Android STAR: An Efficient Interaction-Preserving Record-Replay System For Messenger App Usage Surveillance

Lun-Pin Yuan  
Penn State University  
lunpin@psu.edu

Peng Liu  
Penn State University  
pliu@ist.psu.edu

Sencun Zhu  
Penn State University  
szhu@cse.psu.edu

ABSTRACT
Messenger apps on smart phones are widely used for easy communication in a collaborative workplace. However, the use of messengers increases risks to both the organization and the collaborators. For example, an employee may receive proprietary information from one app and then accidentally leak it with another app, but neither the employer nor the employee can effectively prove or disprove what has happened inside messengers. To prove mental elements in a lawsuit, the capability of inspecting the use of messengers in a workplace is desirable to both parties: one can prove misconduct and the other can prove innocence. Yet, guilty intention is subtle if not literally described, and how to prove whether there was a guilty intention has not yet been resolved. To provide new kind of evidence, we propose Android STAR, an inspection-purposed record-and-replay service that replays conversation histories and user interactions with apps. We assume that the employer has obtained consents of employees, and the employees have installed Android STAR in their company devices. The challenge to app-usage inspection includes app variety and evidence veracity. We evaluate STAR with 10 popular messenger apps (including Telegram, LINE, and WeChat). Our results show that while STAR can replay in high-fidelity, it only introduces small performance overhead.

CCS CONCEPTS
• Applied computing → Surveillance mechanisms;

KEYWORDS
Android; Messenger Inspection; Communication Surveillance

ACM Reference Format:

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

WiSec ’18, June 18–20, 2018, Stockholm, Sweden  
© 2018 Association for Computing Machinery.  
ACM ISBN 978-1-4503-5731-9/18/06 ... $15.00  
https://doi.org/10.1145/3212480.3212503

1 INTRODUCTION
Nowadays messenger apps on smart phones are widely used for easy communication in a collaborative workplace. While some people embrace the use [6, 37, 50], some questioned the possible impacts of using them [11, 25, 26, 54]. Using messengers also increases risks to both the organization and the collaborators [5, 12, 15]. Take data leakage as an example, an employee may receive proprietary information from one app and then accidentally leak it with another app, but neither the employer nor the employee can effectively prove what has happened inside messengers. In a lawsuit, the capability of inspecting the use of messengers in a workplace is desirable to both parties: one can prove misconduct and the other can prove innocence. In common legal practice, employers are allowed to monitor employees communication on company devices with prior consents [13, 16, 17]. However, to prove “mens rea”, we need not only the conversation but also the user interaction with messengers, because from interaction we may be able to perceive user’s intention. Yet, app-usage inspection (or surveillance) has not been addressed nor proposed. To this end, we propose Android STAR, an Android Service That Archives Renderings (Fig. 1). Android STAR is a lightweight inspection-purposed record-and-replay service which allows authorized inspectors (e.g., employer and employee) to replay conversation histories and user interactions with apps. We aim to provide new types of replay evidence, nonetheless it is not limited to workplace scenarios.

The challenge of app-usage inspection includes high app variety (most are not open-sourced), app velocity (new apps and updates are released frequently), evidence veracity (e.g., did the user read through all the messages?), and the performance overhead of enabling inspection. Rather than inspection, more work focus on content-recovery forensics (e.g., [2, 7–9, 14, 18, 21, 23, 24, 27, 30–34, 36, 39, 44–48, 51, 55, 56]) or system-level monitoring (e.g., mobile spyware products listed in [19, 52]). They, however, each has certain limitations regarding to conversation history or interaction recovery. On one hand, conversation histories (texts and images) intuitively are the most crucial evidence that should be carefully inspected. Although histories can commonly be found in the device and in server databases, it is not true for some instant messengers that take user privacy very seriously. Telegram secret-chat, for instance, deletes all messages once the user has seen them; furthermore, the conversation is encrypted in an end-to-end fashion so the plaintext is not obtainable elsewhere. On the other hand, user interactions may provide crucial clues. For example, constantly revisiting particular dialog or picture could demonstrate intentions if not literally described, scrolling dynamics could prove what has been seen or been scrolled over without being checked, and type-and-wipe progressions could show the thinking of the user.
We design Android STAR very carefully so that it can record-and-replay many apps with high fidelity but small overhead. We assume that the employer has obtained consents of employees, and the employees have installed Android STAR in their company devices. Android STAR, in a nutshell (Fig. 1), records graphical events through dedicated hooks, schedule-and-composes the audit logs, and then replays the audit logs record-by-record based on time-of-rendering. Since Android STAR examines graphical events in the Android framework rather than apps, its recorder module can work without prior knowledge of messenger app implementation. Moreover, its replayer module can re-use the graphical APIs. However, the replayer module cannot simply replay a sequence of graphical APIs with parameters due to hierarchical GUI structure (parameters are relative values inside a parent frame), invalidation behaviors (i.e., selective invalidation/redrawing and temporal gap between invalidation and redrawing), and app-specific optimization (i.e., projection, which will be defined in Section 4.3.3). The above issues result in disorganized contents, constant flashes, and absence of images, respectively. With tree reconstruction, defer-invalidation, and projection matching, Android STAR resolves the above issues, and therefore it can present the replay smoothly. Although, in this paper, we lay our STAR design in Android framework, the STAR concepts are also applicable to iOS and Windows.

