Translating Video Recordings of Complex Mobile App UI Gestures into Replayable Scenarios

Carlos Bernal-Cárdenas, *Member, IEEE,* Nathan Cooper, *Student Member, IEEE,* Madeleine Havranek, *Student Member, IEEE,* Kevin Moran, *Member, IEEE,* Oscar Chaparro, *Member, IEEE,* Denys Poshyvanyk, *Member, IEEE,* and Andrian Marcus, *Member, IEEE*

Abstract—Screen recordings of mobile applications are easy to obtain and capture a wealth of information pertinent to software developers (*e.g.*, bugs or feature requests), making them a popular mechanism for crowdsourced app feedback. Thus, these videos are becoming a common artifact that developers must manage. In light of unique mobile development constraints, including swift release cycles and rapidly evolving platforms, automated techniques for analyzing all types of rich software artifacts provide benefit to mobile developers. Unfortunately, automatically analyzing screen recordings presents serious challenges, due to their graphical nature, compared to other types of (textual) artifacts. To address these challenges, this paper introduces V2S+, an automated approach for translating video recordings of Android app usages into replayable scenarios. V2S+ is based primarily on computer vision techniques and adapts recent solutions for object detection and image classification to detect and classify user *gestures* captured in a video, and convert these into a replayable test scenario. Given that V2S+ takes a computer vision-based approach, it is applicable to both hybrid and native Android applications. We performed an extensive evaluation of V2S+ involving 243 videos depicting 4,028 GUI-based actions collected from users exercising features and reproducing bugs from a collection of over 90 popular native and hybrid Android apps. Our results illustrate that V2S+ can accurately replay scenarios from screen recordings, and is capable of reproducing $\approx$ 90.2% of sequential actions recorded in native application scenarios on physical devices, and $\approx$ 83% of sequential actions recorded in hybrid application scenarios on emulators, both with low overhead. A case study with three industrial partners illustrates the potential usefulness of V2S+ from the viewpoint of developers.

Index Terms—Bug Reporting, Screen Recordings, Object Detection.

1 Introduction

Mobile application developers rely on a diverse set of software artifacts to help them make informed decisions throughout the development process. These information sources include user reviews, crash reports, bug reports, and emails, among others. An increasingly common component of these software artifacts is graphical information, such as screenshots or screen recordings [1]. This is primarily due to the fact that they are relatively easy to collect and, due to the GUI-driven nature of mobile apps, they contain rich information that can easily demonstrate complex concepts, such as a bug or a feature request. In fact, many crowd-testing and bug reporting frameworks have built-in screen recording features to help developers collect mobile application usage data and faults [2], [3], [4], [5]. *Screen recordings* that depict application usages are used by developers to: (i) help understand how users interact with apps [6], [7]; (ii) process bug reports and feature requests from end-users [8]; and (iii) aid in bug comprehension for testing related tasks [9]. However, despite the growing prevalence of visual mobile development artifacts, developers must still manually inspect and interpret screenshots and videos in order to glean relevant information, which can be time consuming and ambiguous. The manual effort required by this comprehension process complicates a development workflow that is already constrained by language dichotomies [10] and several challenges unique to mobile software, including: (i) pressure for frequent releases [11], [12], (ii) rapidly evolving platforms and APIs [13], [14], (iii) constant noisy feedback from users [15], [16], [17], [18], [19], and (iv) fragmentation in the mobile device ecosystem [20], [21], [22] among others [23]. Automation for processing graphical software artifacts is necessary and would help developers shift their focus toward core development tasks.

To improve and automate the analysis of video-related mobile development artifacts, we introduce *Video to Scenario+* (V2S+), an easy-to-use and modular automated approach for translating video screen recordings of Android app usages into replayable scenarios. We designed V2S+ to operate solely on a video file recorded from an Android device, and as such, it relies primarily on computer vision techniques. It is based on our previously-published work [24]. V2S+ adapts recent Deep Learning (DL) models for object detection and image classification to accurately detect and classify different types of user actions performed

- *C. Bernal-Cárdenas is with the Liquid Team, Microsoft, Redmond, WA, 98052.*
  E-mail: carlosbe@microsoft.com
- *N. Cooper, M. Havranek, O. Chaparro, and D. Poshyvanyk are with the Department of Computer Science, College of William & Mary, Williamsburg, VA, 23185.*
  E-mail: [nacooper01, mrhavranek]@email.wm.edu, oscarch@wm.edu, denys@cs.wm.edu
- *K. Moran is with the Department of Computer Science, George Mason University, Fairfax, VA, 22030.*
  E-mail: kpmoran@gmu.edu
- *A. Marcus is with the Department of Computer Science, The University of Texas at Dallas, Richardson, TX, 75080.*
  E-mail: amarcus@utdallas.edu

Manuscript received Oct 2021;

on the screen. These classified actions are then translated into replayable scenarios that can automatically reproduce user interactions (from the video) on a target device, making V2S+ the first purely graphical Android record-and-replay technique.

In addition to helping automatically process the graphical data that is already present in mobile development artifacts, V2S+ can also be used for improving or enhancing additional development tasks that do not currently take full advantage of screen-recordings, such as creating and maintaining automated GUI-based test suites, and crowdsourcing functional and usability testing.

We conducted a comprehensive evaluation of V2S+ to assess its *accuracy*, *robustness*, *efficiency*, and *usefulness* in generating replayable scenarios of videos collected from users interacting with both native and hybrid applications. Users interacted with these applications using a variety of gestures, including both single- and multi-fingered gestures. We evaluated V2S+ on a set of popular native Android applications on physical devices. Users reproduced bugs and exercised multiple features on the top-rated apps of 32 categories in the Google Play market using single-fingered gestures. We also evaluated V2S+ using videos collected from top-rated apps in 2 categories where users executed multi-fingered actions (*e.g.,* two-fingered rotations, two-fingered pinches to zoom, *etc.*). We chose to conduct separate studies to evaluate V2S+'S performance on native and hybrid applications in order to assess its generalizability for these applications and highlight V2S+'S support on hybrid applications, given the relative lack of GUI testing tools available for these apps. Hybrid applications are built using a combination of web standards (*e.g.,* HTML and CSS) and native SDKs and target multiple platforms (*e.g.,* both iOS and Android), whereas native applications are built using only native SDKs and target a single platform [25]. Unlike other testing tools (*e.g.,* CRAFTDROID [26] and MONKEY [27]), V2S+ does not require any GUI metadata and instead leverages the detected spatial location of each touch event on the screen, which can be useful for app testing regardless of the application framework (web and/or native). For these hybrid applications, we analyzed V2S+'S performance with user scenarios recorded in 14 popular applications from 7 categories on emulated devices. We evaluated V2S+ on video recordings collected for all of these scenarios. We also assessed the overhead of our technique and conducted a case study with three industrial partners to understand the practical applicability of V2S+.

The results of our evaluation indicate that V2S+ is *accurate*, and is able to correctly reproduce $\approx$ 83% - 90% of events across collected videos. The approach is also *robust* in that it is applicable to a wide range of popular native and hybrid apps currently available on Google Play. In terms of *efficiency*, we found that V2S+ imposes acceptable overhead, and is perceived as *potentially useful* by developers.

In summary, the main contributions of our work are:

- V2S+, the first record-and-replay approach for Android that functions purely on screen-recordings of app usages. V2S+ adapts computer vision solutions for object detection and image classification, to effectively recognize and classify single- and multi-fingered user actions in the video frames of a screen recording;
- An automated pipeline for dataset generation and model training to identify user interactions from screen recordings. This pipeline is publicly available in our online appendix [28] and it is fully automated and documented for others to use and adapt.
- The results of an extensive empirical evaluation of V2S+ that measures the *accuracy*, *robustness*, and *efficiency* across 243 videos from over 90 applications;
- The results of a case study with three industrial partners who develop commercial apps, highlighting V2S+'S potential usefulness, as well as areas for improvement and extension;
- An online appendix [28] , which contains examples of videos replayed by V2S+, experimental data, source code, trained models, and our evaluation infrastructure to facilitate reproducibility of the approach and the evaluation results.

This paper is a substantial extension of our prior research published at the 42nd International Conference on Software Engineering (ICSE'20) [24]. The extension includes the following new contributions: (i) a new version of V2S, called V2S+, which is fully implemented in Python and is capable of recognizing and replaying single- and multi-fingered gestures, (ii) an expanded video data set that includes a new set of hybrid applications, as well as new mobile app usages which include multi-finger gestures; and (iii) two new research questions that focus on assessing the ability of V2S+ to replay videos recorded using both single- and multi-finger gestures on native and/or hybrid applications.

2 BACKGROUND

We briefly discuss DL techniques for image classification and object detection that we adapt for touch/gesture recognition in V2S+.

2.1 Image Classification

Recently, DL techniques that make use of neural networks consisting of specialized layers have shown great promise in classifying diverse sets of images into specified categories. Advanced approaches leveraging Convolutional Neural Networks (CNNs) for highly precise image recognition [29], [30], [31], [32], [33] have reached human levels of accuracy for image classification tasks.

Typically, each *layer* in a CNN performs some form of computational transformation to the data fed into the model. The initial layer usually receives an input image. This layer is typically followed by a convolutional layer that extracts features from the pixels of the image, by applying *filters* (*a.k.a.* kernels) of a predefined size, wherein the contents of the filter are transformed via a pair-wise matrix multiplication (*i.e.,* the convolution operation). Each filter is passed throughout the entire image using a fixed *stride* as sliding window to extract *feature maps*. Convolutional layers are used in conjunction with *max pooling* layers to further reduce the dimensionality of the data passing through the network. The convolution operation is linear in nature. Since images are generally non-linear data sources, activation functions such as Rectified Linear Units (ReLUs)

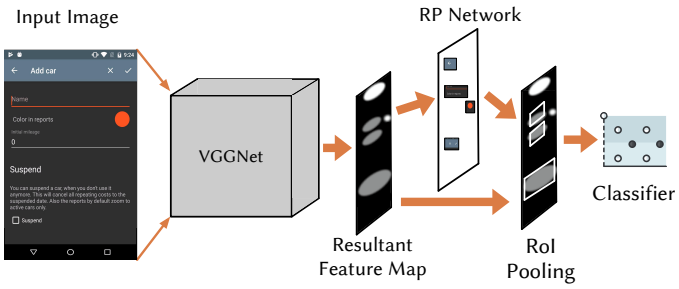


Fig. 1: Illustration of the FASTER R-CNN Architecture

are typically used to introduce a degree of non-linearity. Finally, a fully-connected layer (or series of these layers) are used in conjunction with a *Softmax classifier* to predict an image class. The training process for CNNs is usually done by updating the weights that connect the layers of the network using gradient descent and back-propagating error gradients.

Recently, new models that either improve upon CNN models or use other architectures have shown promise for image classification. EfficientNet [34] used a separate neural network to learn to build an efficient and performant CNN architecture. The Transformer [35] architecture, which traditionally had been only used on natural language processing (NLP) tasks, has also been applied to image classification by using a novel tokenization scheme that allowed for an image to be converted into a sequence of visual tokens instead of subword tokens [36]. Additionally, MultiLayer Perceptrons (MLPs) have also shown potential promise for image classification on par with state of the art models [37].

V2S+ implements a customized CNN for the specific task of classifying the opacity of an image segment to help identify GUI interactions represented by a touch indicator (see Section 3.2).

2.2 Object Detection

In the task of image classification, a single, usually more general label (*e.g.*, *bird* or *person*) is assigned to an entire image. However, images are typically multi-compositional, containing different objects to be identified.

Similar to image classification, DL models for object detection have advanced dramatically in recent years, enabling object tracking and counting, as well as face and pose detection among other applications. One of the most influential neural architectures that has enabled such advancements is the Region-based CNN (R-CNN) introduced by Girshick *et al.* [38]. The R-CNN architecture combines algorithms for image region proposals (RPs), which aim to identify image regions where content of interest is likely to reside, with the classification prowess of a CNN. An R-CNN generates a set of RP bounding-boxes using a selective search algorithm [39]. Then, all identified image regions are fed through a pre-trained ALEXNET [29] (*i.e.*, the *extractor*) to extract image features into vectors. These vectors are fed into a support vector machine (*i.e.*, the *classifier*) that determines whether or not each image region contains a class of interest. Finally, a greedy non-maximum suppression algorithm (*i.e.*, the *regressor*) is used to select the non-overlapping regions with the highest likelihood as classified objects. Recently, DETR [40] was proposed as an end-to-end way of performing object detection using a CNN and an encoder-decoder Transformer architecture. DETR does not need many of the hand-craft heuristics that the R-CNN architecture requires making the architecture simpler and the authors found it performs both in accuracy and speed on par with the hyper optimizer FASTER R-CNN [41] architecture.

We leverage the FASTER R-CNN architecture (Fig. 1), which improves upon R-CNN architecture through the introduction of a separate neural network to predict image region proposals. By integrating the training of the region proposal network into the end-to-end training of the network, both the speed and accuracy of the model increase. For V2S+, we adapt the FASTER R-CNN model to detect a touch indicator representing a user action in video frames (see Section 3.2.2).

3 THE V2S APPROACH

This section outlines the V2S+ approach for automatically translating Android screen recordings into replayable scenarios. Fig. 2 depicts V2S+'s execution flow, which is divided into three main phases: (i) the *Touch Detection* phase, which identifies user touches in each frame of an input video; (ii) the *Action Classification* phase that groups and classifies the detected touches into discrete user actions (*i.e.*, `Tap`, `Long Tap`, `Gesture`, and `Multi-Fingered Gestures`), and (iii) the *Scenario Generation* phase that exports and formats these actions into a replayable script.

V2S [24] was originally implemented in Java, Bash, and Python. We have since redesigned the V2S+ structure to be implemented entirely in Python with extension and modularity in mind. Each phase and component of V2S+ can now be easily substituted and/or extended by way of abstract classes which makes the entire pipeline malleable to fit a wide variety of research and development contexts. Before discussing each phase in detail, we discuss some preliminary aspects of our approach, input specifications, and requirements.

3.1 Input Video Specifications

In order for a video to be consumable by V2S+, it must meet a few requirements to ensure proper functioning with our computer vision (CV) models. First, the video frame size must match the full-resolution screen size of the target Android device, in order to be compatible with a specified pre-trained object-detection network. This requirement is met by nearly every modern Android device that has shipped within the last few years. These videos can be recorded either by the built-in Android `screenrecord` utility, or via third-party applications [42]. The second requirement is that input videos must be recorded with at least 30 "frames per second" (FPS), which again, is met or exceeded by a majority of modern Android devices. This requirement is due to the fact that the frame-rate directly corresponds to the accuracy with which "quick" gestures (*e.g.*, fast scrolling) can be physically resolved in constituent video frames. Third, the video needs to start at the initial screen/state of the application once it is opened or at the home screen of the device. This will guarantee that the reproduction will

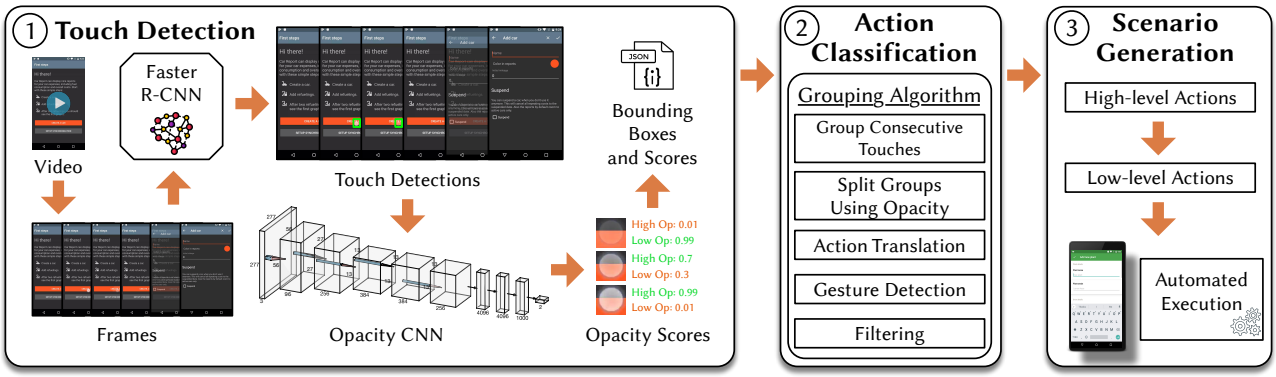


Fig. 2: The V2S+ approach and phases (numbered)

not miss any actions that are required to reproduce the video. Finally, the videos must be recorded with the "Show Touches" option enabled on the device, which is accessed through an advanced settings menu [43], and is available by default on nearly all Android devices since at least Android 4.1. This option renders a *touch indicator*, which is a small semi-transparent circle, that gives a visual feedback when the user presses her finger on the device screen. The opacity of the indicator is fully solid from the moment the user first touches the screen and then fades from more to less opaque when a finger is lifted off the screen (see Fig. 3).

