proceedings of the 2016 ACM International Joint Conference on Pervasive and Ubiquitous Computing: Adjunct, ACM, 2016, pp. 1448–1453.

[7] L.M. Bergasa, D. Almería, J. Almazán, J.J. Yebes, R. Arroyo, Drivesafe: An app for alerting inattentive drivers and scoring driving behaviors, in: Intelligent Vehicles Symposium Proceedings, 2014 IEEE, IEEE, 2014, pp. 240–245.

[8] S. Morishita, S. Maenaka, D. Nagata, M. Tamai, K. Yasumoto, T. Fukukura, K. Sato, Sakurasensor: quasi-realtime cherry-lined roads detection through participatory video sensing by cars, in: Proceedings of the 2015 ACM International Joint Conference on Pervasive and Ubiquitous Computing, ACM, 2015, pp. 695–705.

[9] F. Grijalva, J. Rodriguez, J. Larco, L. Orozco, Smartphone recognition of the us banknotes' denomination, for visually impaired people, in: ANDESCON, 2010 IEEE, IEEE, 2010, pp. 1–6.

[10] K.B. Raja, R. Raghavendra, V.K. Vemuri, C. Busch, Smartphone based visible iris recognition using deep sparse filtering, Pattern Recognit. Lett. 57 (2015) 33–42.

[11] R. Raghavendra, C. Busch, B. Yang, Scaling-robust fingerprint verification with smartphone camera in real-life scenarios, in: Biometrics: Theory, Applications and Systems (BTAS), 2013 IEEE Sixth International Conference on, IEEE, 2013, pp. 1–8.

[12] D. Amalfitano, A.R. Fasolino, P. Tramontana, N. Amatucci, Considering context events in event-based testing of mobile applications, in: Software Testing, Verification and Validation Workshops (ICSTW), 2013 IEEE Sixth International Conference on, IEEE, 2013, pp. 126–133.

[13] R. Tonjes, E.S. Reetz, M. Fischer, D. Kuemper, Automated testing of context-aware applications, in: Vehicular Technology Conference (VTC Fall), 2015 IEEE 82nd, IEEE, 2015, pp. 1–5.

[14] T. Griebe, V. Gruhn, A model-based approach to test automation for context-aware mobile applications, in: Proceedings of the 29th Annual ACM Symposium on Applied Computing, ACM, 2014, pp. 420–427.

[15] S. Bobek, ContextSimulator, http://glados.kis.agh.edu.pl/doku.php?id=pub:software:contextsimulator:start 2017.

[16] D. Ferreira, V. Kostakos, A.K. Dey, Aware: mobile context instrumentation framework, Front. ICT 2 (2015) 6.

[17] L. Gomez, I. Neamtiu, T. Azim, T. Millstein, Reran: Timing-and touch-sensitive record and replay for android, in: Software Engineering (ICSE), 2013 35th International Conference on, IEEE, 2013, pp. 72–81.

[18] Z. Qin, Y. Tang, E. Novak, Q. Li, Mobiplay: A remote execution based record-and-replay tool for mobile applications, in: Proceedings of the 38th International Conference on Software Engineering, ACM, 2016, pp. 571–582.

[19] D. Chu, N.D. Lane, T.T.-T. Lai, C. Pang, X. Meng, Q. Guo, F. Li, F. Zhao, Balancing energy, latency and accuracy for mobile sensor data classification, in: Proceedings of the 9th ACM Conference on Embedded Networked Sensor Systems, ACM, 2011, pp. 54–67.

[20] Q. Luo, D. Poshyvanyk, A. Nair, M. Grechanik, Forepost: A tool for detecting performance problems with feedback-driven learning software testing, in: Software Engineering Companion (ICSE-C), IEEE/ACM International Conference on, IEEE, 2016, pp. 593–596.