We evaluate Android STAR with ten popular messenger apps, including Telegram, LINE, and WeChat. Our results show that while Android STAR can replay in high fidelity (over 95% in similarity), it only introduces small performance overhead (freeze-free) and small log size (less than 10MB over 1800 interactions). Moreover, Android STAR can replay Telegram secret-chats because secret messages are decrypted and then displayed via Android graphics pipeline. However, app-specific customization must be made to the replayer module for overly-customized apps, such as mobile games and streaming apps. At current stage, Android STAR focuses on messenger apps and has no per-app customization. Although there are some existing apps that are also capable of replaying screens, none serves the same purpose and thus none fully meets our requirements (defined in Section 3.2). Take efficiency as an example, VNC-based remote control [42, 53] are capable of streaming content, and therefore it can present the replay smoothly. Although, in this paper, we lay our STAR design in Android framework, the STAR concepts are also applicable to iOS and Windows.

In summary, we make the following contributions:

- To the best of our knowledge, Android STAR is the first communication surveillance work that focuses on efficiently record-and-replay the use of messenger apps.
- We proposed multiple novel recording and replaying techniques, and thus Android STAR meets a number of high-level design requirements.
- We implemented Android STAR and evaluated it with 10 popular instant messenger apps. Our results show that while Android STAR can replay in high-fidelity, it only introduces small performance overhead and log size.

In the literature of workplace regulations, the following three terms are interchangeable: surveillance, monitoring, and surveillance. The rest of this paper is organized as follows. Section 2 gives an introduction to Android graphics pipeline, Section 3 states the definitions and requirements, Section 4 details the Android STAR design, Section 5 shows our evaluation results, Section 6 discusses several technical questions, Section 7 addresses legality and privacy issues, Section 8 overviews prior related work, and lastly Section 9 concludes this paper.

## 2 Android Graphics Pipeline

Android occupies more than 68% of mobile and tablet market share by November 2017 [40]. Among them, Android 6 Marshmallow occupies more share than any other versions. Android graphics pipeline is sophisticated, and different Android version have slightly different implementation and optimization in their graphics pipeline. Generally speaking, it works as follows (Fig. 2). An application draws texts and images onto a canvas instance. The canvas will then compose draw-op lists and invoke software (skia) or hardware (hwui) renderer. Whichever the renderer is, the renderer will then render a dedicated buffer in a surface, an interface object used for communication between renderers and the surface flinger. After accepting this surface, the surface flinger will then invoke a hardware composer in HAL. Finally, the hardware composer renders a frame buffer with available hardware, and brings the frame buffer onto the screen by swapping buffers.

At the application level, a tree formed by display components is built in each activity, where a display component can be (but not limited to) a view, a text layout, or an image. To bring objects onto the screen, an app developer can either use Android-provided display components (e.g., Layout and ImageView) or design customized components that overwrite the formers. If, however, an app is not open-sourced, the implementation is unknown unless reverse-engineered. Regardless of implementation, certain Android APIs must be invoked to put display components onto the screen.

Here we give an example of how a component tree is drawn onto a canvas (Fig. 3). The draw of a tree starts from the draw(·) method of the DecorView (view root) of an activity. During this
Figure 3: An example view-and-object tree: when an application draws on canvas, it invokes draw methods of display components one by one in a single thread.

draw(·), DecorView draws its children views and children view-groups by invoking their draw(·) methods regardless how they are designed. Each child view or viewgroup will then draw itself by invoking its developer-defined onDraw(·) and dispatchDraw(·) methods, and in these methods it will invoke children’s draw(·) if any. To draw text or bitmap in a component, the component must invoke designated draw methods, such as canvas.drawText(·) or drawBitmap(·). Throughout the process, the hierarchical structure is important, because some parameters (e.g., x, y-axis, matrix) of these draw methods are relative values inside their parent component. For example, the y-axis value does not indicate the distance between the component and the screen border. Instead, it is the distance between the component and the edge of its parent. In this work, we use the above knowledge to build a view-and-object tree that has an equivalent graphical structure to the original tree.

3 REQUIREMENTS

3.1 Definitions

Android STAR records and replays objects, contexts, and interactions. Formally, the definitions are given as follows.

An audit object is either text or bitmap object on the screen. It can be a text message, a tag, a timestamp, a picture, an avatar, an emoji, or an icon. Note that an audit object is different from its display component object – the latter is the container of the former.

An audit object context, or context, is a set of attributes that defines how an audit object is shown on the screen. For example, matrix, paint, and background together is a context. A matrix is a set of vectors describing location, rotation, and scale. A paint specifies how an object is drawn by color, font, brush-width, and by more. A background is defined by a Drawable, an abstract class that can be extended by app developers. Android framework provides basic Drawable implementations, such as ColorDrawable, PaintDrawable, BitmapDrawable, and NinePatchDrawable.

An audit user interaction, or interaction, is a continuous animation displayed on the screen, including scrolling, zooming, navigating, and typing. Scrolling is an up-down and left-right animation. Zooming is an in-out rescaling animation, most likely inside a photo viewer. Navigating is an animation of the transition from one view (or layout) to another, including highlighting (clicking) and the sliding-in animation. Typing is a type-and-wipe progression animation.

3.2 Requirements and Challenges

Our goal is to design a service that enables authorized inspectors to see whatever shown on the screen during the use of certain apps. Here we list our requirements.

Message Preserving: The challenge is how we can extract raw message contents from applications that are not open-sourced. If we do not know the implementation, we may not know how the messages are constructed and presented at run-time. To preserve messages, Android STAR monitors certain classes and methods through dedicated hooks placed in Android graphics pipeline.