3.2 Phase 1: Touch Detection

The *goal* of this phase is to accurately identify the locations where a user touched the device screen during a video recording. To accomplish this, V2S+ leverages the DL techniques outlined in Sec. 2 to both accurately find the position of the touch indicator appearing in video frames, and identify its opacity to determine whether a user's finger is being pressed or lifted from the screen. The main reason to use DL techniques over traditional image processing techniques (*e.g.*, Canny edge detection [**?**] and color analysis) is that the latter do not perform consistently on video frames with varying levels of contrast (*e.g.*, between the touch indicator and the background), as they require different parameter values, depending on the image. For example, canny edge detection requires specifying two thresholds or parameters (*i.e.*, min and max value for the gradient) that help to identify the edges in an image. However, finding the optimal values for these parameters for all the images across the videos is challenging because there can be different levels of contrast in the images across all the videos, due to the variety of the visual content that each application can show during its execution. At the start of our research project, we experimented with a set of parameter values; however, we quickly realized that finding the optimal parameters would demand a large effort (*i.e.*, a wide range of experiments with different values) without guaranteeing the selected parameters would generalize to other images and apps. By using Deep Learning techniques, we account for the variability in contrast across the images from video screen recordings of mobile apps, controlling for experimental effort and improving generalizability.

More specifically, we adapt an implementation of FASTER R-CNN [41], [44], which makes use of VGGNET [30] for feature extraction of RPs in order to perform touch indicator detection (see Fig. 1). To differentiate between low and high-opacity detected touch indicators, we build an OPACITY CNN, which is a modified version of ALEXNET [29]. Given that we adapt well-known DL architectures, here we focus on describing our adaptions, and provide model specs in our appendix [28].

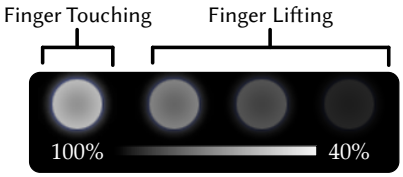


Fig. 3: Illustration of touch indicator opacity levels

The *Touch Detection* Phase begins by accepting as input a video that meets the specifications outlined in Sec. 3.1. First, the video is *parsed* and decomposed into its constituent frames. Then, the FASTER R-CNN network is utilized to *detect* the presence of the touch indicator, if any, in every frame. Finally, the OPACITY CNN *classifies* each detected touch indicator as having either low or high-opacity. The output of this phase is a structured `JSON` with a set of touch indicator bounding boxes in individual video frames wherein each detected touch indicator is classified based on the opacity.

3.2.1 Parsing Videos

Before V2S+ parses the video to extract single frames, it must first normalize the frame-rate for those videos where it may be variable to ensure a constant FPS. Certain Android devices may record variable frame-rate video for efficiency [45]. This may lead to inconsistencies in the time between frames, which V2S+ utilizes in the action classification and scenario replay phases to synthesize the timing of touch actions. Thus, to avoid this issue, we normalize the frame rate to 30fps and extract individual frames using the `FFmpeg` tool [46].

3.2.2 Faster R-CNN

After the individual frames have been parsed from the input video, V2S+ applies its object detection network to localize the bounding boxes of touch indicators. However, before using the object detection, it must be trained. As described in Sec. 2, the DL models that we utilize typically require large, manually labeled datasets to be effective. However,

to avoid the manual curation of data, and make the V2S+ approach practical, we designed a fully automated dataset generation and training process. To bootstrap the generation of V2S+'s object detection training dataset, we make use of the existing large-scale REDRAW dataset of Android screenshots [47]. This dataset includes over 14k screens extracted from the most popular Android applications on Google Play using a fully-automated execution technique.

To create the dataset for training these individual models, we randomly sample 5k unique screenshots of different apps and programmatically superimpose an image of the *touch indicator* at a random location in each screenshot. During this process, we took two steps to ensure that our synthesized dataset reflects actual usage of the touch indicator: (i) we varied the opacity of the indicator icon between 40% and 100% to ensure our model is trained to detect instances where a finger is lifted off the screen; (ii) we placed indicator icons on the edges of the screen to capture instances where the indicator may be occluded. This process is repeated three times per screenshot to generate 15k unique images. We then split this dataset using a 70%-30% random partitioning to create the training and testing sets respectively. We performed this partitioning such that all screenshots except one appear only in the testing set, wherein the one screenshot that overlapped had a different location and opacity value for the touch indicator. During testing, we found that a training set of 15k screens was large enough to train the model to extremely high levels of accuracy (*i.e.*, $> 97\%$).

To train these models we use the *TensorFlow Object Detection API* [48] that provides functions and configurations of well-known DL architectures. We provide details regarding our training process for V2S+'s object detection network in Sec. 5.1. Note that, despite the training procedure being completely automated, it needs to be run only once for a given device screen size, after which it can be re-used for inference. After the model is trained, inference is run on each frame, resulting in a set of output *bounding box* predictions for each screen, with a confidence score.

3.2.3 Opacity CNN

Once V2S+ has localized the screen touches that exist in each video frame, it must then determine the opacity of each detected touch indicator to aid in the *Action Classification* phase. This helps V2S+ in accurately identifying instances where there are multiple actions in consecutive frames with very similar locations or when a swipe ends, which is especially important for fast flicks due to Android needing to calculate the momentum of the swipe. For example, if a user is typing on the keyboard and quickly taps the same key multiple times or double taps a back button, it causes to have multiple touch indicators with similar locations. Thus, identifying when an action finishes and another starts, by using touch indicator's opacity, facilitates action separation and accurate recognition of gestures that involve single and multiple fingers.

To differentiate between low and high-opacity touch indicators, V2S+ adopts a modified version of the ALEXNET [29] architecture as an OPACITY CNN that predicts whether a cropped image of the touch indicator is fully opaque (*i.e.*, finger touching screen) or has low opacity (*i.e.*, indicating a finger being lifted off the screen).

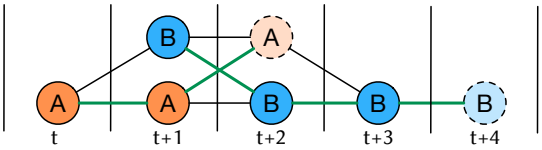


Fig. 4: Illustration of the graph traversal problem for splitting discrete actions. Faded nodes with dotted lines represent touches where a finger is being lifted off the screen.

Similar to the object detection network, we fully automate the generation of the training dataset and training process for practicality. We again make use of the REDRAW dataset and randomly select 10k unique screenshots, randomly crop a region of the screenshot to the size of a touch indicator, and an equal number of full and partial opacity examples are generated. For the low-opacity examples, we varied the transparency levels between 20% and 80% to increase the diversity of samples in the training set. During initial experiments, we found that our OPACITY CNN required fewer training samples than the object detection network to achieve a high accuracy (*i.e.*, $> 97\%$). Similar to the FASTER R-CNN model, this is a one-time training process, however, this model can be re-used across varying screen dimensions. Finally, V2S+ runs the classification for all the detected touch indicators found in the previous step. Then V2S+ generates a `JSON` file containing all the detected bounding boxes, confidence levels, and opacity classifications.

3.3 Phase 2: Action Classification

The `JSON` file generated by the *Touch Detection* phase contains detailed data about the bounding boxes, opacity information, and the video frame corresponding to each detected touch indicator[1]. This `JSON` file is used as input into the *Action Classification* phase where single touches are grouped and classified as high-level actions (*e.g.*, `Taps`).

The classification of these actions involves three main parts: (i) a *grouping algorithm* that associates touches across subsequent frames into groups, which represent single discrete actions; (ii) *action translation*, which identifies the action type of the grouped touches; and (iii) *single/multi-fingered action identification*, which identifies which of the detected actions are single- and multi-fingered. The result of the *Action Classification* phase is a structured list of actions (*i.e.*, `Tap`, `Long Tap`, `Gesture`, or `Multi-Fingered Gesture`), where each action is a series of touches, or a series of actions in the case of Multi-Fingered Gestures, associated with video frames and screen locations.

3.3.1 Action Grouping

The first step of V2S+'s *action grouping* step filters out detected touches where the model's confidence is lower than 0.7. The second step groups touches belonging to the same atomic action according to a tailored heuristic and a graph connection algorithm. This procedure is necessary because discrete actions performed on the screen will persist across

1. We use "touch indicator" and "touch" interchangeably moving forward.

several frames, and thus, need to be grouped and segmented accordingly.

Grouping Consecutive Touches. The first heuristic groups touch indicators present in consecutive frames into the same group. As a measure taken to avoid (the rare occurrence of) a falsely detected touch indicator, touches that exist across two or fewer frames are discarded. This is due to the fact that, we observed in practice, even the quickest of touchscreen taps last across at least five frames.

Discrete Action Segmentation. There may exist successive single-fingered touches that were carried out extremely fast, such that there is no empty frame between two touches. In other cases, the touch indicator of one action may not fully disappear before the user starts a new action, leading to two or more touch indicators appearing in the same frame. These two situations are common when a user is swiping through a list and quickly tapping an option from the list, or typing quickly on the keyboard. Or, in the case of multi-fingered actions, two or more touches may appear in the same frame as part of the same gesture, but each component finger's action must be classified separately. V2S+ is able to determine where one action ends and another action begins as follows.

V2S+ analyzes groups of consecutive or overlapping touches and segments them into discrete actions per finger using a heuristic-based approach. We model the grouping of touch indicators as a graph connectivity problem (see Fig. 4). In this formulation, touch indicators are represented as nodes, vertical lines separate consecutive frames, and edges are possible connections between consecutive touches that make up a discrete action. The goal is to derive the proper edges for traversing the graph such that all touches for action A are in one group and all touches for action B are in another group (illustrated by the green edges in Fig. 4). Our algorithm decomposes the lists of consecutive touches grouped together into a graph. Starting from the first node, our algorithm visits each subsequent node and attempts to link it to the previous node. If there is only one node in a subsequent frame, then the two successive nodes are linked. If there is more than one node in a subsequent frame, our algorithm looks at the spatial distance between the previous node and both subsequent nodes, and groups previous nodes to their closer neighbors (as shown between frame t and $t+1$). However, if multiple nodes in one frame are at a similar distance from the previous node (as between frames $t+1$ and $t+2$ in Fig. 4), then the opacity of the nodes is used to perform the connection. For example, it is clear that node A in frame $t+2$ is a finger being lifted off the screen. Thus, it must be connected to the previously occurring action A (*i.e.,* in $t+1$), and not the action B that just started.

Finally, after this process, the opacity of all linked nodes are analyzed to determine further splits. Distinct actions with no empty frame between them may exist. Therefore, if a low-opacity node is detected in a sequence of successively connected nodes, they are split into distinct groups representing different actions.

At the end of this stage, distinct groups of touches that represent atomic actions (*e.g.*, `Tap`, `Long Tap`, or `Gesture`) are identified.

Algorithm 1: Single/Multi-Fingered Action Identification

Input: Detected Actions (DA)
Output: Single- and Multi-Fingered Actions (SFAs, MFAs)

```
DAc = countMultiTouchFrames(DA)
pMFAs = select(DA, DAc > 0.5) // Potential MFAs
SFAs = select(DA, DAc ≤ 0.5) // Initial SFAs

SDA = sortActions(pMFAs) // By starting frame
S = createNewStack() // Entries are action groups
initializeStack(S, SDA_1) // Pushes the 1st action

// For each action
foreach Action A ∈ SDA do
    AG = popActionGroup(S)
    // For each action in the group
    foreach Action at ∈ AG do
        if checkOverlap(A, at) then
            addToActionGroup(AG, A)
            break
        end
    end
    if notInActionGroup(A,AG) then
        PushNewActionGroup(S, A)
    end
end

MFAs = [] // List of MFAs
foreach Action Group AG ∈ S do
    c = countActions(AG)
    // Does the group have a single action?
    if c == 1 then
        addSFA(SFAs, AG)
    else
        addMFA(MFAs, AG)
    end
end
```

3.3.2 Action Translation

This process analyzes the derived groups of touches and classifies them based on the duration and touch locations of each group. Actions in which all of the touch indicators remain inside a specified small radius on the screen are classified as a `Tap` or a `Long Tap`. To correctly determine the threshold for this radius, we take direction from the Android Open Source Project, which defines the touch slop (*i.e.,* the distance in pixels that a touch can "wander" before it is no longer interpreted as a tap) to be 8px [49], [50], [51]. The action is classified as a `Tap` if it lasts across 20 or fewer frames, and as a `Long Tap` otherwise. Every other action is classified as a `Gesture`.

Filtering. V2S+ removes actions that have touch indicators with low average opacity (*e.g.,* $< 0.1\%$) across a group, as this could represent a rare *series* of misclassified touch indicators from the FASTER R-CNN. V2S+ also removes groups whose size is below or equals a threshold of two frames, as these might also indicate rare misclassifications.

After this stage, actions have been detected and classified, where each action is represented as a frame group with touch locations and a type (*i.e.,* `Tap`, `Long Tap`, or `Gesture`).

3.3.3 Single/Multi-Fingered Action Identification

It is crucial that V2S+ is able to detect and classify both *SFAs* and *MFAs*. Many applications rely on *MFAs* to accomplish core functionality within the app (*e.g.,* Google Maps utilizes two-finger rotations to allow users to rotate the map on the screen). If V2S+ cannot support these *MFAs*, then its utility

is limited to only certain applications and gestures, which is undesirable.

V2S+ first identifies frame groups that contain *Single-Fingered Actions* (SFAs) and *Multi-Fingered Actions* (MFAs).

MFAs are collections of *SFAs* that occur within the same frames, with each SFA corresponding to a single finger. After each action has been identified through the above steps (*i.e.,* described in Sect. 3.3.1 and 3.3.2), V2S+ regroups these actions based on their overlapping frames so that *SFAs* and *MFAs* can be classified appropriately and translated correctly into low-level events to be executed on the device (see Sect. 3.4). V2S+ executes Algorithm 1 to do so.

Initially, frame groups where more than 50% of the frames have multiple detected touches are considered *potential MFAs* (lines 1-2 in Algorithm 1). Other frame groups are considered as *SFAs* (line 3). Based on our initial experiments and observations, true *MFAs* have a percentage of frames with multiple detected touches much higher than 50% and *SFAs* have a much smaller percentage than 50%. Thus, 50% is a reasonable value for such a threshold. Furthermore, when a user is tapping quickly on the keyboard, touch indicators may appear on the screen at the same time as one another, yet each finger's actions still must be identified as *SFAs* rather than *MFAs*. By introducing this 50% threshold, we reduce the likelihood that these gestures will be linked together because it is unlikely that they overlap for over 50% of their duration.

