V2S: A Tool for Translating Video Recordings of Mobile App Usages into Replayable Scenarios

Madeleine Havranek*, Carlos Bernal-Cárdenas*, Nathan Cooper*,
Oscar Chaparro*, Denys Poshyvnayk*, Kevin Moran†
*College of William & Mary (Williamsburg, VA, USA), †George Mason University (Fairfax, VA, USA)
mrhavranek@email.wm.edu, cebernal@cs.wm.edu, nacooper01@email.wm.edu,
oscarch@wm.edu, denys@cs.wm.edu, kpmoran@gmu.edu

***Abstract*—Screen recordings are becoming increasingly important as rich software artifacts that inform mobile application development processes. However, the amount of manual effort required to extract information from these graphical artifacts can hinder resource-constrained mobile developers. This paper presents Video2Scenario (V2S), an automated tool that processes video recordings of Android app usages, utilizes neural object detection and image classification techniques to classify the depicted user actions, and translates these actions into a replayable scenario. We conducted a comprehensive evaluation to demonstrate V2S's ability to reproduce recorded scenarios across a range of devices and a diverse set of usage cases and applications. The results indicate that, based on its performance with 175 videos depicting 3,534 GUI-based actions, V2S is able to reproduce ≈ 89% of sequential actions from collected videos. Demo URL: `https://tinyurl.com/v2s-demo-video`**

I. Introduction

Rich software artifacts such as crash reports, bug reports, and user reviews provide invaluable information to mobile application developers throughout the development cycle. Recently, it has become increasingly common that screenshots and screen recordings comprise or are used among such artifacts. These graphical artifacts are both easy to obtain and, based on mobile app's heavy reliance on GUI elements to enable functionality, are often better suited to communicate the complex concepts included in a feature request or bug report than a textual description. Unfortunately, while the prevalence of these visual mobile development artifacts continues to rise, the manual effort required to glean relevant information from them remains a challenge. This indicates the need for automated techniques that can extract and analyze data from screen recordings and screenshots.

In this paper, we present V2S, the first Android replay tool that serves to automate the analysis of video-based mobile development artifacts. V2S functions *solely* on videos; it processes screen recordings of Android app usages, detects and classifies the depicted actions using recently-developed Deep Learning (DL) techniques, and translates these actions into a replayable scenario for a given target device.

We comprehensively analyzed the ability of V2S to reproduce depicted actions and found that it was capable of correctly reproducing ≈89% of the sequential events across 175 collected videos, illustrating its accuracy. We also conducted a case study to evaluate V2S's perceived utility in supporting developers during mobile app development. We found that, across the board, V2S was perceived as a useful tool for app testing and debugging. V2S is an open-source tool that can be found online at `https://tinyurl.com/v2s-tool`.

II. V2S Design

Fig. 1 provides an overview of V2S and its three phases, each of which play a crucial role in accomplishing this functionality. Given that V2S was designed and built with extension in mind, each of these phases and each of the component elements of these phases can be extended or substituted to further the functionality of the pipeline as a whole (*e.g.,* users could substitute our object detection model for a custom model). The intent of this design choice is to allow researchers and developers to customize V2S for future projects or development use cases.

V2S receives a user-specified configuration file as input that includes the path to the video file to be processed, the path to the object detection and opacity models, and information about the target device. Using the video file's path, V2S enters Phase 1 of the pipeline, the *video manipulation* and *touch detection* phase, where V2S first extracts each individual video frame and then detects the location and opacity of the touches exhibited in these frames. V2S then enters Phase 2, the *action classification* phase, where these detected touches are classified as **`Tap`**, **`Long Tap`**, or **`Gesture`**. Finally, V2S enters Phase 3, the *scenario generation* phase, in which a replay script is produced and the input scenario is emulated on the target device. V2S, in its current implementation, can process one video at a time. Details about V2S's algorithms and evaluation can be found in its original research paper [1].