Order and Origin Preserving: Timestamps and tags cannot help with message order and origin, because they are standalone texts, and because time of rendering does not reflect message ordering. To preserve order and origin, Android STAR records contexts as well. Since contexts are relative knowledge to their parents, Android STAR also archives hierarchical relations between components.

Interaction Preserving: An interaction is nothing more than a series of audit objects and their changes in contexts over time. A frame-to-frame transition may have tens of thousands of field changed in contexts, and an interaction animation may have tens of frames per second. The challenge here is how to present them in a spatially efficient way, so that the presentation will not use up memory. To efficiently and effectively preserve interactions, we design a special timestamped presentation for objects and contexts.

Eye-friendly: The goal is to replay the logs smoothly at the right speed, as if it was recorded as a video. The challenge, however, is not only how to re-establish the view-and-object tree, but also how to differentiate between animation and duplication: when one image instance is drawn at two locations, it can be either an animation of this image moving from one location to another, or two duplicates co-exist on the screen. To tell the difference, Android STAR archives and replays invalidation behaviors.

Runtime Efficient: The solution must be computationally efficient. Computational efficiency is a challenge when there are images on the screen. Archiving multiple large photos or moving (animated) pictures at real-time may cause short freezes to the system if not carefully handled. To cope with this issue, Android STAR schedules the recording work for bitmaps after rendering them.

Recording Generic: The solution should work in a target device without the need of updates for different applications or for different versions. Android STAR is cross-app and cross-version generic in recording (but not in replaying).

4 ANDROID STAR DESIGN

4.1 STAR Architecture Overview

Fig. 1 overviews Android STAR: STAR Recorder composes STAR Log whenever the user uses certain messenger apps. Upon receiving STAR Log, STAR Replayer replays the use.

STAR Recorder (Fig. 4) is a service module integrated with Android graphics pipeline. Its job is to quietly archive the renderings and later send out the STAR Log. It consists of three components: a real-time composer, a bitmap handler, and a root service. The real-time composer composes STAR records (unit record in STAR Log).
according to objects and contexts (defined in Section 3.1) in a way that messages can be reconstructed on the Repayer side. The bitmap handler handles the archiving work of raw bitmaps when the system is computationally convenient, so that the system will not suffer from lag or freeze due to multiple large pictures or animated pictures (regarding why we cannot copy pictures from disk is discussed in Section 4.4). Lastly, the root service gathers and compresses STAR records from multiple apps and send out STAR Logs periodically.

STAR Replayer (Fig. 5), on the other hand, is a stand-alone application accessible to the authorized inspectors. Its job is to replay a given STAR Log record-by-record. It leverages three replaying techniques: tree reconstruction, defer-invalidation, and projection matching. With tree reconstruction, Repayer can render its local canvas with full hierarchical graphical contexts (parameters such as $x$, $y$-axis, and matrix) and therefore it can replay screenshots. With defer-invalidation mechanism, Repayer can differentiate animation and duplication, and therefore it can replay smoothly as if the STAR Log was a video. Lastly, with projection matching, Repayer can correctly replay image components that were not in a view-and-object tree.

4.2 STAR Log Design

The design goal of STAR Log is to pass sufficient information from Recorder to Repayer with only little space usage. A STAR log is a set of tables (DIP, Text, Paint, Drawable, and Bitmap), each of which consists of a sequence of STAR records. A STAR record is a unit record, which takes the format of (type, id, context). In our implementation, there are 21 types.

In DIP stores a chronological series of Draws, Invalidation, and Projections records. A sequence of DIP records describes how an application draws its activities. Table 1 demonstrates an example.
parent view for this component through $T_1$, creates or finds an instance for this component through $T_2$, updates the component instance through $T_3$, and finally connects the component instance with its parent through $T_4$.

Algorithm 2 Tree Reconstruction Baby Step

**Require:** a constructing tree with a view root

**Require:** a draw record = (type, id, context)

$T_1$: parent ← determine a parent for (type, id)

$T_2$: child ← find or create an instance for (type, id)

$T_3$: update child instance with context

$T_4$: connect child with parent in tree

In $T_1$, Parent-child view-object relationship in the tree is not as simple to obtain as view-view relationship: while views have Android API methods for getting view parents, text layouts and bitmaps have no such methods. Instead, Replayer acquires parent-child view-object relations by examining the chronological order and encapsulations of draw records. In Replayer, we implement a simple view stack to track nested currently-drawing views: if Replayer sees a ViewDraw record, it pushes the view into the stack, and if it sees an EndViewDraw record, it pops the view out of the stack. When processing a display object (e.g., LayoutDraw), the view on top of the stack is chosen to be the parent of the object.

Throughout $T_2$ to $T_4$, Replayer finds the component instance in our component pool (by id), updates the component in the pool, and reconnect the component to its parent. We have two reasons for not finding and updating components in our established view-and-object tree. First, based on different app implementation, views may be put under different parents at different time (e.g., view migrates due to re-layout). If we try to find a view in the view after the view is moved away, we may not find the view under the previous parent and thus mistakenly result in instatianting a new view. The second reason is more complicated. In practice, icons (e.g., the “checked” icon) may share the same instance but are drawn at different places (under multiple parents) with different matrices (e.g., location and size). To this end, Replayer handles objects by updating them in the pool and copying the clones to the parent view that is found in $T_1$. If the object is not cloned, then updating one object will result in updating the rest (they share instance), then these objects will share the same matrix (same location and same size). If we search and update the clone in the tree rather than clone from the pool, it will have incorrect contexts because we record only differed fields and the clone does not have previous updates.