V2S+ then sorts the *potential MFAs* by their starting frame in *sortActions* (line 4), and creates a stack (S) which holds groups of actions (*i.e., MFAs*), by calling *createNewStack* (line 5). Each group on the stack is a list of actions that execute simultaneously during the duration of the action group (on a device). The group on the top of the stack is the one currently being built. V2S+ initializes the stack (via *initializeStack*) by pushing the first chronological detected action onto the stack to begin the first group (line 6).

Next, V2S+ considers each of the sorted detected actions (line 7: $A \in SDA$) and each of the actions within the group being built on the top of the stack (lines 8-9: $at \in AG$). V2S+ uses the *checkOverlap* function to determine whether both actions A and at occur within common frames: if the first frame of A occurs before the last frame of at, the actions overlap and V2S+ appends A to the current action group AG (lines 10-13). For example, consider the action A=`Gesture:[3,...,20]` with `[3,...,20]` being the list of frames where the gesture occurs. For a group AG=[$\texttt{Gesture}_a$`:[1,...,20]`, $\texttt{Gesture}_b$`:[2,...,17]`], Algorithm 1 will append A to AG because the first frame of A (*i.e.,* frame #3) occurs before the last frame of $\texttt{Gesture}_a$ (*i.e.,* frame #20), as indicated by the frame indices (frame #3 $<$ frame #20).

If V2S+ has compared every action $at \in AG$ to A, and none of these actions overlap (line 15), V2S+ finishes building AG, and pushes A on the stack S, starting a new action group. Continuing with the above example, if AG is now [$\texttt{Gesture}_a$`:[1,...,20]`, $\texttt{Gesture}_b$`:[2,...,17]`, $\texttt{Gesture}_c$`:[3,...,20]`] and A is `Gesture:[70,...,92]`, V2S+ will not append A to AG because the starting frame of A occurs after the ending frame of any action $at \in AG$ (*i.e.,* frame #70 $>$ frame #20). Therefore, V2S+ pushes A onto S and A becomes the first action of new group being built in S. Since V2S+ first sorts the detected actions by their starting frames, it is safe to close AG when a given A does not belong to AG as any subsequent action will not overlap either.

After all of the actions have been processed, V2S+ iterates over the created action groups (line 20). If the group length is one (line 22), the group contains a single action and is considered as a *SFA* (line 23). Otherwise, V2S+ considers the group as a *MFA* (line 25).

Multi-Fingered Action Classification. Once V2S+ has successfully identified *SFAs* and *MFAs*, *MFAs* are further classified according to the number of fingers involved in the actions. This is computed by calculating the number of touch indicators per frame that is most frequent across all the frames in a *MFA*. For example, if the majority of the frames within a *MFA* contain three touch indicators, the MFA is classified as a `Three-Fingered Gesture`.

We used this approach because we observed that, in rare instances, V2S+ incorrectly classifies application imagery similar to the touch indicator as an additional `Tap` on the screen. When executing the *MFA*, this small `Tap` does not usually derail the successful execution of the scenario, so we do not remove it from the *MFA*. However, if V2S+ only relied on the number of actions detected within an *MFA* during the classification phase, the pipeline will then incorrectly classify the number of fingers in the *MFA*. For example, V2S+ would classify the `Three-Fingered Gesture` referenced above as a `Four-Fingered Gesture` because of this additional `Tap`, which harms V2S+'s classification accuracy. Instead, because most actions that compose an *MFA* start and end during the same frames and the additional `Tap` usually lasts a shorter duration, classification of the number of fingers for the *MFA* is more accurately represented by calculating the number of touch indicators per frame that is most frequent across all the frames in a *MFA*.

V2S+ supports classification of one- to ten-fingered gestures in an attempt to allow support for actions involving all fingers of a user's hands. We recognize that, in practice, higher-digit gestures occur infrequently, and thus in V2S+'s evaluation (see Sect. 5), we mainly focused on two-fingered *MFAs* (*e.g.,* pinch to zoom or rotations) when considering V2S+'s support for multi-touches.

3.4 Phase 3: Scenario Generation

After all the actions have been derived by the *Action Classification* phase, V2S+ proceeds by generating commands using the Android Debug Bridge (`adb`) that replays the classified actions on a device. To accomplish this, V2S+ converts the classified, high-level actions into low-level instructions in the `sendevent` command format, which is a utility included in Android's Linux kernel. Then, V2S+ uses a modified RERAN [52] binary to replay the events on a device.

Generating the Scenario Script. The `sendevent` command uses a limited instruction set in order to control the UI of an Android device. The main instructions of interest are the `start_event`, `end_event`, (x, y) coordinates where the user's finger touched the screen, and certain special instructions required by devices with older Android API levels. To create the script, low-level actions are generated in

chronological order, but *SFAs* and *MFAs* are translated using separate techniques because their translation procedures progress slightly differently.

Single-Fingered Actions. To generate single-fingered events, each action is exported starting with the `start_event` command (line 1 in Fig. 5). Then, for actions classified as a `Tap`, V2S+ provides a single (x, y) coordinate pair, derived from the center of the detected bounding box of the `Tap`. For `Gestures`, V2S+ iterates over each touch that makes up the `Gesture` action and appends the (x, y) pairs of each touch indicator to the list of instructions (lines 3-4, 7-8, *etc.* in Fig. 5). For `Long Taps`, V2S+ performs similar processing to that of `Gestures`, but instead uses only a single (x, y) pair from the initial detected touch indicator bounding box. Then, V2S+ ends the set of instructions for an action with the appropriate `end_event` command (*e.g.*, lines 275 and 276 in Fig. 5).

```
/*
   Time  |  Input Location  |Act. Code| Value
*/
1 [0.000] /dev/input/event1: 0003 0039 00000001 // set action ID
2 [0.000] /dev/input/event1: 0003 003a 00000081 // assign pressure
3 [0.000] /dev/input/event1: 0003 0035 00000184 // x-coord
4 [0.000] /dev/input/event1: 0003 0036 000000c3 // y-coord
5 [0.000] /dev/input/event1: 0003 0030 00000005 // touch radius
6 [0.000] /dev/input/event1: 0000 0000 00000000
7 [0.033] /dev/input/event1: 0003 0035 0000018c // x-coord
8 [0.033] /dev/input/event1: 0003 0036 000000c2 // y-coord
9 [0.033] /dev/input/event1: 0000 0000 00000000
10 [0.066] /dev/input/event1: 0003 0035 00000190 // x-coord
11 [0.066] /dev/input/event1: 0003 0036 000000c4 // y-coord
12 [0.066] /dev/input/event1: 0000 0000 00000000
...
272 [3.000] /dev/input/event1: 0003 0035 00000409 // x-coord
273 [3.000] /dev/input/event1: 0003 0036 000000c3 // y-coord
274 [3.000] /dev/input/event1: 0000 0000 00000000
275 [3.000] /dev/input/event1: 0003 003a 00000000 // pressure = 0
276 [3.000] /dev/input/event1: 0003 0039 ffffffff // mark end of action
277 [3.000] /dev/input/event1: 0000 0000 00000000
```

Fig. 5: `One-Fingered Gesture sendevent` example

Multi-Fingered Actions. V2S+ translates *MFAs* differently compared to *SFAs*. This is because, in order to achieve the intended replay behavior for a *MFA*, all of the touches that occur within the same frame must be sent to and executed on the device at the same time. This means that, rather than translate one finger's action in direct sequence as is accomplished for *SFAs* (see Fig. 5), each finger's action on the screen must be decomposed into its component touches per frame and translated along with the others that occur within the same frame. This requires including a command to alert the device which finger is being translated at each moment by indicating its slot value (*e.g.*, lines 7 and 14 in Fig. 6) [53].

Each finger's action is exported beginning with the `start_event` command (*e.g.*, lines 1 and 8 in Fig. 6). When one finger touch ends, V2S+ terminates this slot's behavior by appending the `end_event` command to the script (*e.g.*, lines 105-107 and 109-111 in Fig. 6). Because each finger's action is translated in association with a specific slot, separate finger's actions are allowed to continue even if another finger's action ends.

For all *SFAs* and *MFAs*, the speed and duration of each instruction is extremely important in order to accurately replay the user's actions. To derive the speed and duration of these actions, V2S+ adds timestamps to each (x, y) touch

```
/*
   Time  |  Input Location  |Act. Code| Value
*/
1 [0.000] /dev/input/event1: 0003 0039 00000000  // fing1 action ID
2 [0.000] /dev/input/event1: 0003 003a 00000032 // fing1 pressure
3 [0.000] /dev/input/event1: 0003 0035 00000188 // fing1 x-coord
4 [0.000] /dev/input/event1: 0003 0036 00000429 // fing1 y-coord
5 [0.000] /dev/input/event1: 0003 0030 00000005 // fing1 tap radius
6 [0.000] /dev/input/event1: 0000 0000 00000000
7 [0.000] /dev/input/event1: 0003 002f 00000001 // switch to fing2
8 [0.000] /dev/input/event1: 0003 0039 00000001 // fing2 action ID
9 [0.000] /dev/input/event1: 0003 003a 00000032 // fing2 pressure
10 [0.000] /dev/input/event1: 0003 0035 00000337 // fing2 x-coord
11 [0.000] /dev/input/event1: 0003 0036 00000323 // fing2 y-coord
12 [0.000] /dev/input/event1: 0003 0030 00000005 // fing2 tap radius
13 [0.000] /dev/input/event1: 0000 0000 00000000
14 [0.033] /dev/input/event1: 0003 002f 00000000 // switch fing1
15 [0.033] /dev/input/event1: 0003 0035 00000188
16 [0.033] /dev/input/event1: 0003 0036 00000429
17 [0.033] /dev/input/event1: 0000 0000 00000000
18 [10.641] /dev/input/event1: 0003 002f 00000001 // switch fing2
...
104 [1.485] /dev/input/event1: 0000 0000 00000000
105 [1.485] /dev/input/event1: 0003 002f 00000000 // switch fing1
106 [1.485] /dev/input/event1: 0003 003a 00000000 // fing1 pressure = 0
107 [1.485] /dev/input/event1: 0003 0039 ffffffff // end fing1
108 [1.485] /dev/input/event1: 0000 0000 00000000
109 [1.485] /dev/input/event1: 0003 002f 00000001 // switch fing2
110 [1.485] /dev/input/event1: 0003 003a 00000000 // fing2 pressure = 0
111 [1.485] /dev/input/event1: 0003 0039 ffffffff // end fing2
112 [1.485] /dev/input/event1: 0000 0000 00000000
```

Fig. 6: `Two-Fingered Gesture sendevent` example

location (see the *Time* values in Fig. 5 and Fig. 6) based on the timing between video frames (*i.e.*, for 30fps, there is a 33 millisecond delay between each frame), which will temporally separate each touch command sent to the device.

And, in order to determine the delays between both *SFAs* and *MFAs*, the timing between video frames is again used. Our required 30fps frame-rate provides V2S+ with millisecond-level resolution of event timings. Higher frame-rates can increase the fidelity of replay timing.

Scenario Replay. Once all the actions have been converted into low-level `sendevent` instructions, they are written to a log file. This log file is then fed into a *Translator* which converts the file into a runnable format that can be directly replayed on a device. This converted file along with a modified version of the RERAN engine [52] is pushed to the target device. We optimized the original RERAN binary to replay event traces more efficiently. Depending on the architecture of the target device, V2S+ provides a selection of RERAN binaries to allow for potential replay on a broader range of both physical and emulated devices[2]. Finally, the binary is executed using the converted file to faithfully replay the user actions originally recorded in the initial input video. We provide more examples of V2S+'s generated `sendevent` scripts, alongside our updated versions of the RERAN binaries in our online appendix [28].

4 V2S+ Implementation

The original V2S was implemented using a mix of Java, Python, and Bash, which made it difficult to use and manipulate to fit different research and development contexts. With this in mind, V2S+ was implemented entirely in Python with modularity in mind to encourage reuse and extension by future developers and researchers. Each phase

2. This has been particularly useful during the COVID-19 pandemic when physical access has been reduced.

and component can be easily substituted and/or extended by way of abstract classes which makes the entire pipeline malleable depending on the intended use case (*e.g.,* mobile application debugging, demonstration, *etc.*). In order to execute V2S+, users specify the paths to the desired input video, FASTER R-CNN, OPACITY CNN, `adb` binary, and the target device model in a modifiable `JSON` configuration file that is utilized throughout the pipeline.

4.1 Phase 1: Touch Detection

This phase of V2S+ reads the path to the input video that depicts a natural application usage scenario from the configuration file. V2S+ defines and utilizes a distinct *FrameExtractor (FE)* entity to prepare the video for touch detection. This class utilizes the `FFmpeg` [46] tool to normalize the input video frame rate to 30fps and extract individual frames. Individual implementations of this portion of V2S+ can be substituted by implementing the *AbstractVideoManipulator* class.

Then, V2S+ allows its *TouchDetector (TD)* class to utilize the pre-trained FASTER R-CNN specified in the configuration file to locate individual touches within each of the frames extracted by the *FE*. This *TD* class can be reconfigured and customized by implementing the abstract *AbstractTouchDetector* super class.

And finally, V2S+ utilizes an *OpacityDetector (OD)* class, which mobilizes the OPACITY CNN specified in the configuration file, to classify each touch localized by the *TD* class as having either a high or low opacity value. Again, this class's behavior can be modified or customized using the *AbstractOpacityDetector* super class.

The output of Phase 1 is a structured `JSON` file that contains all the detected touches details including their corresponding frame number, spatial location on the screen, and detected opacity.

4.2 Phase 2: Action Classification

This phase receives Phase 1's structured `JSON` output file as input. Then, V2S+'S *GUIActionClassifier (GAC)* is responsible for detecting and classifying the *SFAs* and *MFAs* depicted in the input video based on the input from Phase 1. This can be altered by implementing the *AbstractGUIActionClassifier* class. The output of Phase 2 is another structured `JSON` file that includes each classified action's component touches (or actions in the case of *MFAs*).

4.3 Phase 3: Scenario Generation

This phase utilizes Phase 2's `JSON` file as input. V2S+'S *Action2EventConverter (A2EC)* then uses `sendevent` commands to translate the high-level actions detected by the *GAC* into a low-level replayable output script. Depending on the specifications necessary to do so, developers can implement the *AbstractAction2EventConverter* class to alter the behavior of this class.

V2S+ then connects to the intended target device using the Android Debug Bridge (`adb`) and pushes the script to the device along with the appropriate pre-compiled RERAN [52] binary. V2S+ then executes the replay and saves the recording as output to complete the pipeline.

5 EMPIRICAL EVALUATION

In this section, we describe the methodology we used to evaluate V2S+. The *goal* of our empirical study is to assess the *accuracy*, *robustness*, *performance*, and *industrial utility* of V2S+. The *context* of this evaluation consists of: (i) sets of 15,000 and 10,000 images, corresponding to the evaluation of V2S+'S FASTER R-CNN and OPACITY CNN respectively; (ii) a set of over 90 Android applications including 64 of the top-rated apps from Google Play, 5 open source apps with real crashes, 5 open source apps with known bugs, 5 open source apps with controlled crashes, 14 hybrid apps, and 4 apps that have multi-touch capabilities; (iii) 2 popular physical target Android devices (Nexus 5 and Nexus 6P) and 1 emulated target Android device (Nexus 5). The main *quality focus* of our study is the extent to which V2S+ can generate replayable scenarios that mimic original user app usages.