A. Input Video Specifications

Input videos can be captured on a range of devices in one of several different ways, including using the built-in Android **`screenrecord`** utility or any number of other recording applications available on Google Play [2]. To ensure compatibility with V2S, input videos must adhere to a few specific constraints. Firstly, the frame size of the input video must align with one of our predefined models and equal the screen size of the target Android device. Currently, V2S

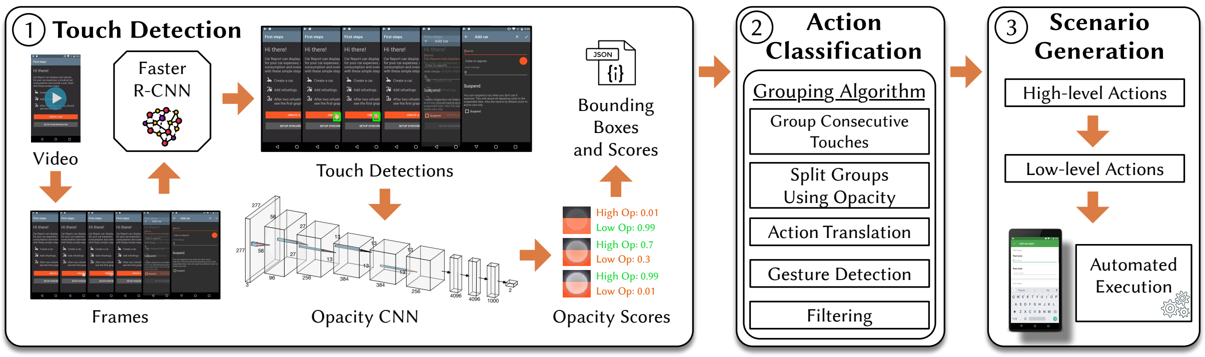
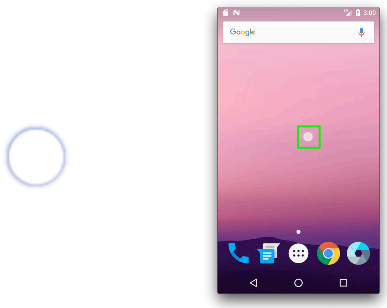


Fig. 1: V2S Design

supports the Nexus 5 and Nexus 6P screen sizes[1]. Secondly, V2S requires each input video to be recorded with at least 30 "frames per second" (FPS) so that fast-paced gestures (such as flicks or rapid swipes) can be accurately detected and replayed. Finally, because V2S aims to detect the location of a user's finger on the screen, input videos must be recorded with the "Show Touches" option enabled on the device in developer mode. This enables an opaque, circular *touch indicator* to appear as the user presses and/or moves their finger on the screen (see Fig. 2.a); as the user lifts their finger to finish an action, the opacity of the indicator decreases.



(a) Touch Indicator (b) Touch Indicator on Device

Fig. 2: Default Touch Indicator

B. Phase 1: Video Manipulation and Touch Detection

Phase 1 first preprocesses the input video and then detects the location of individual touches throughout the video using a Neural Object Detection framework. Phase 1 executes three components: (i) the *FrameExtractor*, (ii) the *TouchDetector-FRCNN*, and (iii) the *OpacityDetector*.

1) FrameExtractor (FE): To account for different frame rates in input videos, V2S first manipulates each video to adhere to a standard 30 FPS. Then, V2S extracts individual frames from the normalized video using **`ffmpeg`** [3].

2) TouchDetectorFRCNN (TD): The *TD* component applies a modified FASTER R-CNN model (trained utilizing the Tensorflow Object Detection API [4]) to each frame and accurately predicts the location of the bounding box of the touch indicators if any (see Fig. 2.b) [5].

[1]Support for additional devices and operating systems can be added, given developers have a connection to the device, the touch indicator image, and device specifications (screen size, etc.). See [1] for more information.

3) OpacityDetector (OD): After the touches have been localized by the *TD*, the *OD* crops each touch around its bounding box and then feeds these into the OPACITY CNN, which is an extended version of the ALEXNET architecture [6]. This architecture classifies each touch as high or low opacity.

Phase 1 then pairs the touch locations detected by the *TD* with the opacity values predicted by the *OD* to form a "complete" list of detected touch indicators in each video frame. This list is written to a *detection_full.json* file.