Although cloning objects from the component pool solves the above issues, it introduce another. Without removal indicators, outdated components will remain on the screen. To solve this issue, we replay invalidations as well (section 4.3.2).

4.3.2 Replaying Defer-Invalidations. An invalidation is a complicated operation in Android for marking an area of a view “dirty”, so that the view will be redrawn very soon (e.g., in a few milliseconds). When a view is invalidated, Android graphics pipeline will also invalidate the parent view and the children view according to the marked area. Later, the view will be redrawn, followed by children redraws.

When replay, we consider an invalidation an indicator for object removal. If a view is invalidated but not redrawn, then the view and its children all vanish. This is because the view area is redrawn but the view itself is not. If otherwise a view is invalidated and then redrawn, Replayer examines the invalidated view area and decide which children views and children objects should be removed (if the children are redrawn, the children will be brought back by tree reconstruction mechanism). Although invalidations are removal indicators, the removals must be deferred a little bit during the replay, or else one will observe constant flashing on the screen due to the time gaps between invalidations and the following redraws. To avoid flashing, Replayer sets a timer (40 ms in our implementation) before removing views and objects.

We also consider an invalidation an indicator to differentiate animation and duplication: when two object-draws are observed, Replayer determines by invalidation whether it is the case of an object being moved from one context to another (animation), or it is the case of two identical objects co-exist on the screen (duplication). If an invalidation of a parent view happens between the two children draws, it is the case of animation because the second object-draw is a redraw of the first one (the area is marked “dirty” and then redrawn); otherwise, if there is no invalidation of the parent, it is the case of duplication because the two draws happen in the same draw round of the parent.

We implement defer-invalidation by four subroutines. However, we skip the implementation details due to the page limit.

4.3.3 Replaying Projections. We define a projection as a resulting bitmap instance that is created after an app projects a source bitmap onto a canvas (or a canvas recorder) and then transforms the

<table>
<thead>
<tr>
<th>Table 1: Readable Format of Sample STAR Records from Slack</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Type</strong></td>
</tr>
<tr>
<td><strong>DIP</strong></td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td><strong>EndViewDraw</strong></td>
</tr>
<tr>
<td><strong>Paint</strong></td>
</tr>
<tr>
<td><strong>ViewDraw</strong></td>
</tr>
<tr>
<td><strong>EndViewDraw</strong></td>
</tr>
</tbody>
</table>
canvas into a bitmap. Large pictures are occasionally projected and then brought onto the screen for better performance. The difference between a projection-draw and a normal bitmap-draw is that the latter draws a part of bitmap (specified by a source matrix) onto somewhere on a canvas (specified by a destination matrix), whereas a projection is always projected at (0, 0) coordinate (top-left of a canvas). Rather than directly drawing bitmaps, developers may like to project a very large bitmap onto a smaller canvas so that the app can constantly reuse the smaller bitmap whenever needed. We observe it is used in some messengers (e.g., Telegram).

Projections, however, are not part of the tree traversal nor immediately brought onto the screen. Hence tree reconstruction fails to pinpoint their parents. What is worse, the location vectors are gone, so we cannot just find a parent by location. Nevertheless, we can find a match by size. Projections must be put at somewhere on the screen, and we observe that they are commonly placed in chat bubbles, where chat bubbles are commonly ninepatch (9.png) images (a special image format that enables rescaling without losing quality). At the time of replay, Replayer matches a resized ninepatch with a projection according to the the Hamming distance between their sizes as well as the timestamps. If a proper match is found, Replayer binds them together and will never revoke this relation. We omit the details due to the page limit.

Although there are some other possible matching strategies (e.g., matching views), we find matching ninepatches provides the best replay because of the following observations. First, different from views, ninepatches are resized according to conversation contexts (image included). Second, there are only a few of ninepatches on the screen at a time.

4.4 STAR Recorder Design

STAR Recorder is a service module integrated with Android graphics pipeline (Fig. 4). Its design goal is to compose records efficiently through dedicated hooks without harming usability.

4.4.1 Hook Placement. Where we place hooks determines what information Recorder gets. Placing hooks at a lower level inside Android graphics pipeline (e.g., below Canvas in Fig. 2) has the advantage of being generic since they are not likely to be modified or customized by either users and developers. However, without high-level contextual information, massive unsorted low-level data will pile up and cause spatial inefficiency. Placing hooks at a higher level (e.g., application layer) is efficient but not generic because some hooks might be bypassed due to implementations.

Trying to be efficient and generic, we place hooks in high (views) and intermediate (canvas) levels accordingly to the type of information. For example, for text-objects, we place hooks in high-level text-draw methods (e.g., Layout.drawText()); for bitmap-objects, we place hooks in intermediate-level bitmap-draw methods (e.g., Canvas.drawBitmap()). These placements are sufficient to recording texts and images because of the following two observations. First, text layouts are very well defined and optimized in Android framework and they requires only a few parameters, so developers use text layouts for almost every texts (very few directly invoke Canvas.drawText()). Second, there are more high-level implementation variations on images, but invoking native Canvas bitmap-draw methods is inevitable.