To achieve our evaluation goals, we formulated the following six research questions:

- $\boldsymbol{RQ_1}$*: How accurate is* V2S+ *in identifying the location of the touch indicator on the screen?*
- $\boldsymbol{RQ_2}$*: How accurate is* V2S+ *in identifying the opacity of the touch indicator?*
- $\boldsymbol{RQ_3}$*: How effective is* V2S+ *in generating a sequence of events that accurately mimics single-fingered user actions from video recordings of different applications?*
- $\boldsymbol{RQ_4}$*: How effective is* V2S+ *in generating a sequence of events that accurately mimics multi-fingered user actions from video recordings of different applications?*
- $\boldsymbol{RQ_5}$*: What is* V2S+*'s overhead in terms of scenario generation?*
- $\boldsymbol{RQ_6}$*: Do practitioners perceive* V2S+ *as useful?*

5.1 RQ_1: Accuracy of Faster R-CNN

To answer RQ_1, we first evaluated the ability of V2S+'S FASTER R-CNN to accurately identify and localize the *touch indicators* present in screen recording frames with *bounding boxes*. To accomplish this, we followed the procedure to generate training data outlined in Sec. 3.2 with the 70%–30% split for the training and testing sets, respectively. The implementation of the FASTER R-CNN object detection used by V2S+ is coupled to the size of the images used for training and inference. Thus, to ensure V2S+'S model functions across different devices, we trained many FASTER R-CNN models.

For the original implementation of V2S+, we trained two distinct FASTER R-CNN models (*i.e.,* one for Nexus 5 and one for Nexus 6P). For all of these models, we resized the images from the REDRAW dataset to the target device image size. As we show in the course of answering other RQs, we found that resizing the images in the already large REDRAW dataset [54], as opposed to re-collecting natively sized images for each device, resulted in highly accurate detections in practice.

We used the *TensorFlow Object Detection API* [48] to train our model. Moreover, for the training process, we modified several of the hyper-parameters after conducting an initial set of experiments. These changes affected the number of classes (*i.e.,* 1) and the maximum number of detections per class or image (*i.e.,* 10). We also modified the learning rate

(a) Touch indicators (b) False positive detections

Fig. 7: Touch indicators and failed detections

for V2S+ after 50k (*i.e.,* 3×10^{-5}) and 100k (*i.e.,* 3×10^{-6}) iterations. Each training process was run for 150k steps with a batch size of 1, and our implementation of FASTER R-CNN utilized a VGGNET [30] instance pre-trained on the MSCOCO dataset [55]. We provide our full set of model parameters in our online appendix [28].

To validate the accuracy of V2S+'S FASTER R-CNN models we utilize *Mean Average Precision* (mAP) which is commonly used to evaluate techniques for the object detection task. This metric is typically computed by averaging the *precision* over all the categories in the data; however, given that we have a single class (the touch indicator icon), we report results only for this class. Thus, our mAP is computed as $mAP = TP/(TP + FP)$ where TP corresponds to an identified image region with a correct corresponding label, and FP corresponds to the identified image regions with the incorrect label, which in our case would be an image region that is falsely identified as a touch indicator. We use different values of the Intersection Over Union (IoU) [41] between the region of the prediction and the ground truth region (IoU thresholds ranging from 0.5 to 0.95 with 0.05 increments), to determine if the prediction is a TP or FP. Additionally, we evaluate the *Average Recall* of our model in order to determine if our model misses detecting any instances of the touch indicator. This is computed by $AR = TP/k$ where TP is the same definition stated above, and k corresponds to the total number of possible TP predictions.

During preliminary experiments with FASTER R-CNN using the default *touch indicator* (see Fig. 7a), we found that, due to the default touch indicator's likeness to other icons and images present in apps, it was prone to very occasional false positive detections (Fig. 7b). Thus, we analyzed particular cases in which the default touch indicator failed and replaced it with a more distinct, high-contrast touch indicator (see Fig. 7a). We found that this custom touch indicator marginally improved the accuracy of our models. It should be noted that replacing the touch indicator on a device, requires the device to be rooted. While this is an acceptable option for most developers, it may prove difficult for end-users. However, even with the default touch indicator, V2S+'S FASTER R-CNN model still achieves extremely high levels of accuracy. For these reasons, we only evaluate the custom touch indicator for RQ_1 and RQ_2.

5.2 RQ_2: Accuracy of Opacity CNN

To answer RQ_2, we evaluated the ability of V2S+'s OPACITY CNN to predict whether the opacity of the touch indicator is solid or semi-transparent. To accomplish this, we followed dataset generation procedure outlined in Sec. 3.2, where equal number of full and partial opacity examples are generated. Thus, the generated dataset contains equal numbers of full and partial opacity examples for a total of 10k, which are evenly split into 70%–30% training and testing sets. We used the TensorFlow framework in combination with Keras to implement the OPACITY CNN. In contrast to the FASTER R-CNN model previously used, we do not need to create a separate model for each device. This is due to the *touch indicator* being resized when fed into the OPACITY CNN. Similarly to the FASTER R-CNN, we evaluate OPACITY CNN using *mAP* across our two classes (solid and semi-transparent).

5.3 RQ_3: Replay Accuracy for Scenarios with Single-Fingered Actions

To answer RQ_3, we carried out three different studies to assess the accuracy of V2S+ with *SFAs* within different types of applications: 2 studies with native applications and 1 study with hybrid applications.

5.3.1 Replay Accuracy on Native Applications

We carried out two studies designed to assess both the *depth* and *breadth* of V2S+'S abilities to reproduce user events depicted in screen recordings within native applications. The first, the *Controlled Study*, measures the depth of V2S+'S abilities through a user study during which we collected real videos from end users depicting: bugs, real crashes, synthetically injected crashes, and normal usage scenarios for 20 apps (*e.g.,* Car Report, Another Monitor, *etc.*). Next in the *Popular Applications Study* we measured the breadth of V2S+'S abilities by recording scenarios for a larger, more diverse set of 64 most popular apps from the Google Play (*e.g.,* Tasty, Twitter, Airbnb, *etc.*). We provide the full details of these apps in our online appendix [28].

Controlled Study. In this study, we considered four types of recorded usage scenarios depicting: (i) normal usages, (ii) bugs, (iii) real crashes, and (iv) controlled crashes. Normal usage scenarios refer to video recordings exercising different features on popular apps. Bug scenarios refer to video recordings that exhibit a bug on open source apps. Finally, controlled crashes refer to injected crashes into open source apps. This allows us to control the number of steps before the crash is triggered.

For this study, eight participants including 1 undergraduate, 3 masters, and 4 doctoral students were recruited from William & Mary (approved by the Protection of Human Subjects Committee (PHSC) at W&M under protocol PHSC-2019-01-22-13374) to record the videos, with each participant recording eight separate videos, two from each of the categories listed above. Four participants recorded videos on the physical Nexus 5 and four used the physical Nexus 6P. This accounts for a total of 64 videos, from 20 apps evenly distributed across all scenarios. Before recording each app, participants were either asked to use the app to become familiar with it, or read a bug/crash rep1ort before reproducing the fault. All of the real bugs and crashes were taken from past studies on mobile testing and bug reporting [56], [57], [58]. These videos depict a total of 1,141 *SFAs*, with an average of ≈ 18 *SFAs* per video.

Popular Applications Study. In this study, we considered a larger and more diverse set of apps from Google Play. Specifically, we downloaded the two highest-rated apps from each of non-game categories (*i.e.,* 32) for a total of 64 applications (see Table 4).

Two of the authors then recorded two scenarios per app accounting for 32 apps each, one using a physical Nexus 5

and the other using a physical Nexus 6P. The authors strived to use the apps as naturally as possible, and this evaluation procedure is in line with past evaluations of Android record-and-replay techniques [52], [59]. The recorded scenarios represented specific use cases of the apps that exercise at least one of the major features, and were independent of one another. During our experiments, we noticed certain instances where our recorded scenarios were not replicable, either due to non-determinism or dynamic content (*e.g.,* random popups appearing). Thus, we discarded these instances and were left with 111 app usage scenarios from 60 apps. It is worth noting that it would be nearly impossible for existing techniques such as RERAN [52] or Barista [60] to reproduce the scenarios due to the nondeterminism of the dynamic content, hence our decision to exclude them. These usage scenarios depict a total of 2,150 *SFAs* and an average of ≈ 19 *SFAs* per usage video.

5.3.2 Replay Accuracy on Hybrid Applications

We carried out a study to assess V2S+'s ability to reproduce scenarios on hybrid applications, *i.e.,* those implementing features natively using Android's API and features using web-based technologies. This study is motivated by the increasing prevalence of hybrid applications in the Google Play store [25] and the current relative lack of GUI testing tools for these applications. We chose to evaluate V2S+'s performance with hybrid applications, separately from native applications, to demonstrate that V2S+ is one of few tools to support both hybrid and native apps.

Popular Applications Study. In this study, we utilized 14 apps with hybrid features from 7 popular application categories – 2 per category (see Table 5). For each of these 14 apps, we devised 2 scenarios that interacted with the application's core hybrid functionality. We recruited 7 participants for the study (6 undergraduate students and 1 graduate student). Each of the 28 scenarios was recorded twice within the group, which allowed for each participant to record 8 different scenarios on a Nexus 5 emulator. We used an emulator (rather than a physical device) because of restrictions related to COVID-19. In an attempt to allow participants to interact with each application as naturally as possible, each participant was allowed to familiarize themselves with the application and with the scenario before recording. A total of 56 videos were collected for this study, but due to non-deterministic or dynamic content or `apk` issues, we discarded 3 videos. These videos depict a total of 622 *SFAs*, for an average of ≈ 11 *SFAs* per video.

5.3.3 Scenario Evaluation

In addition to producing the video recordings, participants were asked to watch their scenarios and record their (ground truth) sequence of actions depicted in the videos. Participants were asked to classify each action depicted in each scenario as one of three options: (i) a `Tap`, an action where the touch indicator remains in approximately the same place and lasts for a short duration; (ii) a `Long Tap`, an action where the touch indicator remains in about the same place for the entire duration of the action and the duration of the action lasts over a longer period than a `Tap` (*i.e.,* usually about 1 second or more); or (iii) a `Gesture`, an action where the touch indicator moves about the screen that can be short or long in duration. An action sequence is, then, a sequence of action types (according to the three types defined above). After the videos and action sequences were collected, two authors verified the correctness of: (1) the recordings; and (2) the action sequences (compared to the recordings). Only in three cases, we revised the action sequence to be properly aligned with the video and/or we trimmed out parts at the end or beginning of the video that were not relevant to app usage scenario, including cases where participants had included opening or closing the application or interacting with other applications in their recordings.

To measure how accurately V2S+ replays videos that depict *SFAs*, we use 3 different metrics. To compute these metrics, we manually derived the ground truth sequence of action types for each recorded video. First, we use Levenshtein distance, which is commonly used to compute distances between words at character level, to compare the original list of action types to the list of classified actions generated by V2S+. Thus, we consider each type of action being represented as a character, and scenarios as sequences of characters which represent series of actions. A low Levenshtein distance value indicates fewer required changes to transform V2S+'s output to the ground truth set of actions. Additionally, we compute the longest common subsequence (LCS) to find the largest sequence of each scenario from V2S+'s output that aligns with the ground truth scenario from the video recording. For this LCS measure, the higher the percentage (between the LCS and the ground truth sequence), the closer V2S+'s trace is to a perfect match of the original trace. Moreover, we also computed the precision and recall for V2S+ to predict each type of action across all scenarios when compared to the ground truth. Finally, in order to validate the fidelity of the replayed scenarios generated by V2S+ compared to the original video recording, we manually compared each original video to each reproduced scenario from V2S+, and determined the number of actions for each video that were faithfully replayed.

5.4 RQ_4: Replay Accuracy for Scenarios with Multi-Fingered Actions

To evaluate RQ_4, we conducted a study of popular applications involving one of the authors to demonstrate V2S+'s ability to correctly identify and replay *MFAs* that are commonly seen in the wild. We studied 4 different native applications in 2 different categories in the Google Play store as shown in Table 1. These categories were chosen because they are known to host many applications that rely on *MFAs* when using core app functionality. Within each of the 4 applications, we enumerated the supported *MFAs* that result in specific behavior (also listed in Table 1) and designed scenarios that utilized these in both *artificial* and *natural scenarios*. *Artificial scenarios* are scenarios where the user simply executes multiple *MFAs* in sequence; these types of scenarios were recorded to ensure that V2S+ can replicate the breadth of *MFAs* that each application supports. *Natural scenarios* are those that utilize basic functionality of the application and also incorporate one or more *MFAs*; these were included in the study to ensure that V2S+ can accurately replay *MFAs* when they occur in sequence with other actions supported

TABLE 1: Multi-Touch Study Applications

App Category	Apps	Actions Demonstrated
Art & Design	Infinite Painter	Pinch to zoom in Pull to zoom out Two-finger rotation
	Sketchbook	Pinch to zoom in Pull to zoom out Two-finger rotation
Navigation	Google Maps	Pinch to zoom in Pull to zoom out Two-finger rotation Two-finger tap Two-finger "shove"
	Waze	Pinch to zoom in Pull to zoom out Two-finger rotation Two-finger tap

by V2S+ (*e.g.,* `Taps`). For the Art & Design applications, we designed 1 natural scenario and 1 artificial scenario. For the Navigation applications, we designed 2 natural scenarios and 2 artificial scenarios. We chose to record more scenarios for Navigation apps than Art & Design applications because there are more relevant multi-fingered actions in Navigation apps, as shown in Table 1. This study comprised 12 videos that depicted 115 actions: 44 *MFAs* and 71 *SFAs*. These videos must include *SFAs* in addition to MFAs because the natural scenarios required SFAs between the MFAs to create meaningful interactions with the applications. On average, each video contained ≈ 4 *MFAs* and ≈ 6 *SFAs*.

One author of this paper recorded each of these 12 scenarios on a physical Nexus 5 Android device. This author familiarized herself with the applications and the scenarios before recording to ensure that the scenarios progressed in a natural fashion. Then, after filming the videos, this author rewatched each recording and classified each action either as a (i) `Tap`, a (ii) `Long Tap`, or a (iii) `Gesture`. These classifications were then used for empirically verifying the performance of V2S+ at replicating the sequences of gestures using the metrics listed in Sec. 5.3.3.

5.5 RQ_5: Performance

To address RQ_5, we evaluated V2S+ by measuring the average time it takes for a video to pass through each of the three phases of the V2S+ approach on commodity hardware (*i.e.,* a single NVIDIA GTX 1080Ti). We see this as a worst case scenario for V2S+ performance, as our approach could perform substantially faster on specialized hardware. Note that we modified the RERAN engine to run fast enough for more precise swipes. Therefore, we expect our V2S+ to have a lower overhead compare to the one reported in RERAN's respective paper [52].

5.6 RQ_6: Perceived Usefulness

Ultimately, our goal is to integrate V2S+ into real-world development environments. Thus, as part of our evaluation, we investigated V2S+'s perceived usefulness with three developers who build Android apps for their respective companies.

The developers (*a.k.a.* participants) were contacted through direct contact of the authors. Participant #1 (P1) was a front-end developer on the image search team of the Google Search app [61], participant #2 (P2) is a developer of the 7-Eleven Android app [62], and participant #3 (P3) is a backend developer for the Proximus shopping basket app [63]. We interviewed the participants using a set of questions organized in two sections. The first section aimed to collect information on participants' background, including their role at the company, the information used to complete their tasks, the quality of this information, the challenges of collecting it, and how they use videos in their every-day activities. The second section aimed to assess V2S+'s potential usefulness as well as its accuracy in generating replayable scenarios. This section also asked the participants for feedback to improve V2S+ including likert scale questions. We provide the complete list of interview questions used in our online appendix [28], and discuss selected questions in Sec. 6.5.

The participants answered questions from the second section by comparing two videos showing the same usage scenario for their respective app: one video displaying the scenario manually executed on the app, and the other one displaying the scenario executed automatically via V2S+'s generated script. Specifically, we defined, recorded, and manually executed a usage scenario on each app. Then, we ran V2S+ on the resulting video recordings. To define the scenarios, we identified a feature on each app involving any of the action types (*i.e.,* taps, long taps, and gestures). Then, we generated a video showing the original scenario (*i.e.,* video recording) and next to it the replayed scenario generated when executing V2S+'s script. Both recordings highlight the actions performed on the app. We presented the video to participants as well as V2S+'s script with the high-level actions automatically identified from the original video.