C. Phase 2: Action Classification

Phase 2 groups the touches detected in Phase 1 and translates them into a set of actions. Phase 2 begins by reading each individual detected touch from the *detection_full.json* file and, as a preliminary attempt to ensure that each touch that enters the classification process is a true positive, it removes any taps with a confidence measure below 0.7. V2S passes these resulting taps to the *GUIActionClassifier*.

1) GUIActionClassifier (GAC): The *GAC* component executes three distinct steps to classify the depicted actions accurately: (i) an action grouping step to organize individual touches across consecutive frames into discrete actions, (ii) a segmentation step to determine the start and end points of complex actions, and (iii) an action translation step which allows for these groupings to be associated with a specific action type. The output of this component is a list of detected actions that can be written to a **`json`** file.

Grouping Consecutive Touches – As complete actions may occur across consecutive frames, the *GAC* begins by detecting consecutive frames and grouping the detected taps in them into distinct groups based on the timing of the video frames.

Discrete Action Segmentation – For some fast-paced actions, there may not be separating frames to delineate distinct actions executed by the user. This behavior may be seen when a user is typing on the keyboard or swiping quickly through an article. In these cases, multiple touches will appear in the same frame, and a technique to separate these touches into distinct actions is necessary. The *GAC* component tackles this using a heuristic-based approach, modeling each group of consecutive frames and its associated touches as a connected components graph problem. Each touch within the frame group is considered a node. Nodes are connected if they occur across consecutive frames and are clustered according to pixel-based distance. This means that touches that share similar spatial characteristics across frames are treated as continuous actions. *Action Translation* – Actions are translated into one of three predetermined options based on their average location and duration. **`Taps`** remain in the same relative on-screen location and persist for <20 frames. **`Long Taps`** have similar location requirements but last for >=20 frames. Any actions that do not fit into these two categories become **`Gestures`**.

Phase 2 outputs a *detected_actions.json* file containing the list of detected actions with their touches from Phase 1.

D. Phase 3: Scenario Generation

Phase 3 of V2S harnesses the actions classified in Phase 2 and translates them into a script in the **`sendevent`** format available in Android's Linux kernel [7]. V2S then utilizes a modified version of the RERAN [8] binary to replay the predicted scenario on the target device. Phase 3 reads in the actions specified in the *detected_actions.json* file output by Phase 2 and begins the execution of the *Action2EventConverter*.

1) Action2EventConverter (A2EC): The *A2EC* converts the high-level actions produced by Phase 3 into low-level commands in the **`sendevent`** format. This component utilizes the **`start_event`** and **`end_event`** commands, and the x and y coordinates of the various actions to generate the replay script and control the UI behavior on the target device.

Each action begins with the **`start_event`** command to indicate the start of an event. For the **`Tap`** action, the *A2EC* component calculates the centroid or average location. For the **`Long Tap`** action, the *A2EC* component utilizes this same centroid point but sustains this command for a specified duration of time, depending on the number of frames for which the action persists. Finally, for **`Gesture`** actions, the *A2EC* component iterates over and appends each (x, y) coordinate pair of the action to the script. For each of these actions, the *A2EC* terminates the action by appending the **`end_event`** command to the script.

The *A2EC* allots a duration of 33 milliseconds per frame and calculates an appropriate timestamp for each command appended to the script. As 33 milliseconds is the delay between frames for a video at 30 FPS, this ensures proper timing of actions to be executed in the generated script.

2) Scenario Replay: The script generated by the *A2EC* is written to a *send_events.log* file. This log file is then converted by the *Translator* component into a format that is executable on the target device. Once this translation has occurred, V2S pushes the executable file and a modified version of the RERAN binary to the device, starts a screen recording of the generated scenario, and executes the V2S script.

III. V2S Evaluation

In order to evaluate the capabilities of V2S as a whole, we asked five different research questions and conducted associated studies. We measured: (i) the accuracy of the touch detection Faster R-CNN model; (ii) the accuracy of the opacity detection AlexNet model; (iii) the accuracy of V2S on different usage scenarios; (iv) the runtime performance of V2S; and (v) the practical utility of V2S. The results here are summarized from the original V2S paper [1].