4.4.2 Real-Time Composer. Real-time Composer records everything except bitmaps. It generates real-time STAR records based on the data extracted from dedicated hooks. For spatial efficiency, it composes STAR records only based on the differed fields in components. To do so, it caches previous contexts, compares new contexts, and composes records using formatted templates. To be computationally efficient during the use of apps, some optimization is applied. For example, deleting component-parameter memory when the corresponding component is free or deleted.

4.4.3 Bitmap Handler. We introduce Bitmap Handler to STAR Recorder because recording bitmaps is computationally expensive. Recording multiple large photos and animated pictures in real-time will causes short freezes to the system. As such, Bitmap Handler schedules the archiving work after the bitmaps are rendered.

After being notified by Real-time Composer with a bitmap instance, Bitmap Handler schedules the work and generates a temporary id for Real-time Composer to reference in STAR records. Whenever the system is convenient, it hashes the bitmap and uses the value as the bitmap filename. As a result, not only it saves apps from freezes, but also saves disk usage from duplicated images. However, since bitmaps are scheduled for work in bitmap handler, they will not be freed by garbage collection until the process is done; hence this will introduce some memory overhead.

Note that Recorder always archives bitmaps from Android graphics pipeline rather than directly copying the raw images from disks because of the following reasons. First, bitmaps do not necessarily ever exist on disk (could be loaded by links). Second, if a bitmap is stored on disk, it is not necessarily kept for a long time (could be deleted shortly by apps), so they might be gone when archiving them. Third, even if a bitmap is kept for a long time on disk, its locations (uri) or resource identification number (rid) is not necessarily loaded in the framework (developer-defined readfile), so we do not know where the bitmap resides on disk.

5 EVALUATION

5.1 Implementation and Hook Placement

We evaluate Android STAR on Asus Google Nexus 7 (wifi) running Android Open Source Project (6.0.1_r43, aosp_flo_user-debug). Currently our Recorder uses the xz compression suite, and Replayer has no per-app customization for evaluation.

To obtain texts, we place a hook inside Layout.drawText(), and to obtain bitmaps and ninepatches, we place hooks inside Canvas.drawBitmap() and Canvas.drawPatch(). As for views, we place hooks inside draw(), onDraw(), and dispatchDraw(), specifically at the beginning and right before every return statements. Invalidation methods are also hooked. To identify which view is the DecorView in an activity, we place hooks in Activity.onStart(), Activity.onResume(), and Activity.onRestart(), and Activity.onResume(), so STAR can access DecorView getters.

5.2 Genericity and Overall Accuracy

We evaluate Android STAR upon 10 popular instant messengers (Table 2). The overall accuracy is measured by perceptual hashing (pHash) [58], a hash algorithm that produces fingerprints for multimedia. If two perceptual fingerprints are analogous, two media
TABLE 2: EVALUATION RESULTS ON GENERICITY AND ACCURACY (𝑣: ARCHIVED AND REPLAYED; -: ARCHIVED BUT NOT REPLAYED)

<table>
<thead>
<tr>
<th>App</th>
<th>Downloads</th>
<th>Distance</th>
<th>Similarity</th>
<th>Texts</th>
<th>Photos</th>
<th>Icons</th>
<th>Avatar</th>
<th>Matrix</th>
<th>Paint</th>
<th>Background</th>
<th>Navigate</th>
<th>Scroll</th>
<th>Zoom</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Telegram</td>
<td>100M-500M</td>
<td>6</td>
<td>99.93%</td>
<td>v</td>
<td>v</td>
<td>v</td>
<td>-</td>
<td>v</td>
<td>v</td>
<td>v</td>
<td>v</td>
<td>v</td>
<td>v</td>
<td>v</td>
</tr>
<tr>
<td>GroupMe</td>
<td>10M-50M</td>
<td>12</td>
<td>98.66%</td>
<td>v</td>
<td>v</td>
<td>v</td>
<td>v</td>
<td>v</td>
<td>v</td>
<td>v</td>
<td>v</td>
<td>v</td>
<td>v</td>
<td>v</td>
</tr>
<tr>
<td>FB Messenger</td>
<td>1000M-5000M</td>
<td>0</td>
<td>99.85%</td>
<td>v</td>
<td>v</td>
<td>v</td>
<td>v</td>
<td>v</td>
<td>v</td>
<td>v</td>
<td>v</td>
<td>v</td>
<td>v</td>
<td>v</td>
</tr>
<tr>
<td>Google Hangout</td>
<td>1000M-5000M</td>
<td>6</td>
<td>99.94%</td>
<td>v</td>
<td>v</td>
<td>v</td>
<td>v</td>
<td>v</td>
<td>v</td>
<td>v</td>
<td>v</td>
<td>v</td>
<td>v</td>
<td>v</td>
</tr>
<tr>
<td>Skype</td>
<td>500M-1000M</td>
<td>4</td>
<td>95.82%</td>
<td>v</td>
<td>v</td>
<td>-</td>
<td>v</td>
<td>v</td>
<td>v</td>
<td>v</td>
<td>v</td>
<td>v</td>
<td>v</td>
<td>v</td>
</tr>
<tr>
<td>Slack</td>
<td>1M-5M</td>
<td>2</td>
<td>99.91%</td>
<td>v</td>
<td>v</td>
<td>v</td>
<td>v</td>
<td>v</td>
<td>v</td>
<td>v</td>
<td>v</td>
<td>v</td>
<td>v</td>
<td>v</td>
</tr>
<tr>
<td>BBM</td>
<td>100M-500M</td>
<td>4</td>
<td>99.91%</td>
<td>v</td>
<td>v</td>
<td>v</td>
<td>v</td>
<td>v</td>
<td>v</td>
<td>v</td>
<td>v</td>
<td>v</td>
<td>v</td>
<td>v</td>
</tr>
<tr>
<td>WeChat</td>
<td>100M-500M</td>
<td>16</td>
<td>99.29%</td>
<td>v</td>
<td>v</td>
<td>v</td>
<td>v</td>
<td>v</td>
<td>v</td>
<td>v</td>
<td>v</td>
<td>v</td>
<td>v</td>
<td>v</td>
</tr>
<tr>
<td>Line</td>
<td>500M-1000M</td>
<td>14</td>
<td>96.26%</td>
<td>v</td>
<td>v</td>
<td>v</td>
<td>v</td>
<td>v</td>
<td>v</td>
<td>v</td>
<td>v</td>
<td>v</td>
<td>v</td>
<td>v</td>
</tr>
<tr>
<td>WhatsApp</td>
<td>1000M-5000M</td>
<td>6</td>
<td>99.46%</td>
<td>v</td>
<td>v</td>
<td>v</td>
<td>v</td>
<td>v</td>
<td>v</td>
<td>v</td>
<td>v</td>
<td>v</td>
<td>-</td>
<td>v</td>
</tr>
</tbody>
</table>