TABLE 2: Touch Indicator Detection Accuracy - Original

Model	Device	mAP	mAP@.75	AR
FASTER R-CNN-Original	Nexus 5	97.36%	99.01%	98.57%
FASTER R-CNN-Original	Nexus 6P	96.94%	99.01%	98.19%
FASTER R-CNN-Modified	Nexus 5	97.98%	99.01%	99.33%
FASTER R-CNN-Modified	Nexus 6P	97.49%	99.01%	99.07%

6 Evaluation Results

We present and discuss the results of V2S+'s evaluation.

6.1 RQ_1: Accuracy of FASTER R-CNN

Table 2 depicts the precision and recall for V2S+'s FASTER R-CNN network for touch indicator detection on different devices and datasets. The first column identifies the usage of either the default touch indicator or the modified version. The second column describes the target device for each trained model. The third column provides the mAP, which is calculated based on the CoCo detection metric[3] that uses different values of the Intersection Over Union (IoU) [41] between the area of the prediction and the ground truth area (IoU thresholds ranging from 0.5 to 0.95 with 0.05 increments). The fourth column shows the mAP when the IoU between the area of the prediction and the ground truth is above 75%. All models achieve a least ≈97% mAP, indicating that ≈97% of the touch indicator predictions made

3. https://cocodataset.org/detection-eval

TABLE 3: Confusion Matrix for Opacity CNN. Low Opacity Original (LO-Orig.), High Opacity Original (HO-Orig.), Low Opacity Custom (LO-Cust.), High Opacity Custom (HO-Cust.)

	Total	LO-Orig.	HO-Orig.	LO-Cust.	HO-Cust.
Low Op	5000	97.8%	2.2%	99.7%	0.3%
High Op	5000	1.4%	98.6%	0.8%	99.2%

by V2S+'s object detection network are correct. mAP only improves when we consider bounding box IoUs that match the ground truth bounding boxes by at least 75%, which illustrates that when the model is able to predict a reasonably accurate bounding box, it nearly always correctly detects the touch indicator (in ≈99% of the cases). As illustrated by the last column in Table 2, the model also achieves extremely high recall, detecting at least ≈98% of the touch indicators.

> **Answer to RQ$_1$**: V2S+ benefits from the strong performance of its object detection technique to detect touch indicators. All FASTER R-CNN models achieved at least ≈ 97% precision and at least ≈ 98% recall across devices.

6.2 RQ$_2$: Precision of the OPACITY CNN

To illustrate the OPACITY NETWORK's precision in classifying the two opacity levels of touch indicators, we present the confusion matrix in Table 3. The results are presented for both the default and modified (aka customized) touch indicator. The overall precision for the original touch indicator is 98.2% whereas for the custom touch indicator is 99.4%. These percentages are computed by dividing the correctly predicted label (i.e., Low/High-Opacity) by the number of members of that label for the original and custom touch indicators. Hence, it is clear V2S+'S OPACITY CNN is highly precise at distinguishing between differing opacity levels.

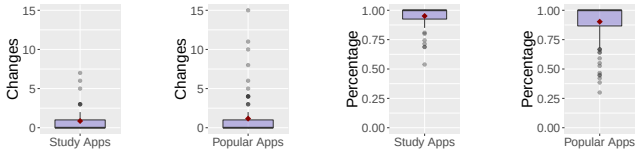
> **Answer to RQ$_2$**: V2S+ benefits from the CNNs precision in classifying levels of opacity. OPACITY CNN achieved an average precision above 98% for both touch indicators.

6.3 RQ$_3$: Replay Accuracy for Scenarios with Single-Fingered Actions

We present and discuss the replay accuracy results of V2S+ for scenarios with single-fingered actions, for both native and hybrid applications.

6.3.1 Replay Accuracy on Native Applications

Levenshtein Distance. Fig. 8a and 8b depict the number of changes required to transform the output event trace into the ground truth for the apps used in the *Controlled Study* and the *Popular Apps Study*, respectively. For the controlled study apps, on average it requires 0.85 changes (*i.e.*, # of action changes) per user trace to transform V2S+'S output into ground truth event trace, whereas for the popular apps it requires slightly more with 1.17 changes. Overall, V2S+ requires minimal changes per event trace, being very similar to the ground truth.



(a) LD-Study (b) LD-Popular (c) LCS-Study (d) LCS-Popular

Fig. 8: Effectiveness Metrics - Native Apps

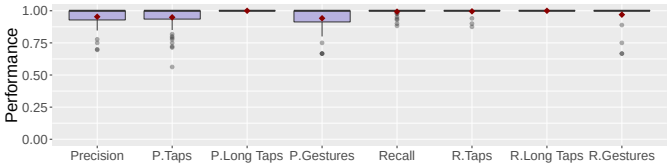


Fig. 9: Precision and Recall - Controlled Study (Native Apps)

Longest Common Subsequence. Fig. 8c and 8d presents the percentage of events for each trace that match those in the original recording trace for the *Controlled Study* and *Popular Apps Study* respectively. On average V2S+ is able to correctly match 95.1% of sequential events on the ground truth for the controlled study apps and 90.2% for popular apps. These results suggest that V2S+ is able to generate sequences of actions that closely match the original trace in terms of action types.

Precision and Recall. Fig. 9 and 10 show the precision and recall results for the *Controlled Study* and *Popular Apps Study*, respectively. These plots were constructed by creating an order agnostic "bag of actions" for each predicted action type, for each scenario in our datasets. Then, precision and recall are calculated by comparing the actions to a ground truth "bag of actions" and counting True/False Positives and False Negative. Finally, an overall average precision and recall are calculated across all action types. The results indicate that on average, the precision of the event traces is 95.3% for the controlled study apps and 95% for popular apps. This is also supported for each type of event showing also a high level of precision across types except for the precision on `Long Taps` for the popular apps. This is mainly due to the small number (*i.e.*, 9 `Long Taps`) of this event type across all the popular app traces. Also, Fig. 9 and 10 illustrate that the recall across action types is high with an average of 99.3% on controlled study apps and 97.8% on the popular apps for all types of events. In general, we conclude that V2S+ can accurately predict the correct number of event types across traces.

Success Rate. We also evaluated success rate of each replayed action for all scenarios across both RQ$_3$ studies. The 175 videos were analyzed manually and each action was marked as successful if the replayable scenario faithfully exercised the app features according to the original video. This means that in certain cases, videos will not *exactly* match the original video recording (*e.g.*, due to a single keyboard keystroke error that still led to the same feature result).

After validating all 64 videos for the controlled study, V2S+ fully reproduces 93.75% of the scenarios, and 94.48% of the consecutive actions. V2S+ fully reproduced 96.67% of

TABLE 4: Detailed Results for RQ_3 popular applications study. Green cells indicate fully reproduced videos, Orange cells >50% reproduced, and Red Cells<50% reproduced. Blue cells show non-reproduced videos due to non-determinism/dynamic content.

AppName	Rep. Actions		AppName	Rep. Actions		App Name	Rep. Actions		App Name	Rep. Actions		App Name	Rep. Actions	
Ibis Paint X	11/11	36/36	Firefox	22/22	N/A	Tasty	20/20	14/36	SoundCloud	12/12	13/13	LetGo	17/17	15/15
Pixel Art Pro	20/20	9/9	MarcoPolo	12/12	30/30	Postmates	26/26	12/12	Shazam	12/15	20/20	TikTok	14/14	11/11
Car-Part.com	20/20	16/16	Dig	13/13	N/A	Calm	9/9	11/16	Twitter	14/14	19/19	LinkedIn	18/18	13/13
CDL Practice	8/8	13/13	Clover	15/15	19/19	Lose It!	36/36	N/A	News Break	1/18	9/9	CBSSports	25/25	16/16
Sephora	4/9	13/13	PlantSnap	39/39	18/24	U Remote	14/14	18/18	FamAlbum	19/19	8/26	MLBatBat	11/13	N/A
SceneLook	14/14	16/16	Translator	20/20	28/28	LEGO	52/52	24/24	Baby-Track	14/14	12/12	G-Translate	14/17	15/15
KJ Bible	16/16	19/19	Tubi	2/19	30/30	Dev Libs	35/35	22/22	Walli	8/8	N/A	G-Podcast	9/9	15/15
Bible App	12/12	15/15	Scan Radio	24/27	N/A	Horoscope	24/24	19/19	ZEDGE	9/9	18/18	Airbnb	9/9	14/14
Indeed Jobs	15/15	19/19	Tktmaster	30/30	14/14	Waze	17/17	19/19	G-Photo	18/18	18/18	G-Earth	8/13	1/30
UPS Mobile	16/16	19/24	Greet Cards	23/23	N/A	Transit	26/26	18/18	PicsArt	18/18	39/39	DU Record	15/15	9/9
Webtoon	17/17	15/21	QuickBooks	47/47	28/28	WebMD	7/34	7/26	G-Docs	3/26	N/A	AccuWeath	13/13	21/21
MangaToon	16/16	28/28	Yahoo Fin	23/23	N/A	K-Health	10/10	15/24	M. Outlook	27/27	21/26	W. Radar	14/14	13/13

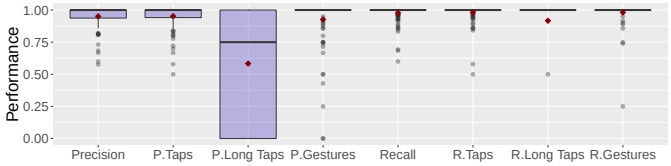
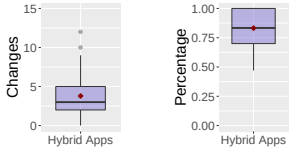


Fig. 10: Precision and Recall - Popular Native Apps



(a) LD-Hybrid (b) LCS-Hybrid

Fig. 11: Effectiveness Metrics - Hybrid Apps

the scenarios for bugs and crashes and 91.18% of apps usages. Detailed results for the *Popular Apps Study* are shown in Table 4, where each app, scenario (with total number of actions), and successfully replayed actions are displayed. *Green* cells indicate a fully reproduced video, *Orange* cells indicate more than 50% of events reproduced, and *Red* cells indicate less than 50% of reproduced events. *Blue* cells show non-reproduced videos due to non-determinism/dynamic content. For the 111 scenarios recorded for the popular apps, V2S+ fully reproduced 81.98% scenarios, and 89% of the consecutive actions. Overall, this signals strong replayability performance across a highly diverse set of applications. Instances where V2S+ failed to reproduce scenarios are largely due to minor inaccuracies in `Gesture` events due to our video resolution of 30fps. We discuss potential solutions to this limitation in Sec. 8.

> **Answer to RQ_3 (Native apps)**: For native popular applications used on physical devices, V2S+ is capable of generating event traces that require on average ≈ 1 change to match original user scenarios. Moreover, at least 90.2% of events match the ground truth, when considering the sequence of event types. Overall, precision and recall are ≈95% and ≈98% respectively for event types produced by V2S+. Finally, in 96.67% and 91.18% of the cases, V2S+ successfully reproduces bugs/crashes- and app-usage-related videos, respectively.

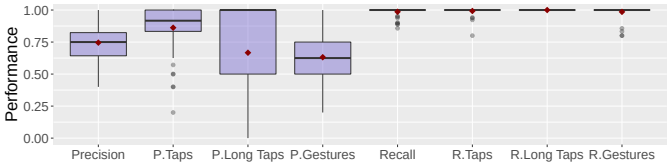


Fig. 12: Precision and Recall - Hybrid Apps

TABLE 5: Detailed Results for RQ_3 hybrid scenarios study. Green cells indicate fully reproduced videos, Orange cells >50% reproduced, and Red Cells<50% reproduced. Blue cells show non-reproduced videos due to non-determinism/dynamic content or `apk` issues.

AppName	Scen. 1 Rep. Actions		Scen. 2 Rep. Actions	
Wikihow	4/4	1/7	3/9	10/10
Wikipedia	3/14	13/13	23/23	17/17
Platt	2/10	14/14	3/3	3/3
USPS	16/16	16/16	24/24	18/23
CNBC	10/10	10/10	11/11	14/14
Seeking Alpha	7/7	5/5	0/5	1/7
Old Navy	7/9	1/11	0/6	N/A
Pinterest	22/22	0/21	24/24	24/24
Epocrates	10/13	12/17	10/10	6/6
Leafly	18/18	11/11	N/A	9/9
Newsbreak	5/5	5/5	9/9	5/5
Guardian	5/9	1/8	9/9	9/9
Rainbow	12/12	2/19	12/12	11/11
VS	12/12	5/11	2/10	N/A

6.3.2 Replay Accuracy on Hybrid Applications

Levenshtein Distance. Fig. 11a depicts the number of changes required to transform V2S+'s event traces into the ground truth traces for apps used in the *Hybrid Study*. In this study, 3.79 changes are required on average per user trace to transform V2S+'s output into the ground truth event trace.

Longest Common Subsequence. Fig. 11b presents the percentage of events for each trace that match those in the original recording trace for the *Hybrid Study*. On average V2S+ is able to correctly match 83.19% of sequential events on the ground truth for these hybrid applications.

Precision and Recall. Fig. 12 details the precision and recall results for the *Hybrid Study*. The results indicate that on average, the precision of the event traces is 74.54% for hybrid applications. Fig. 12 also illustrates that the recall across action types is high with an average of 98.63% for all types of events.

Success Rate. Finally, we also evaluated success rate of each replayed action in our *Hybrid Study*. The 56 videos were analyzed manually and each action was marked as successful if the replayable scenario faithfully exercised the app features according to the original video.

After validating all 56 videos for this study, 3 were removed because of non-determinism or problems with

the apks. V2S+ fully reproduces 39 out of 53 scenarios (73.58%), and 486 out of 622 of the consecutive actions (78.15%). Detailed results are shown in Table 5, where each app, scenario (with total number of actions), and successfully replayed actions are displayed.

Discussion. The results show that V2S+'s performance is lower for hybrid apps than for native apps. Due to restrictions related to COVID-19, we chose to conduct the hybrid-app study virtually and required participants to record scenarios on emulated devices on their personal machines. Our results analysis reveals that the use of these Android emulators in place of physical devices affected the quality of our results in two main ways: (i) utilizing a mouse or track pad to complete actions on an emulator is unnatural as it does not allow for the same ease of movement as a finger on a physical device, which resulted in more "jagged" actions that are more often incorrectly classified and replayed by V2S+; and (ii) the speed performance of the emulator is directly proportional to the speed of the computer's processing power, which manifested in our study as some participants' videos depicted "laggy" actions that are not well-detected or replayed by V2S+. We discuss potential solutions to these limitations in Sec. 8.

We did not find any particular app-type-related factors that impacted the replay results for hybrid apps. As such, the gap in our observed results (between hybrid and native apps) is likely not due to differences in the types of apps or V2S+'s ability to support them, but rather to limitations of the video collection procedure.

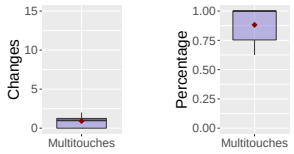
> **Answer to RQ$_3$ (Hybrid apps)**: For hybrid applications used on emulated devices, V2S+ is capable of generating event traces that require on average ≈ 3.5 changes to match original user scenarios. Moreover, at least 83% of events match the ground truth when considering the sequence of event types. Overall, precision and recall are $\approx$ 75% and $\approx$ 99% respectively for event types produced by V2S+.

6.4 RQ$_4$: Replay Accuracy for Scenarios with Multi-Fingered Actions

We discuss the replay accuracy results of V2S+ for scenarios with multi-fingered actions.