A. Accuracy of Faster R-CNN

1) Design: In order to evaluate the Faster R-CNN model's ability to correctly identify the location of a touch indicator within an image, we generated a dataset of 15,000 images containing touch indicators and split this data 70%-30% into training and testing sets, respectively. For this preliminary study, we trained two different models for two different screen sizes (the Nexus 5 and the Nexus 6P) using the Tensorflow Object Detection API [4]. More information about this training process can be found in the V2S paper [1].

For this study, we considered the model's *Mean Average Precision* (mAP), which is a commonly-used metric to determine accuracy for object detection tasks [1]. This metric is calculated as $mAP = TP/(TP + FP)$, where a TP is a region correctly identified as a touch indicator and an FP is a region that is incorrectly marked as containing a touch indicator. We also considered the recall metric of this model to determine the frequency with which our model misses a touch indicator when one is present.

2) Results: All of the models achieved $\approx 97\%$ mAP in predicting the location of the touch indicator. The model was also able to achieve $\approx 99\%$ recall, meaning that the model rarely missed the detection of the touch indicator.

B. Accuracy of Opacity CNN

1) Design: We evaluated the Opacity CNN model's accuracy in correctly classifying whether a touch indicator has a high or low opacity. We generated a dataset of 10,000 images containing equal numbers of high-opacity and low-opacity touch indicators and then split these images 70%-30% into training and testing datasets, respectively. We used TensorFlow and Keras to create and train a modified Opacity CNN.

For this study, we considered the *Mean Average Precision* (mAP) of this model, where a TP is an indicator that was identified as having the correct opacity, and an FP is an indicator that was predicted to have the incorrect opacity value.

2) Results: We found that the precision of this model was $\approx$ 98-99% mAP. This indicates that the opacity model is very accurate in its ability to correctly distinguish between high and low-opacity touch indicators.

C. Accuracy on Usage Scenarios

1) Design: We designed two studies to test V2S's ability to replicate an original usage scenario depicted in a video recording. The *Controlled Study* (*CS*) was meant to ensure the *depth* of V2S's abilities in reproducing a variety of bug crashes, synthetically-injected and real application failures, and normal usage scenarios. Then, in the *Popular Applications Study* (*PAS*), we assessed the *breadth* of V2S in its ability to accurately reproduce a variety of usage scenarios depicted in a diverse set of applications from the Google Play store.

In the *CS*, eight participants were recruited from William & Mary to record eight different scenarios each. These scenarios came from two of the following four categories: (i) normal usage cases, (ii) bugs in open source apps, (iii) real crashes, and (iv) synthetically-injected crashes in open source apps. Four participants recorded on the Nexus 5 and four recorded on the Nexus 6P. The participants familiarized themselves with the scenario before recording. Each of the buggy scenarios was extracted from previous studies of a similar nature [9]–[11].

For the *PAS*, we downloaded the top-two rated applications from 32 categories in the Google Play store. Two of the authors then recorded two scenarios per application, with each scenario differently exhibiting one of the major features of the application. To ensure that V2S could reproduce videos on different devices, one author recorded their scenarios on the Nexus 5 and the other on the Nexus 6P.

To verify V2S' accuracy, we manually determined the ground truth action sequences and used this to compute four different metrics: (i) Levenshtein distance, (ii) Longest Common Subsequence (LCS), (iii) precision and recall, and (iv) manual video comparison. Levenshtein distance is a metric that depicts the number of alterations necessary to transform one sequence into another. The LCS metric represents the longest sequence of continuous matching actions produced by V2S when compared to the ground truth sequence. To compute the precision and recall, we created an unordered "action pool" for each scenario in our studies and for each predicted action type. Comparing this predicted "action pool" to the ground truth "action pool" allowed us to calculate these metrics overall and per individual action type. Finally, each scenario was manually reviewed and marked as successful as long as the reproduced behavior exercised the same overall functionality as the original.