Figure 6: Input screenshots for pHash: the upper row are the originals, and the lower row are the replays.

sources are perceptually similar in features. The evaluated replays are appended in Fig. 6.

Although we tried to complicate the dialogs by filling them with texts and images and even links, our evaluation results still show very high scores in similarity (over 95%). With objects and contexts being replayed, STAR is able to meet the message preserving and order and origin preserving requirements. Furthermore, STAR is capable of correctly replaying the logs literally "as video", so it also fulfills the interaction preserving and eye-friendly requirements. Lastly, since STAR Recorder does not need to be updated for different versions or different applications, STAR Recorder is cross-app cross-version generic in recording messengers.

However, Android STAR does have limitations. Since Android STAR focuses on messenger app surveillance in a way that it can present communication-wide conversation and interaction in a lawsuit, it does not take non-messaging apps (e.g., mobile games, streaming apps, and utility apps) into consideration. These apps are usually overly customized in their rendering mechanism, and thus Android STAR will have issues record-replay them without being per-app customized. Furthermore, at current stage, Android STAR does not archive audio and video clips because they are out of rendering scope. Regarding the challenges and possible solution of archiving audio/video clips is discussed in Section 6.

5.3 Computational and Spatial Efficiency

To measure the overheads on the recording side, we implement a test script that generates exactly the same random input for both Android STAR and vanilla Android. Our test script (shown in Fig. 7) tries to simulate human behavior by 1800 random interactions. In typing, the script randomly generates a meaningless ascii-strings no longer than 50 characters.

5.3.1 Computational Efficiency. Table 3 shows the CPU and MEM usage of each app in vanilla Android and in Android STAR. Although STAR does introduce computational overhead to Android, each app runs smoothly as usual, and the device is still very usable. The growth of usage is different based on different app implementation. The growth in CPU usage comes from archiving objects and contexts, and especially from archiving (and hashing) bitmaps. Nevertheless, bitmap handler keeps STAR freeze-free by scheduling the work. The growth in MEM usage mainly comes from caching contexts and caching bitmaps objects (section 4.4).
5.3.2 Spatial Efficiency. The growth of STAR logs (without compression) is linear as shown in Fig. 8. Only the foreground app has growth in STAR Log. Background apps will not have noticeable growth in log size because they are not rendered, and therefore no graphical STAR record is composed. From Table 3, we can see that STAR logs use only a few MB over 1800 interactions after xz compression; that is, less than 5 KB per interaction (Table 4). Since VNC-based solution (with compression) requires 5.6 Mbps data usage [53], we consider our solution better. However, this statistic is approximate because our test script generates deterministic interactions. Interactions in real life may vary.

From Table 3, we can see that DIPs occupy most of the space in STAR Log, and it is DIPs that make logs linear to time without compression; however, it is not DIPs that impacts the compression ratio. As for the others, texts do not occupy much because they are naturally small; bitmaps, drawables, and paints do not occupy much either, because they are likely to repeat and because STAR stores them in hash tables and thus eliminates duplicates. From Table 4, we can see that the original DIP records commonly are about 11 bytes long and each DIP record is about one byte long after compression. At current stage, STAR records are stored in text format rather than binary format for debugging purpose. Regarding whether there is a better alternative to using DIP representation is discussed in Section 6. The disk usage proportion in Table 3 are calculated by proportion = (partial usage/total usage), and the average compressed DIP record size in Table 4 is calculated by size ≤ (compressed log size/record count).

5.4 View-and-Object Tree in the Replayer
The main computational concern on the Replayer side is how large the view-and-object tree will grow. Although the tree originally may only have around 3000 nodes on the Recorder side, we can see from Table 4 that the numbers of tree nodes grow significantly larger in the STAR Logs. The numbers come from labeling views and objects in increasing order throughout the recording process. On the Recorder side, if a display component is deleted and then instantiated again, it counts as two components in a tree. On the Replayer side, however, Replayer does not know when to remove the component from the tree, and thus the tree size never shrinks throughout the replayering process (similarly, Replayer never removes components from the component pool). A component idling for a very long time does not necessarily mean it will not be reused again, as it could be the case that the activity is sent to background. This component could be reused when the app is brought to foreground.
If a view is removed from the tree (or an object is removed from the pool), *Replayer* will encounter problems when the component is redrawn due to the loss of previous graphical contexts. Nevertheless, though not yet implemented, one can temporarily dump inactive display components (or subtrees) onto the disk to save run-time memory (and later recover them from disk whenever used).