Levenshtein Distance. Fig. 13a depicts the number of changes required to transform the output event trace into the ground truth for the apps used in the *Multi-Touch Study*. In this study, ≈ 0.92 changes are required on average per user trace to transform V2S+'s output into the ground truth event trace. If we separate the videos into their *artificial* and *natural* categories, *artificial* scenarios only require on average ≈ 0.33 changes per trace and *natural* videos require on average ≈ 1.5 changes. If, instead, we separate the actions depicted in these videos into sequences of either *SFAs* or *MFAs*, then *SFAs* require an average of one change per sequence and *MFAs* require 0.083 changes per sequence, on average.

Longest Common Subsequence. Fig. 13b presents the percentage of sequential events for each trace that match those in the original recording trace for the *Multi-Touch Study*.



(a) LD-Multi (b) LCS-Multi

Fig. 13: Effectiveness Metrics - Multi-Fingered Scenarios

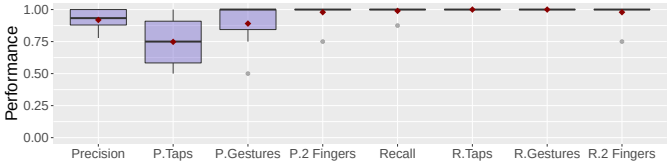


Fig. 14: Precision and Recall - Multi-Fingered Actions

On average V2S+ is able to correctly match $\approx$ 88.2% of sequential events on the ground truth for these multi-touch scenarios. If we separate the videos into their *artificial* and *natural* categories, V2S+ is able to correctly match $\approx$ 96.7% of sequential events on the ground truth for *artificial scenarios* and $\approx$ 79.7% for *natural scenarios*.

If we separate *SFAs* from *MFAs* within these videos, V2S+ is able to correctly match $\approx$ 99.2% of the sequential events in the ground truth for *SFAs* and $\approx$ 97.9% of the sequential events for *MFAs*. According to our analysis, these results are higher than those obtained when aggregating these actions (reported above) for two main reasons. First, considering *MFAs* or *SFAs* separately shortens the ground-truth action sequences, making the LCS metric more likely to be higher. Second, the videos included in this study were only able to depict `Two-Fingered Gestures` (as gestures involving more fingers do not produce meaningful application behavior), which also makes the *MFA* sequences achieve a higher LCS.

Precision and Recall. Fig. 14 details the precision and recall results for the *Multi-Touch Study*. The results indicate that on average, the precision of the event traces is $\approx$ 91.9% for multi-touch scenarios. Fig. 14 also reveals that the recall across action types is very high with an average of $\approx$ 99% for all types of events in multi-touch scenarios. On average, V2S+ achieves $\approx$ 93.9% precision and 100% recall for *artificial scenarios*, and the $\approx$ 89.9% precision and $\approx$ 97.9% recall for *natural scenarios*. If we separate the actions depicted into these videos for *SFAs* and *MFAs*, we find that, for *SFAs*, precision is $\approx 83\%$ and recall is 100%. For *MFAs*, precision is $\approx 97.9\%$ and recall is $\approx 97.9\%$.

Success Rate. Finally, we also evaluated success rate of each replayed action in our *Multi-Touch Study*. After manually validating the 12 videos for this study, we found that, on average, V2S+ fully reproduces 83.3% of the scenarios, and 94.8% of the consecutive actions. For *artificial scenarios*, V2S+ is able to reproduce 83.3% of the scenarios and 95.7% of the consecutive actions. For *natural scenarios*, V2S+ is able to reproduce 83.3% of the scenarios and 94.5% of sequential actions. Detailed results are shown in Table 6, where each app, scenario (with total number of actions), and successfully replayed actions are displayed. Because this metric is based on the progression of the depicted scenario as a whole, we were unable to separate *SFAs* from *MFAs* to compute the

TABLE 6: Detailed Results for RQ_4 multi-touch scenarios study. Green cells indicate fully reproduced videos, Orange cells >50% reproduced, and Red Cells<50% reproduced. Blue cells show non-reproduced videos due to non-determinism/dynamic content or `apk` issues.

AppName	Art. Scen.	Nat. Scen.
Infinite Painter	5/5	19/19
Sketchbook	5/5	19/19
Google Maps	4/5	15/15
	4/4	3/8
Waze	3/3	13/13
	2/2	17/17

metrics.

> **Answer to RQ_4**: V2S+ is capable of generating event traces involving multi-fingered actions that require on average < 1 change to match original user scenarios. Moreover, at least 88% of events match the ground truth, when considering the sequence of event types. Overall, precision and recall are $\approx$ 92% and $\approx$ 99% respectively for all event types reproduced by V2S+. V2S+ successfully reproduces $\approx$ 83.3% of multi-fingered scenarios.

6.5 RQ_5: Approach Performance

To measure the performance of V2S+, we measured the average time in seconds/frame (s/f) for a single video frame to be processed across all recorded videos for three components: (i) the frame extraction process (0.045 s/f), (ii) the touch indicator detection process (1.09 s/f), and (iii) the opacity classification process (0.032 s/f). The script generation time is negligible compared to these other processing times, and is only performed once per video. This means that an average video around 3 mins in length[4] would take V2S+ $\approx$105 minutes to fully process and generate the script. However, this process is *fully automated*, can run in the background, and can be accelerated by more advanced hardware. We expect the overhead of our replayed scripts to be similar or better than RERAN since V2S+ replay engine is essentially an improved version of RERAN's.

> **Answer to RQ_5**: V2S+ is capable of fully processing an average 3-min screen recording in $\approx$105 mins.

6.6 RQ_6: Perceived Usefulness

We discuss the results of the user study we conducted to assess V2S+'s perceived accuracy and usefulness.

First of all, the three industry participants (strongly) agreed that the scenarios produced by V2S+ (in the generated scripts) are accurate with respect to the scenarios that were manually executed.

The participants also agreed that further tool support is needed for helping QA members and other stakeholders with generating video recordings. For example, P3 mentions that while videos are "more useful than images" (in some cases), they "may be difficult to record" because of "time constraints". P1 remarked that the QA team could use V2S+ to help them create videos more optimally. P2 supported this claim as he mentions that V2S+ could help "the QA team write/provide commands or steps, then the tool would read and execute these while recording a video of the scenario and problem. This solution could be integrated in the continuous integration pipeline". In addition, P3 mentions that V2S+ could be used during app demos: V2S+ could "automatically execute a script that shows the app functionality and record a video. In this way, the demo would focus on business explanation rather than on manually providing input to the app or execute a user scenario".

P2 also indicated that V2S+ could be used to analyze user behavior within the app, which can help improve certain app screens and navigation. He mentions that V2S+ "could collect the type of interactions, # of taps, *etc.* to detect, for example, if certain screens are frequently used or if users often go back after they navigate to a particular screen". He mentions that this data "could be useful for marketing purposes". P3 finds V2S+ potentially useful for helping reproduce hard-to-replicate bugs.

The participants provided valuable and specific feedback for improving V2S+. They suggested to enrich the videos produced when executing V2S+'s script with a bounding box of the GUI components or screen areas being interacted with at each step. They also mention that the video could show (popup) comments that explain what is going on in the app (*e.g.,* a comment such as "after 10 seconds, button X throws an error"), which can help replicate bugs. They would like to see additional information in the script, such as GUI metadata that provides more in-depth and easy-to-read information about each step. For example, the script could use the names or IDs of the GUI components being interacted with and produce steps such as "the user tapped on the send button" instead of "the user tapped at (10.111,34.56)". P3 mentioned that "it would be nice to change the script programmatically by using the GUI components' metadata instead of coordinates, so the script is easier to maintain". They suggest to include an interpreter of commands/steps, written in a simple and easy-to-write language, which would be translated into low-level commands.

> **Answer to RQ_6**: Developers find V2S+ accurate in replicating app usage scenarios from input videos, and potentially useful for supporting several development tasks, including automatically replicating bugs, analyzing usage app behavior, helping users/QA members generate high-quality videos, and automating scenario executions.

7 Related Work

We briefly discuss prior research work related to V2S+.

7.1 Analysis of video and screen captures.

Lin *et al.* [64] proposed an approach called Screenmilker to automatically extract screenshots of sensitive information (*e.g.,* user entering a password) by using the Android Debug Bridge. This technique focuses on the extraction of keyboard

4. This is a duration longer than most of the scenarios recorded for our evaluation.

inputs from "real-time" screenshots. Screenmilker is primarily focused upon extracting sensitive information, whereas V2S+ analyzes every single frame of a video to generate a high fidelity replay script from a sequence of video frames.

Krieter *et al.* [65] use video analysis to extract high-level descriptions of events from user video recordings on Android apps. Their approach generates log files that describe what events are happening at the app level. Compared to our work, this technique is not able to produce a script that would automatically replay the actions on a device, but instead simply describe high-level app events (*e.g.*, *"WhatsApp chat list closed"*). Moreover, our work focuses on video analysis to help with bug reproduction and generation of test scenarios, rather than describing usage scenarios at a high level.

Bao *et al.* [66] and Frisson *et al.* [67] focus on the extraction of user interactions to facilitate behavioral analysis of developers during programming tasks using CV techniques. In our work, rather than focusing upon recording developers interactions, we instead focus on understanding and extracting generic user actions on mobile apps in order to generate high-fidelity replay scripts.

Other researchers have proposed approaches that focus on the generation of source code for Android applications from screenshots or mock-ups. These approaches rely on techniques that vary solely from CV-based [68] to DL-based [47], [69], [70].

The most related work to V2S+ is the AppFlow approach introduced by Hu *et al.* [71]. AppFlow leverages machine learning techniques to analyze Android screens and categorize types of test cases that could be performed on them (*i.e.*, a sign in screen whose test case would be a user attempting to sign in). However, this technique is focused on the generation of semantically meaningful test cases in conjunction with automated dynamic analysis. In contrast, V2S+ is focused upon the automated replication of any type of user interaction on an Android device, whether this depicts a usage scenario or bug. Thus, V2S+ could be applied to automatically reproduce crowdsourced mobile app videos, whereas AppFlow is primarily concerned with the generation of tests rather than the reproduction of existing scenarios.

Other work has focused on detecting duplicate video-based bug reports. For example, Cooper *et al.* [72] proposed an approach that processes each new bug report that is submitted and automatically creates a list of the top 5 most similar entries, marking the report in the issue tracker as a likely duplicate. Similar to V2S+, TANGO utilizes popular CV techniques to analyze the video inputs. However, TANGO also relies on the textual element of submitted bug reports to draw conclusions in a way that V2S+ does not, using text retrieval and character recognition tools to analyze this additional information. Kordopatis-Zilos *et al.* [73] introduced an approach that utilizes CNNs to extract near-identical videos that could be included in bug reports. This framework makes use of three distinct deep networks and two feature aggregation techniques to extract and make sense of visual features and collect metadata that can then be used to compare videos. Much like V2S+, this approach utilizes CNN frameworks to turn visual information into data that can be processed and composed in the context of the problem at hand. However, neither of the above approaches accomplish the same action classification or replay techniques implemented in V2S+.

7.2 Record and replay.

Many tools assist in recording and replaying tests for mobile platforms [52], [74], [75], [76], [77]. However, many of these tools require the recording of low-level events using `adb`, which usually requires rooting of a device, or loading a custom operating system (OS) to capture user actions/events that are otherwise not available through standard tools such as `adb`. While our approach uses RERAN [52] to replay system-level events, we rely on video frames to transform touch overlays to low-level events. This facilitates bug reporting for users by minimizing the requirement of specialized programs to record and replay user scenarios.

Hu *et al.* [78] developed VALERA for replaying device actions, sensor and network inputs (*e.g.*, GPS, accelerometer, etc.), event schedules, and inter-app communication. This approach requires a rooted target device and the installation of a modified Android runtime environment. These requirements may lead to practical limitations, such as device configuration overhead and the potential security concerns of rooted devices. Such requirements are often undesirable for developers [79]. Conversely, our approach is able to work on any unmodified Android version without the necessity of a rooted device, requiring just a screen recording.

Nurmuradov *et al.* [80] introduced a record and replay tool for Android applications that captures user interactions by displaying the device screen in a web browser. This technique uses event data captured during the recording process to generate a heatmap that facilitate developers' understanding on how users are interacting with an application. This approach is limited in that users must interact with a virtual Android device through a web application, which could result in unnatural usage patterns. This technique is more focused towards session-based usability testing, whereas V2S+ is focused upon replaying "in-field" app usages from users or crowdsourced testers collected from real devices via screen recordings.

Tuovenen *et al.* [81] applied record and replay techniques to Android game application testing. They present MAuto, a tool that records screenshots to summarize user application interactions in the field, creates a test script for the objects and events that take place in the user interaction, and then replays the test. This allows mobile developers to improve their test suites without manually having to define or run these test cases. There are many similarities between MAuto and V2S+, including the general execution flow of each pipeline. However, V2S+ is an end-to-end tool, while MAuto relies on AKAZE features to accomplish image-based object detection on the input screenshots, and then relies on the Appium framework to replay the events. Additionally, V2S+ requires less cooperation from the user by only requiring them to submit a screen-recording with the touch-indicator enabled on the device to show user taps.

Other work has focused on capturing high-level interactions in order to replay events [74], [82], [83], [84]. For instance Mosaic [82], uses an intermediate representation of user interactions to make replays device agnostic. Additional tools including HiroMacro [85] and Barista [60] are

Android applications that allow for a user to record and replay interactions. They require the installation or inclusion of underlying frameworks such as `replaykit` [75], AirTest [74], or `troyd` [76]. Android Bot Maker [86] is an Android application that allows for the automation of individual device actions, however, it does not allow for recording high-level interactions, instead one must enter manually the type of action and raw (x, y) coordinates. In contrast to these techniques, one of V2S+'s primary aims is to create an Android record and replay solution which an inherently low barrier to usage. For instance, there are no frameworks to install, or instrumentation to add, the only input is a collectable screen recording. This makes V2S+ suitable for use in crowd- or beta-testing scenarios, and improves the likelihood of its adoption among developers for automated testing, given its ease of use relative to developer's perceptions of other tools [87].

Finally, as crowdsourcing information from mobile app users has become more common with the advent of a number of testing services [88], [89], [90], researchers have turned to utilizing usage data recorded from crowd-testers to enhance techniques related to automated test case generation for mobile apps. Linares-Vásquez *et al.* first introduced the notion of recording crowd-sourced data to enhance input generation strategies for mobile testing through the introduction of the MONKEYLAB framework [91]. This technique adapted N-gram language models trained on recorded user data to predict event sequences for automated mobile test case generation. Mao *et al.* developed the POLARIZ approach which is able to infer "motif" event sequences collected from the crowd that are applicable across different apps [9]. The authors found that the activity-based coverage of the SAPIENZ automated testing tool can be improved through a combination of crowd-based and search-based techniques. The two approaches discussed above require either instrumented or rooted devices (MONKEYLAB), or interaction with apps via a web-browser (POLARIZ). Thus, V2S+ is complementary to these techniques as it provides a frictionless mechanism by which developers can collect user data in the form of videos.

7.3 Hybrid application testing.

Application testing is a crucial step in the mobile development process. Despite hybrid applications becoming more prevalent in the Google Play store [25], testing tools and techniques often ignore hybrid apps due to the challenges that they pose. Wan *et al.* [92] discuss that because hybrid applications are built using web standards rather than traditional Android UI elements, app crawlers are not able to effectively test a hybrid application's entire functionality. Thus, other testing methods which rely on visual information can help to augment app crawling techniques. V2S+ provides an alternative means of effectively generating test scenarios for hybrid apps, using only visual information contained in videos. In the future, scenarios captured using V2S+ could be used to build models of hybrid apps for new types of hybrid app UI crawlers.