2) Results: We briefly summarize the results.

a) Levenshtein Distance: For the *CS* and *PAS*, the avg. distance value was 0.85 and 1.17 changes, respectively.

b) LCS: In the *CS*, V2S was able to match 95.1% of the sequential actions, and for the *PAS*, V2S was able to correctly emulate 90.2% of these consecutive events.

c) Precision and Recall: Overall, V2S had very high precision and recall values for all event types: for the *CS*, V2S achieved 95.3% precision and 99.3% recall, and for the *PAS*, V2S achieved 95% precision and 97.8% recall.

d) Manual Video Comparison: V2S accurately replicated 93.75% of the 64 scenarios in the *CS* and 94.48% of sequential actions. For the *PAS*, of the 111 videos fed into V2S, the tool accurately replayed 81.98% of scenarios and 89% of sequential actions.

D. V2S *Runtime Performance*

1) Design: To assess the execution performance of V2S, we calculated the average runtime of the pipeline per video.

2) Results: We measured the performance per frame (in seconds per frame or s/f) of each major step that makes up V2S. We determined the average runtime of (i) the frame extraction process (0.045 s/f); (ii) the touch detection process (1.09 s/f); and (iii) the opacity classification step (0.032 s/f). By our calculations, an average 3-minute video processed by V2S in full would take $\approx$ 105 minutes. This performance could be easily enhanced by parallelizing the computation.

E. *Industrial Utility*

1) Design: We wanted to understand if mobile developers perceive V2S as useful. We interviewed three software engineers to assess the potential role of V2S in their day-to-day operations. The first section of the interview aimed at gaining an understanding of the developer's backgrounds and the tools that they use to accomplish their every-day tasks. The second section of the interview was intended to allow participants to assess the performance and utility of V2S. This was accomplished by presenting them with a demonstration of the input and replication videos and the resulting detections, action list, and replay script that V2S produces.

2) Results: All of the participants agreed that V2S is highly accurate at reproducing actions depicted in an input video. Each of the participants also viewed V2S as a potentially useful tool for app testing and debugging.

IV. Final Remarks & Future Work

In this tool demonstration paper, we have presented V2S, a novel tool for translating video recordings of mobile app usages into replayable scenarios. V2S facilitates different testing and debugging activities by allowing for easy replay of app usages and bugs captured via screen recordings, and our evaluation illustrates its effectiveness across several dimensions. As future work, we plan to add support for multi-fingered actions and train additional object detection models so that V2S can be used with a more diverse set of devices.

V. Acknowledgements.

This research was supported in part by the NSF CCF-1955853 and CCF-1815186 grants. Any opinions, findings, and conclusions expressed herein are the authors' and do not necessarily reflect those of the sponsors.

References

[1] C. Bernal-Cárdenas, N. Cooper, K. Moran, O. Chaparro, A. Marcus, and D. Poshyvanyk, "Translating video recordings of mobile app usages into replayable scenarios," in *ICSE'20*, p. 309–321, 2020.
[2] "Google play screen recording apps https://tinyurl.com/s-recgpl," 2019.
[3] "Ffmpeg tool https://www.ffmpeg.org/," 2019.
[4] "Tensorflow object detection api https://tinyurl.com/tf-ojbdet," 2019.
[5] S. Ren, K. He, R. Girshick, and J. Sun, "Faster r-cnn: towards real-time object detection with region proposal networks," *IEEE Transactions on Pattern Analysis & Machine Intelligence*, no. 6, pp. 1137–1149, 2017.
[6] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," NeurIPS'12, pp. 1097–1105.
[7] "Getevent, https://source.android.com/devices/input/getevent/," 2020.
[8] L. Gomez, I. Neamtiu, T. Azim, and T. Millstein, "RERAN: Timing-and touch-sensitive record and replay for android," ICSE'13, pp. 72–81.
[9] O. Chaparro, C. Bernal-Cárdenas, J. Lu, K. Moran, A. Marus, M. Di Penta, D. Poshyvanyk, and V. Ng, "Assessing the quality of the steps to reproduce in bug reports," ESEC/FSE'19, pp. 86–96, 2019.
[10] K. Moran, M. Linares-Vásquez, C. Bernal-Cárdenas, and D. Poshyvanyk, "Auto-completing bug reports for android applications," ESEC/FSE'15, pp. 673–686, 2015.
[11] K. Moran, M. Linares-Vásquez, C. Bernal-Cárdenas, C. Vendome, and D. Poshyvanyk, "Automatically discovering, reporting and reproducing android application crashes," ICST'16, pp. 33–44, 2016.