### 6 DISCUSSIONS AND FUTURE WORK

#### 6.1 Can STAR be more accurate or efficient?

The answer is affirmative. How we place hooks determines how complete the information *STAR* gets, so it can be more accurate if we place hooks everywhere and dump everything, or it can be more efficient if we omit certain things. Specifically for legal inspection purpose, we prefer spatial efficiency over accuracy, so that we leave network and disk usage unharmed while prolonging the recording duration. We focus on extremely high efficiency while preserving sufficiently high fidelity. Nonetheless, we have an insight that higher efficiency can be achieved when we have more fine-grained recording and replaying mechanism. Our future work includes lowering recording fidelity to achieve higher efficiency, while improving replaying fidelity by extracting certain objects and contexts from certain application packages. Note that the replays throughout this paper are purely based on *STAR Logs* alone.

#### 6.2 Can we reduce DIP size?

Since DIPs occupy the most in *STAR logs*, one may expect smaller overall *STAR Log* size by decreasing DIP size. One alternative is that *Recorder* snapshots the tree states, so that certain DIP subsequences are no longer required in *tree reconstruction*. However, in this alternative, the recorder module must reconstruct the tree using DIPs during runtime (or otherwise require tree implementation knowledge), and the tree may use more space and thus result in bigger log size. We show an expected tree-snapshot size by a rough calculation. In our experiments, *STAR* generates about 2000 records per second on average. Since each record is about 12 bytes long, the total record-generation speed is 24KB per second. While in the alternative, assume we represent each edge of a tree using long, the total record-generation speed is 24KB per second. While not yet implemented, one can temporarily dump inactive display components (or subtrees) onto the disk to save run-time memory (and later recover them from disk whenever used).

<table>
<thead>
<tr>
<th>Interaction Count</th>
<th>Number of Tree Nodes</th>
<th>DIP Record Statistics</th>
<th>Compressed DIP Record Statistics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Telegram</td>
<td>1800</td>
<td>580864</td>
<td>3408086</td>
</tr>
<tr>
<td>GroupMe</td>
<td>1800</td>
<td>2277222</td>
<td>5692156</td>
</tr>
<tr>
<td>FB Messenger</td>
<td>1800</td>
<td>542581</td>
<td>7925213</td>
</tr>
<tr>
<td>Google Hangout</td>
<td>1800</td>
<td>3378944</td>
<td>7524249</td>
</tr>
<tr>
<td>Skype</td>
<td>1800</td>
<td>1314605</td>
<td>5936938</td>
</tr>
<tr>
<td>Slack</td>
<td>1800</td>
<td>665269</td>
<td>3108012</td>
</tr>
<tr>
<td>BBM</td>
<td>1800</td>
<td>1182495</td>
<td>4668170</td>
</tr>
<tr>
<td>WeChat</td>
<td>1800</td>
<td>578867</td>
<td>4782802</td>
</tr>
<tr>
<td>LINE</td>
<td>1800</td>
<td>1060164</td>
<td>6558734</td>
</tr>
<tr>
<td>WhatsApp</td>
<td>1800</td>
<td>914563</td>
<td>5326759</td>
</tr>
</tbody>
</table>

*Table 4: Evaluation Results on Tree Size and DIP Record Statistics*

If we view is removed from the tree (or an object is removed from the pool), *Replayer* will encounter problems when the component is redrawn due to the loss of previous graphical contexts. Nevertheless, though not yet implemented, one can temporarily dump inactive display components (or subtrees) onto the disk to save run-time memory (and later recover them from disk whenever used).

#### 6.3 Is STAR limited to messengers?

*Android STAR* is clearly not limited to messengers although replaying non-messenger apps is not our focus. *STAR* is capable of replaying traditionally designed apps that have predictable contexts and tree-structured rendering behaviors. However, some non-messenger apps have overly customized rendering mechanisms that makes them not replayable to *STAR*. For example, mobile games may directly interact with hardware renderers in order to minimize graphical latency. As such, *STAR* will absolutely fail to archive the renderings. This is when one needs to customize both *STAR Recorder* and *Replayer* for particular apps. For example, on *Recorder* side, one can place additional hooks in renderers for additional graphical information, and on *Replayer* side, one can install specific rendering module or preload application resources.

#### 6.4 Can STAR Archive Audio and Video?

Currently, *STAR* does not archive audio and video clips (similarly, *STAR* cannot archive camera footage). Rather, we focus on a text and image conversation along with interactions. Our goal is to provide contextual interaction evidence that has not been addressed, whereas audio and video evidence can be found on disk in most cases. That said, one can archive audio and video clips from graphics pipeline by inspecting low-level raw audio buffer and raw video buffer that are available inside Android framework. However, the challenge is computational and spatial efficiency, as they are way bigger than bitmaps, so not only will they occupy a lot more space but also will they require a lot more computation. An alternative is simply copying the original multimedia files or multimedia bytestream, so that one no longer needs extra computation, and the clips are likely well-compressed. To do so, *STAR* must have prior knowledge about app implementations.
6.5 Is STAR possible on other platforms?
The key techniques in Android STAR are tree reconstruction and defer-invalidation. The GUI characteristics to which these two techniques correspond are that, in a view system, graphical contexts are hierarchical, and redrawing procedure is initiated after invalidated. If the graphical system of a platform has these two characteristics at high level (low-level implementation varies), then STAR has potential to be deployed on such a platform. Take iOS view hierarchy [3, 4] as an example, its UIViews are hierarchically arranged, and UIViews are invalidated via setNeedsDisplay(); therefore tree reconstruction and defer-invalidation can be used in rebuilding screen footage from iOS. Similarly, in Windows (System.Windows.Forms [38]), Control objects are hierarchically arranged and invalidated via invalidate(), so STAR is conceptually applicable to Windows as well. In these platforms, however, STAR may require additional platform-specific configurations.