Issues related to hybrid apps have also been highlighted when exploring other testing challenges, such as test migration. Talebipour *et al.* [93] explore UI test migration across mobile platforms (*e.g.,* Android or iOS). This paper illustrates that migrating tests between different platforms is more challenging than between different Android devices or versions, due to different UI design languages and icon usage. These challenges are similar to those that hybrid applications face as cross-platform apps are often built using frameworks that rely on web technologies such as React Native. Lin *et al.* [26] develop CRAFTDROID, a technique that transfers testing oracles between applications by tying GUI elements together based on their semantic functionality. This tool is not able to function with hybrid applications because it is unable to interpret the web elements that they use to accomplish functionality.

There has been a limited amount of past work that supports testing hybrid applications. Most notably, Yazdani-BanafsheDaragh *et al.* [94] present *Deep GUI*, a black-box testing framework that is meant to work well with applications that are built using a variety of implementations (*e.g.,* native and hybrid applications). The authors sought to build upon previous testing tools by learning from interactions based on the pixels of an application screen. V2S+ is similar to *Deep GUI* in that it is a cross-platform approach, but it supports a different goal of recording specific user-provided test scenarios as opposed to generating input sequences based on a learned model.

8 LIMITATIONS & THREATS TO VALIDITY

Limitations. Our approach has various limitations that serve as motivation for future work. One current limitation of the FASTER R-CNN implementation our approach utilizes is that it is tied to the screen size of a single device, and thus a separate model must be trained for each screen size to which V2S+ is applied. However, as described in Sec. 3.2, the training data process is fully automated and models can be trained once and used for any device with a given screen size. This limitation could be mitigated by increasing dataset size including all type of screen sizes with a trade-off on the training time. To facilitate the use of our model by the research community, we have released our trained models for the two popular screen sizes of the Nexus 5 and Nexus 6P in our online appendix [28].

Another limitation, which we will address in future work, is that our replayable traces are currently tied to the dimensions of a particular screen, and are not easily human readable. However, combining V2S+ with existing techniques for device-agnostic test case translation [60], and GUI analysis techniques for generating natural language reports [56] could mitigate these limitations.

As discussed in Sec. 6.3, the ability of V2S+ to faithfully replay swipes is also limited by the video frame-rate. During the evaluation, our devices were limited to 30fps, which made it difficult to completely resolve a small subset of gesture actions that were performed very quickly. However, this limitation could be addressed by improved Android hardware or software capable of recording video at or above 60fps, which, in our experience, should be enough to resolve nearly all rapid user gestures.

Finally, V2S+ is also limited in that it performs noticeably better when utilized on a physical device rather than an emulated device. As discussed in Sec. 6.3.1, the unnatural way that users must interact with emulated devices

and varying computer processing speeds produced poor input videos that were not well reproduced by V2S+. We anticipate that conducting emulator screen recordings at a higher fps rate (>30 FPS) or on a machine with a faster or more powerful processor or graphical card might yield better results.

Internal Validity. In our experiments evaluating V2S+, threats to internal validity may arise from our manual validation of the correctness of replayed videos. To mitigate any potential subjectivity or errors, we had two authors manually verify the correctness of the replayed scenarios. Furthermore, we have released all of our experimental data and code [28], to facilitate the reproducibility of our experiments.

Construct Validity. The main threat to construct validity arises from the potential bias in our manual creation of videos for the popular apps studies carried out to answer RQ_3 and RQ_4. It is possible that the author's knowledge of V2S+ influenced the manner in which we recorded videos. To help mitigate this threat, we took care to record videos as naturally as possible (*e.g.*, normal speed, included natural quick gestures). Furthermore, we carried out an orthogonal study in the course of answering RQ_3, where users unfamiliar with V2S+ naturally recorded videos on physical and emulated devices, representing an unbiased set of videos.

Another potential confounding factor concerns the quality of the dataset of screens used to train, test, and evaluate V2S+'s FASTER R-CNN and Opacity CNN. To mitigate this threat, we utilize the REDRAW dataset [47] of screens which have undergone several filtering and quality control mechanisms to ensure a diverse set of real GUIs. One more potential threat concerns our methodology for assessing the utility of V2S+. Our developer interviews only assess the *perceived* usefulness of our technique, determining whether developers actually receive benefit from V2S+ is left for future work.

External Validity. Threats to the generalizability of our conclusions are mainly related to: (i) the number and diversity apps used in our evaluation; (ii) the representativeness of usage scenarios depicted in our experimental videos; and (iii) the generalizability of the responses given by the interviewed developers. To help mitigate the first threat, we performed 2 large-scale studies, 1 with 64 of the top native applications on Google Play mined from 32 categories, and 1 with 14 of the most popular hybrid applications from the Google Play store derived from 7 categories. While performing additional experiments with more applications is ideal, our experimental set of applications represents a reasonably large number of apps with different functionalities, which illustrate the relative applicability of V2S+. To mitigate the second threat, we collected scenarios illustrating bugs, natural apps usages, real crashes, and controlled crashes from 8 participants. In our popular applications studies, we also gathered a variety of scenarios involving combinations of single-fingered and multi-fingered actions in sequence in an attempt to generalize natural app usages. Finally, we do not claim that the feedback we received from developers generalizes broadly across industrial teams. However, the positive feedback and suggestions for future work we received in our interviews illustrate the potential practical usefulness of V2S+.

9 Conclusion & Future Work

V2S+ is an approach for automatically translating video recordings of Android app usages into replayable scenarios. A comprehensive evaluation reveals that V2S+: (i) accurately identifies touch indicators and it is able to differentiate between opacity levels, (ii) is capable of reproducing a high percentage of complete scenarios in *hybrid* and *native applications* involving *single-* and *multi-fingered actions* on *physical* and *emulated devices*, and (iii) is potentially useful to support real developers during a variety of tasks.

By making use of screen recordings, which are notoriously difficult to analyze, V2S+ is the first automated translation tool of its kind. With all of our findings, we demonstrate that V2S+ may prove to be a invaluable tool for improving the rapid debugging processes that are inherent to mobile development. We expect developers may also be heavily inclined to incorporate it in place of other techniques due to V2S+'s low usage barrier, since V2S+ requires only a screen recording as input, and its modular structure, which can be easily manipulated to fit a wide variety of research and development tasks.

Our future work will focus on adding multiple improvements to V2S+, including (i) producing scripts with coordinate-agnostic actions, (ii) generating natural language user scenarios, (iii) improving user experience via behavior analysis, and (iv) facilitating additional maintenance tasks via GUI-based information.

Acknowledgments

This work is supported in part by the NSF CCF-1927679, CCF-1815186, CNS-1815336, CCF-1955837, and CCF-1955853 grants. Any opinions, findings, and conclusions expressed herein are the authors' and do not necessarily reflect those of the sponsors.

References

[1] M. Nayebi, "Eye of the mind: Image processing for social coding," in *Proceedings of the 42nd IEEE/ACM International Conference on Software Engineering (ICSE'20)*, 2020, p. 49–52.
[2] "Watchsend https://new.watchsend.com," 2019.
[3] "Testfairy https://testfairy.com," 2019.
[4] "Instabug https://instabug.com/screen-recording," 2021.
[5] "Bird eats bugs https://birdeatsbug.com," 2021.
[6] R. Schusteritsch, C. Y. Wei, and M. LaRosa, "Towards the perfect infrastructure for usability testing on mobile devices," in *Proceedings of the CHI Extended Abstracts on Human Factors in Computing Systems*, ser. CHI'07, New York, NY, USA, 2007, pp. 1839–1844.
[7] "Mr. tappy mobile usability testing device https://www.mrtappy.com," 2019.
[8] N. Bettenburg, S. Just, A. Schröter, C. Weiss, R. Premraj, and T. Zimmermann, "What makes a good bug report?" in *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE'08, New York, NY, USA, 2008, pp. 308–318.
[9] K. Mao, M. Harman, and Y. Jia, "Crowd intelligence enhances automated mobile testing," in *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE'17, Piscataway, NJ, USA, 2017, pp. 16–26.
[10] K. Moran, C. Bernal-Cárdenas, M. Linares-Vásquez, and D. Poshyvanyk, "Overcoming Language Dichotomies: Toward Effective Program Comprehension for Mobile App Development," in *Proceedings of the 26th International Conference on Program Comprehension*, ser. ICPC'18, 2018, pp. 7–18.

[11] G. Hu, X. Yuan, Y. Tang, and J. Yang, "Efficiently, effectively detecting mobile app bugs with appdoctor," in *Proceedings of the 9th European Conference on Computer Systems*, ser. EuroSys'14, New York, NY, USA, 2014, pp. 18:1–18:15.

[12] N. Jones, "Seven best practices for optimizing mobile testing efforts," Gartner, Technical Report G00248240, 2013.

[13] M. Linares-Vásquez, G. Bavota, C. Bernal-Cárdenas, M. Di Penta, R. Oliveto, and D. Poshyvanyk, "Api change and fault proneness: A threat to the success of android apps," in *Proceedings of the 9th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE'13, New York, NY, USA, 2013, pp. 477–487.

[14] G. Bavota, M. Linares-Vásquez, C. Bernal-Cárdenas, M. Di Penta, R. Oliveto, and D. Poshyvanyk, "The impact of api change- and fault-proneness on the user ratings of android apps," *IEEE Transactions on Software Engineering*, vol. 41, no. 4, pp. 384–407, April 2015.

[15] A. Ciurumelea, A. Schaufelbühl, S. Panichella, and H. Gall, "Analyzing reviews and code of mobile apps for better release planning," in *Proceedings of the IEEE 24th International Conference on Software Analysis, Evolution and Reengineering*, ser. SANER'17, Feb. 2017, pp. 91–102.

[16] A. Di Sorbo, S. Panichella, C. V. Alexandru, J. Shimagaki, C. A. Visaggio, G. Canfora, and H. C. Gall, "What Would Users Change in My App? Summarizing App Reviews for Recommending Software Changes," in *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE'16, Seattle, WA, USA, 2016, pp. 499–510.

[17] F. Palomba, P. Salza, A. Ciurumelea, S. Panichella, H. Gall, F. Ferrucci, and A. De Lucia, "Recommending and localizing change requests for mobile apps based on user reviews," in *Proceedings of the 39th International Conference on Software Engineering*, ser. ICSE'17, Piscataway, NJ, USA, 2017, pp. 106–117.

[18] F. Palomba, M. Linares-Vásquez, G. Bavota, R. Oliveto, M. D. Penta, D. Poshyvanyk, and A. D. Lucia, "Crowdsourcing user reviews to support the evolution of mobile apps," *Journal of Systems and Software*, pp. 143–162, 2018.

[19] F. Palomba, M. Linares-Vasquez, G. Bavota, R. Oliveto, M. D. Penta, D. Poshyvanyk, and A. D. Lucia, "User reviews matter! tracking crowdsourced reviews to support evolution of successful apps," in *Proceedings of the IEEE International Conference on Software Maintenance and Evolution*, ser. ICSME'15, Sept 2015, pp. 291–300.

[20] D. Han, C. Zhang, X. Fan, A. Hindle, K. Wong, and E. Stroulia, "Understanding android fragmentation with topic analysis of vendor-specific bugs," in *Proceedings of the 19th Working Conference on Reverse Engineering*, ser. WCRE'12, Washington, DC, USA, 2012, pp. 83–92.

[21] L. Wei, Y. Liu, and S. C. Cheung, "Taming Android fragmentation: Characterizing and detecting compatibility issues for Android apps," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE'16, Sep. 2016, pp. 226–237.

[22] "Android fragmentation statistics http://opensignal.com/reports/2014/android-fragmentation/," 2014.

[23] M. Linares-Vásquez, K. Moran, and D. Poshyvanyk, "Continuous, Evolutionary and Large-Scale: A New Perspective for Automated Mobile App Testing," in *Proceedings of the IEEE International Conference on Software Maintenance and Evolution*, ser. ICSME'17, 2017, pp. 399–410.

[24] 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 *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering (ICSE'20)*, 2020, pp. 309–321.

[25] I. Malavolta, S. Ruberto, T. Soru, and V. Terragni, "Hybrid mobile apps in the google play store: An exploratory investigation," 08 2015.

[26] J.-W. Lin, R. Jabbarvand, and S. Malek, "Test transfer across mobile apps through semantic mapping," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2019, pp. 42–53.

[27] "Monkey https://developer.android.com/studio/test/other-testing-tools/monkey," 2022.

[28] C. Bernal-Cárdenas, N. Cooper, M. Havranek, K. Moran, O. Chaparro, A. Marcus, and D. Poshyvanyk, "V2s online appendix https://sites.google.com/email.wm.edu/video2scenario/home," 2022.

[29] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Proceedings of the 25th Neural Information Processing Systems*, ser. NeurIPS'12, 2012, pp. 1097–1105.

[30] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," in *Proceedings of the International Conference on Learning Representations*, ser. ICLR'14, vol. abs/1409.1556, 2014.

[31] M. D. Zeiler and R. Fergus, "Visualizing and understanding convolutional networks," in *Proceedings of the 13th European Conference on Computer Vision*, ser. ECCV'14, Cham, 2014, pp. 818–833.

[32] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *Proceedings of the Computer Vision and Pattern Recognition*, ser. CVPR'15, 2015.

[33] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, ser. CVPR'16, June 2016, pp. 770–778.

[34] M. Tan and Q. Le, "EfficientNet: Rethinking model scaling for convolutional neural networks," in *Proceedings of the 36th International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, K. Chaudhuri and R. Salakhutdinov, Eds., vol. 97. PMLR, 09–15 Jun 2019, pp. 6105–6114. [Online]. Available: https://proceedings.mlr.press/v97/tan19a.html

[35] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," in *Proceedings of the 31st International Conference on Neural Information Processing Systems*, ser. NIPS'17. Red Hook, NY, USA: Curran Associates Inc., 2017, p. 6000–6010.

[36] B. Wu, C. Xu, X. Dai, A. Wan, P. Zhang, M. Tomizuka, K. Keutzer, and P. Vajda, "Visual transformers: Token-based image representation and processing for computer vision," *CoRR*, vol. abs/2006.03677, 2020. [Online]. Available: https://arxiv.org/abs/2006.03677

[37] I. O. Tolstikhin, N. Houlsby, A. Kolesnikov, L. Beyer, X. Zhai, T. Unterthiner, J. Yung, A. Steiner, D. Keysers, J. Uszkoreit, M. Lucic, and A. Dosovitskiy, "Mlp-mixer: An all-mlp architecture for vision," *CoRR*, vol. abs/2105.01601, 2021. [Online]. Available: https://arxiv.org/abs/2105.01601

[38] R. Girshick, J. Donahue, T. Darrell, and J. Malik, "Rich feature hierarchies for accurate object detection and semantic segmentation," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, ser. CVPR'14, Washington, DC, USA, 2014, pp. 580–587.

[39] J. R. R. Uijlings, K. E. A. van de Sande, T. Gevers, and A. W. M. Smeulders, "Selective search for object recognition," *International Journal of Computer Vision*, vol. 104, no. 2, pp. 154–171, Sep 2013.

[40] N. Carion, F. Massa, G. Synnaeve, N. Usunier, A. Kirillov, and S. Zagoruyko, "End-to-end object detection with transformers," *CoRR*, vol. abs/2005.12872, 2020. [Online]. Available: https://arxiv.org/abs/2005.12872

[41] S. Ren, K. He, R. Girshick, and J. Sun, "Faster r-cnn: Towards real-time object detection with region proposal networks," in *Proceedings of the 28th International Conference on Neural Information Processing Systems*, ser. NeurIPS'15, Cambridge, MA, USA, 2015, pp. 91–99.

[42] "Google play screen recording apps https://play.google.com/store/search?q=screen%20recording&c=apps," 2019.

[43] "Android show touches option https://tinyurl.com/show-touches," 2019.