[21] C. Min, S. Lee, C. Lee, Y. Lee, S. Kang, S. Choi, W. Kim, J. Song, Pada: power-aware development assistant for mobile sensing applications, in: Proceedings of the 2016 ACM International Joint Conference on Pervasive and Ubiquitous Computing, ACM, 2016, pp. 946–957.

[22] L. Ravindranath, S. Nath, J. Padhye, H. Balakrishnan, Automatic and scalable fault detection for mobile applications, in: Proceedings of the 12th annual international conference on Mobile systems, applications, and services, ACM, 2014, pp. 190–203.

[23] C. Luo, A. Fylakis, J. Partala, S. Klakegg, J. Goncalves, K. Liang, T. Seppänen, V. Kostakos, A data hiding approach for sensitive smartphone data, in: Proceedings of the 2016 ACM International Joint Conference on Pervasive and Ubiquitous Computing, ACM, 2016, pp. 557–568.
[24] K. Haller, Mobile testing, ACM SIGSOFT Softw. Eng. Notes 38 (6) (2013) 1–8.
[25] D. Amalfitano, A.R. Fasolino, P. Tramontana, B. Robbins, Testing android mobile applications: Challenges, strategies, and approaches, in: Advances in Computers. Vol. 89, Elsevier, 2013, pp. 1–52.
[26] Google, Android Monkey, http://developer.android.com/tools/help/monkey.html 2018.
[27] H. Kuehne, H. Jhuang, E. Garrote, T. Poggio, T. Serre, Hmdb: a large video database for human motion recognition, in: Computer Vision (ICCV), 2011 IEEE International Conference on, IEEE, 2011, pp. 2556–2563.
[28] A. Izvorski, scikit-video, https://github.com/aizvorski/scikit-video 2018.
[29] OpenCV, OpenCV, https://opencv.org/ 2018.
[30] cpuboss, Compare CPUs, http://cpuboss.com/compare-cpus 2018.
[31] K. Roberts-Hoffman, P. Hegde, ARM cortex-a8 vs. intel atom: Architectural and benchmark comparisons, Dallas: University of Texas at Dallas, 2009.
[32] Google, Using Prebuilt Libraries, https://developer.android.com/ndk/guides/prebuilts.html 2018.
[33] Google, Building Mobile Apps with TensorFlow, https://www.tensorflow.org/mobile/ 2017.
[34] Android, MediaCodec, https://developer.android.com/reference/android/media/MediaCodec 2019.
[35] Apple, VideoToolbox, https://developer.apple.com/documentation/videotoolbox 2019.
[36] Microsoft, Universal Windows Platform: Audio, video, and camera, https://docs.microsoft.com/en-us/windows/uwp/audio-video-camera/ 2019.
[37] X. Fu, X. Wu, M. Song, M. Chen, Research on audio/video codec based on android, in: Wireless Communications Networking and Mobile Computing (WiCOM), 2010 6th International Conference on, IEEE, 2010, pp. 1–4.
[38] FFmpeg, FFmpeg, http://ffmpeg.org/ 2017.
[39] Google, Hardware Acceleration, https://developer.android.com/guide/topics/graphics/hardware-accel.html 2018.
[40] Google, Android View, https://developer.android.com/reference/android/view/package-summary.html 2018.
[41] PTC, Vuforia, https://www.vuforia.com/ 2019.
[42] SDL, Simple DirectMedia Layer, https://www.libsdl.org/index.php 2018.
[43] Google, Overview of Android Memory Management, https://developer.android.com/topic/performance/memory-overview.html 2018.
[44] R.K. Balan, D. Gergle, M. Satyanarayanan, J. Herbsleb, Simplifying cyber foraging for mobile devices.
[45] M.K. Patterson, The effect of data center temperature on energy efficiency, in: Thermal and Thermomechanical Phenomena in Electronic Systems, 2008. ITHERM 2008. 11th Intersociety Conference on, IEEE, 2008, pp. 1167–1174.