7 LEGALITY AND HUMAN RIGHTS ISSUE

Android STAR serves as a legal protection to both the employer and the employees in a workplace. With Android STAR, the employer can prove misconduct, and the employee can prove innocence. However, issues are obvious when monitoring became obligated.

Consider a scenario where an employee is using an employer-provided device which is shipped with inspection software (monitoring app or spyware), one may first concern the legality issue of “What if the employee does not want to be monitored? Can the employee take any legal action against the employer?” Unfortunately, in a workplace such as government or enterprise, warrantless employee-monitoring is allowed in United States [13, 16] as long as the two following conditions have been fulfilled: first, the target device is provided by the employer; second, the monitoring has a reasonable cause (e.g., to see whether an employee violates any regulation such as [41]). In most European countries, however, the employer must notify employees before monitoring the use of the device [29]. Under BYOD scenario, both parties must obtain approval from each other [16, 17]. The above conclude that the employees in fact have little personal privacy in a workplace.

One may question whether the inspection will confront human rights. This issue is very complicated, and different countries have different constitutional practice. Take Quon case [1] as an example, Quon’s personal message on a government-owned pager is obtained by city of Ontario without requesting a search warrant, and the Supreme Court unanimously held that the audit did not violate the Fourth Amendment to the United States Constitution, because the audit had a reasonable cause. However, in Barbulescu case [10, 57], Barbulescu was fired by a Romanian private company because of using Yahoo messaging system, but the European Court of Human Rights holds that there is a violation of Article 8 of the European Convention on Human Rights, based on (but not limited to) the fact that the inspection extend was not clearly defined. If a clear consent had been made, the court result may be reversed.

In this paper, we focus on the technical aspect rather than the legality or human rights aspects. However, we oppose unlawful use of our tool. We assume that the employer and the employee have reached to an agreement and have made clear consents.

8 RELATED WORK

Android Remote Control: Remote control such as [53] and [42] provide VNC-based remote control for Android. To make improvements, Taylor and Pasquale [49] proposed a message accelerator that improves video performance in VNC under high latency conditions. However, there are several differences between STAR and VNC-based services. First, their services provide remote control capability, while STAR does not. Second, the replayed screens in their services are exact, while in STAR there are minor differences. Third, their advantages come with a much higher cost. According to the evaluation result in [53], VNC-based solutions generate about 5.6 Mbps data stream using the best compression codec (Zlib). Since STAR only generates several megabytes in total for 1800 events, clearly STAR is more efficient.

Application Testing and Replaying: To reproduce bugs and crashes, existing record-and-replay approaches for development focus on recording and replaying user-input GUI events (e.g., touches, scrolls, and typings). Qin et al. proposed MobiPlay [43], an event-driven record-and-replay testing framework that is capable of playing remote server-side screens as a video stream. Kaasila et al. proposed Testdroid [28], an online platform for scripted tests. Gomez et al. proposed HERAN [20], which focuses on not only user interactions, but also low-level events (e.g., sensor events). Liu et al. [35] proposed an approach that can convert events into test scripts. Halpern et al. proposed Mosaic [22], a cross-platform and timing-accurate record-and-replay tool for Android. Among the above work, only MobiPlay [43] allows remote screen replay. However, [43] requires a minimum network bandwidth of 300 Mbps.

Android Forensics: Saltaformaggio et al. proposed GUITAR [44], a forensic tool that recovers Android application screenshots from dumped memory images. Grover proposed DroidWatch [21], a server-client-based forensic framework that monitors an Android device in many aspects (e.g., GPS, SMS, and phone status). Saltaformaggio et al. proposed VCR [45], a memory forensic technique that recovers photographic evidence in cameras. Thing et al. [51] proposed an automatic tool that extracts outgoing and incoming messages. Walls et al. proposed DECODER [55], a tool for recovering in-phone information such as logs and address books. Saltaformaggio [47] proposed an approach for uncovering wireless access point information. Apostolopoulos et al. [2] investigated and extracted the user authentication credentials. Investigators can also use Volatility [18] or [23] for manual memory analysis. There are many other forensic-related work, including [7–9, 14, 24, 27, 30–34, 36, 39, 46, 48, 56].

9 CONCLUSION

We proposed Android STAR, an efficient inspection-purposed record-and-replay service that enables authorized inspectors to inspect conversations and interactions inside messengers. Our evaluation shows that, while Android STAR provides high-fidelity replay, it generates small log size and introduces little performance overhead. To the best of our knowledge, Android STAR is the first communication inspection work that focuses on efficiently record-and-replay the use of messenger apps.
ACKNOWLEDGMENTS

Peng Li was supported by ARO W911NF-13-1-0421 (URI), ARO W911NF-15-1-0576, and NSF CNS-1505664. The work of Zhu was supported through NSF CNS-1618684.

REFERENCES