[44] 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.

[45] "Stackoverflow android screen record https://stackoverflow.com/questions/29546743/what-is-the-frame-rate-of-screen-record/44523688," 2019.

[46] "Ffmpeg tool https://www.ffmpeg.org/," 2019.

[47] K. P. Moran, C. Bernal-Cardenas, M. Curcio, R. Bonett, and D. Poshyvanyk, "Machine Learning-Based Prototyping of Graphical User Interfaces for Mobile Apps," *IEEE Transactions on Software Engineering*, 2018.

[48] "Tensorflow object detection api https://github.com/tensorflow/models/tree/master/research/object_detection," 2019.

[49] "Track touch and pointer movements https://developer.android.com/training/gestures/movement," 2019.

[50] "Manage touch events in a viewgroup: Use viewconfiguration constants https://developer.android.com/training/gestures/viewgroup#vc," 2019.

[51] "Viewconfiguration definition android os https://android.googlesource.com/platform/frameworks/base/+/master/core/java/android/view/ViewConfiguration.java," 2006.

[52] L. Gomez, I. Neamtiu, T. Azim, and T. Millstein, "RERAN: Timing- and touch-sensitive record and replay for android," in *Proceedings of the 35th International Conference on Software Engineering*, ser. ICSE'13, 2013, pp. 72–81.

[53] "Multi-touch (mt) protocol https://www.kernel.org/doc/Documentation/input/multi-touch-protocol.txt," 2009.

[54] K. Moran, C. B. Cardenas, M. Curcio, R. Bonett, and D. Poshyvanyk, "The redraw dataset: A set of android screenshots, gui metadata, and labeled images of gui components https://zenodo.org/record/2530277#.YrOFTy-B27M," 2018.

[55] T. Y. Lin, M. Maire, S. Belongie, J. Hays, P. Perona, D. Ramanan, P. Dollár, and C. L. Zitnick, "Microsoft COCO: Common objects in context," pp. 740–755, 2014.

[56] K. Moran, M. Linares-Vásquez, C. Bernal-Cárdenas, C. Vendome, and D. Poshyvanyk, "Automatically discovering, reporting and reproducing android application crashes," in *Proceedings of the IEEE International Conference on Software Testing, Verification and Validation*, ser. ICST'16. IEEE, 2016, pp. 33–44.

[57] K. Moran, M. Linares-Vásquez, C. Bernal-Cárdenas, and D. Poshyvanyk, "Auto-completing bug reports for android applications," in *Proceedings of the 10th Joint Meeting of the European Software Engineering Conference and the 23rd ACM SIGSOFT Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE'15, Bergamo, Italy, Ausgust-September 2015 2015.

[58] 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," in *Proceedings of the 27th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE'19, Bergamo, Italy, Ausgust 2019 2019.

[59] Z. Qin, Y. Tang, E. Novak, and Q. Li, "Mobiplay: A remote execution based record-and-replay tool for mobile applications," in *Proceedings of the 38th International Conference on Software Engineering*, ser. ICSE'16, New York, NY, USA, 2016, pp. 571–582.

[60] M. Fazzini, E. N. D. A. Freitas, S. R. Choudhary, and A. Orso, "Barista: A technique for recording, encoding, and running platform independent android tests," in *Proceedings of the 10th IEEE International Conference on Software Testing, Verification and Validation*, ser. ICST'17, March 2017, pp. 149–160.

[61] "Google search https://play.google.com/store/apps/details?id=com.google.android.googlequicksearchbox," 2019.

[62] "7-eleven https://play.google.com/store/apps/details?id=com.sei.android," 2019.

[63] "Proximus https://www.proximus.be," 2019.

[64] C.-C. Lin, H. Li, X. Zhou, and X. Wang, "Screenmilker: How to milk your android screen for secrets," in *Proceedings of the Network and Distributed System Security Symposium*, ser. NDSS'14, 2014.

[65] P. Krieter and A. Breiter, "Analyzing mobile application usage: generating log files from mobile screen recordings," in *Proceedings of the 20th International Conference on Human-Computer Interaction with Mobile Devices and Services*, ser. MobileHCI'18, 2018, pp. 1–10.

[66] L. Bao, J. Li, Z. Xing, X. Wang, and B. Zhou, "scvripper: video scraping tool for modeling developers' behavior using interaction data," in *Proceedings of the 37th International Conference on Software Engineering*, ser. ICSE'15. IEEE Press, 2015, pp. 673–676.

[67] C. Frisson, S. Malacria, G. Bailly, and T. Dutoit, "Inspectorwidget: A system to analyze users behaviors in their applications," in *Proceedings of the CHI Conference Extended Abstracts on Human Factors in Computing Systems*, ser. CHI'16. ACM, 2016, pp. 1548–1554.

[68] T. A. Nguyen and C. Csallner, "Reverse engineering mobile application user interfaces with remaui," in *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE'15, Washington, DC, USA, 2015, pp. 248–259.

[69] T. Beltramelli, "pix2code: Generating code from a graphical user interface screenshot," in *Proceedings of the ACM Symposium on Engineering Interactive Computing Systems*, ser. SIGCHI'18. ACM, 2018, p. 3.

[70] C. Chen, T. Su, G. Meng, Z. Xing, and Y. Liu, "From ui design image to gui skeleton: a neural machine translator to bootstrap mobile gui implementation," in *Proceedings of the 40th International Conference on Software Engineering*, ser. ICSE'18. ACM, 2018, pp. 665–676.

[71] G. Hu, L. Zhu, and J. Yang, "AppFlow: using machine learning to synthesize robust, reusable UI tests," in *Proceedings of the 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE'18, 2018, pp. 269–282.

[72] N. Cooper, C. Bernal-Cárdenas, O. Chaparro, K. Moran, and D. Poshyvanyk, "It takes two to TANGO: combining visual and textual information for detecting duplicate video-based bug reports," in *43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021*. IEEE, 2021, pp. 957–969. [Online]. Available: https://doi.org/10.1109/ICSE43902.2021.00091

[73] G. Kordopatis-Zilos, S. Papadopoulos, I. Patras, and I. Kompatsiaris, "Near-duplicate video retrieval by aggregating intermediate cnn layers," vol. 10132, 01 2017, pp. 251–263.

[74] "Airtest project," 2019. [Online]. Available: http://airtest.netease.com/

[75] "Command line tools for recording, replaying and mirroring touchscreen events for android: appetizerio/replaykit," 2019, original-date: 2016-10-28T01:10:09Z. [Online]. Available: https://github.com/appetizerio/replaykit

[76] J. Jeon and J. S. Foster, "Troyd: Integration testing for android," p. 7, 2012.

[77] K. Moran, R. Bonett, C. Bernal-Cardenas, B. Otten, D. Park, and D. Poshyvanyk, "On-Device Bug Reporting for Android Applications," in *Proceedings of the IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems*, ser. MOBILESoft'17, 2017, pp. 215–216.

[78] Y. Hu, T. Azim, and I. Neamtiu, "Versatile yet lightweight record-and-replay for android," in *Proceedings of the ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA'15, 2015, pp. 349–366.

[79] W. Lam, Z. Wu, D. Li, W. Wang, H. Zheng, H. Luo, P. Yan, Y. Deng, and T. Xie, "Record and replay for android: are we there yet in industrial cases?" in *Proceedings of the 11th Joint Meeting on Foundations of Software Engineering - 2017*, ser. ESEC/FSE'17, 2017, pp. 854–859.

[80] D. Nurmuradov and R. Bryce, "Caret-HM: recording and replaying android user sessions with heat map generation using UI state clustering," in *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA'17, 2017, pp. 400–403.

[81] J. Tuovenen, M. Oussalah, and P. Kostakos, "Mauto: Automatic mobile game testing tool using image-matching based approach," *Comput Game*, no. 8, pp. 215–239, 2019.

[82] M. Halpern, Y. Zhu, R. Peri, and V. J. Reddi, "Mosaic: cross-platform user-interaction record and replay for the fragmented android ecosystem," in *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software*, ser. ISPASS'15, 2015, pp. 215–224.

[83] D. T. Milano, "Android ViewServer client. contribute to dtmilano/AndroidViewClient development by creating an account on GitHub," 2019. [Online]. Available: https://github.com/dtmilano/AndroidViewClient

[84] (2019) Create UI tests with espresso test recorder. [Online]. Available: https://developer.android.com/studio/test/espresso-test-recorder

[85] (2019) HiroMacro auto-touch macro - apps on google play. [Online]. Available: https://play.google.com/store/apps/details?id=com.prohiro.macro&hl=en

[86] (2019) [ROOT] bot maker for android - apps on google play. [Online]. Available: https://play.google.com/store/apps/details?id=com.frapeti.androidbotmaker_paid&hl=en_US&gl=US

[87] M. Linares-Vásquez, C. Bernal-Cardenas, K. Moran, and D. Poshyvanyk, "How do Developers Test Android Applications?" in *Proceedings of the IEEE International Conference on Software Maintenance and Evolution*, ser. ICSME'17, Sep. 2017, pp. 613–622.

[88] "Applause crowdsourced testing service https://www.applause.com/crowdtesting," 2019.

[89] "Testbirds crowdsourced testing service https://www.testbirds.com," 2019.

[90] "Mycrowd crowdsourced testing service https://mycrowd.com," 2019.

[91] M. Linares-Vásquez, M. White, C. Bernal-Cárdenas, K. Moran, and D. Poshyvanyk, "Mining android app usages for generating

actionable gui-based execution scenarios," in *Proceedings of the 12th Working Conference on Mining Software Repositories*, ser. MSR'15, 2015, pp. 111–122.
[92] M. Wan, N. Abolhassani, A. Alotaibi, and W. G. J. Halfond, "An empirical study of ui implementations in android applications," in *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2019, pp. 65–75.
[93] S. Talebipour, Y. Zhao, L. Dojcilović, C. Li, and N. Medvidović, "Ui test migration across mobile platforms," in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2021, pp. 756–767.
[94] F. YazdaniBanafsheDaragh and S. Malek, "Deep gui: Black-box gui input generation with deep learning," in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2021, pp. 905–916.

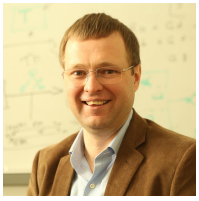
Carlos Bernal-Cárdenas is currently a Software Development Engineer at Microsoft. He received the B.S. degree in systems engineering from the Universidad Nacional de Colombia in 2012 and his M.E. in Systems and Computing Engineering in 2015. He received a Ph.D. degree from William & Mary in August 2021. His research interests include software engineering, software evolution and maintenance, information retrieval, software reuse, mining software repositories, mobile applications development, and user experience. He has published in several top peer-reviewed software engineering venues including: ICSE, ESEC/FSE, ICST, and MSR. He has also received the ACM SIGSOFT Distinguished paper award at ESEC/FSE'15 & '19 and ICSE'20. More information is available at http://www.cs.wm.edu/~cebernal/.

Nathan Cooper is a nerd and received a B.S. degree in Software Engineering from the University of West Florida in 2018. He is currently a Ph.D. candidate in Computer Science at William & Mary under the mentorship of Dr. Denys Poshyvanyk and is a member of the Semeru Research group. He has research interests in Software Engineering, Machine / Deep Learning applications for Software Engineering, information retrieval, and question & answering applications for Software Engineering. He has published in the top peer-reviewed Software Engineering venues ICSE and MSR. He has also received the ACM SIGSOFT Distinguished paper award at ICSE'20. More information is available at https://nathancooper.io/#/.

Madeleine Havranek is an undergraduate student in the Computer Science Department at William & Mary. She is currently a member of the SEMERU research group and is pursuing an undergraduate honors thesis on the topic of automating software testing reproduction. Her research interests lie in applications of deep learning to software engineering and testing tasks. Havranek is an IEEE student member.

Kevin Moran is currently an Assistant Professor at George Mason University where he directs the SAGE research group. He graduated with a B.A. in Physics from the College of the Holy Cross in 2013 and an M.S. degree from William & Mary in August of 2015. He received a Ph.D. degree from William & Mary in August 2018. His main research interest involves facilitating the processes of software engineering, maintenance, and evolution with a focus on mobile platforms. He has published in several top peer-reviewed software engineering venues including: ICSE, ESEC/FSE, TSE, USENIX, ICST, ICSME, and MSR. His research has been recognized with ACM SIGSOFT distinguished paper awards at ESEC/FSE 2019 and ICSE 2020, and a Best Paper Award at CODASPY'19. He is a member of the ACM and IEEE. More information is available at http://www.kpmoran.com

Oscar Chaparro is an Assistant Professor at William & Mary. He received his B.Eng. (2010) and M.Eng. (2013) degrees in Systems Engineering and Computing from the National University of Colombia. He received this Ph.D. (2019) in Software Engineering at the University of Texas at Dallas, advised by Dr. Andrian Marcus. His research interests lie in software maintenance and evolution, program comprehension, code refactoring, code quality, developer's productivity, and text analysis applied to software engineering. Oscar has authored several publications in top software engineering venues, such as ICSE, ESEC/FSE, and ICSME. He received ACM SIGSOFT Distinguished Paper Awards at ESEC/FSE'19 and ICSE'20, and the IEEE TCSE Distinguished Paper Award at ICSME'17. Oscar has served on the organizing and program committees of several conferences and workshops, including ASE, ICSME, MSR, ICPC, and the DySDoc3 and DocGen2 workshops. Oscar is a member of the IEEE and ACM. More information is available at: https://ojcchar.github.io/



Denys Poshyvanyk is a Professor of Computer Science at William & Mary. He received the MS and MA degrees in Computer Science from the National University of Kyiv-Mohyla Academy, Ukraine, and Wayne State University in 2003 and 2006, respectively. He received the PhD degree in Computer Science from Wayne State University in 2008. He served as a program co-chair for ASE'21, MobileSoft'19, ICSME'16, ICPC'13, WCRE'12 and WCRE'11. He currently serves on the editorial board of IEEE Transactions on Software Engineering (TSE), Empirical Software Engineering Journal (EMSE, Springer), Journal of Software: Evolution and Process (JSEP, Wiley) and Science of Computer Programming. His research interests include software engineering, software maintenance and evolution, program comprehension, reverse engineering and software repository mining. His research papers received several Best Paper Awards at ICPC'06, ICPC'07, ICSM'10, SCAM'10, ICSM'13, CODAPSY'19 and ACM SIGSOFT Distinguished Paper Awards at ASE'13, ICSE'15, ESEC/FSE'15, ICPC'16, ASE'17, ESEC/FSE'19 and ICSE'20. He also received the Most Influential Paper Awards at ICSME'16, ICPC'17, ICPC'20 and ICSME'21. He is a recipient of the NSF CAREER award (2013). He is a senior member of the IEEE and ACM. More information is available at: http://www.cs.wm.edu/~denys/

Andrian Marcus is a Professor of Computer Science and Software Engineering at the University of Texas at Dallas. He obtained his Ph.D. in Computer Science from Kent State University (US), and has prior degrees in Computer Science and European Studies from The University of Memphis (US) and Babes-Bolyai University (Cluj-Napoca, Romania). His research interests are in software engineering, with focus on program understanding and software evolution. He is a former Fulbright Scholar and the recipient of the NSF CAREER award. The research with his students and collaborators earned six Best/Distinguished Paper Awards and six Most Influential Paper Awards at software engineering conferences. His professional service includes serving on the Steering Committees of ICSME and VISSOFT. He was the General Chair and the Program Co-chair of ICSME'11 ICSME'10, respectively, and Program Co-Chair for other conferences (ICPC'09, VISSOFT'13, SANER'17). He served on the editorial boards of the Empirical Software Engineering Journal, the Journal of Software: Evolution and Process, and IEEE Transactions on Software Engineering. More information is available at: https://personal.utdallas.edu/~amarcus/