Screen after Previous Screens: Spatial-Temporal Recreation of Android App Displays from Memory Images

Brendan Saltaformaggio, Rohit Bhatia, Xiangyu Zhang, and Dongyan Xu, ***Purdue University;***
Golden G. Richard III, ***University of New Orleans***

https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/saltaformaggio

This paper is included in the Proceedings of the 25th USENIX Security Symposium

August 10–12, 2016 • Austin, TX

ISBN 978-1-931971-32-4

Open access to the Proceedings of the 25th USENIX Security Symposium is sponsored by USENIX

Screen After Previous Screens: Spatial-Temporal Recreation of Android App Displays from Memory Images

Brendan Saltaformaggio[1], Rohit Bhatia[1], Xiangyu Zhang[1], Dongyan Xu[1], Golden G. Richard III[2]

[1]*Department of Computer Science and CERIAS, Purdue University*
{*bsaltafo, bhatia13, xyzhang, dxu*}*@cs.purdue.edu*

[2]*Department of Computer Science, University of New Orleans*
golden@cs.uno.edu

Abstract

Smartphones are increasingly involved in cyber and real world crime investigations. In this paper, we demonstrate a powerful smartphone memory forensics technique, called RetroScope, which recovers multiple previous screens of an Android app — in the order they were displayed — from the phone's memory image. Different from traditional memory forensics, RetroScope enables *spatial-temporal* forensics, revealing the progression of the phone user's interactions with the app (e.g., a banking transaction, online chat, or document editing session). RetroScope achieves near perfect accuracy in both the recreation and ordering of reconstructed screens. Further, RetroScope is app-agnostic, requiring no knowledge about an app's internal data definitions or rendering logic. RetroScope is inspired by the observations that (1) app-internal data on previous screens exists much longer in memory than the GUI data structures that "package" them and (2) each app is able to perform context-free redrawing of its screens upon command from the Android framework. Based on these, RetroScope employs a novel interleaved re-execution engine to selectively reanimate an app's screen redrawing functionality *from within a memory image*. Our evaluation shows that RetroScope is able to recover full temporally-ordered sets of screens (each with 3 to 11 screens) for a variety of popular apps on a number of different Android devices.

1 Introduction

As smartphones become more pervasive in society, they are also increasingly involved in cyber and real world crimes. Among the many types of evidence held by a phone, an app's prior screen displays may be the most intuitive and valuable to an investigation — revealing the intent, targets, actions, and other contextual evidence of a crime. In this paper, we demonstrate a powerful forensics capability for Android phones: recovering *multiple previous screens* displayed by each app from the phone's memory image. Different from traditional memory forensics, this capability enables *spatial-temporal* forensics by revealing what the app displayed over a time interval, instead of a single time instance. For example, investigators will be able to recover the multiple screens of a banking transaction, deleted messages from an online chat, and even a suspect's actions before logging out of an app.

Our previous effort in memory forensics, GUITAR [35], provides a related (but less powerful) capability: recovering the most recent GUI display of an Android app from a memory image. We call this GUI display *Screen 0*. Unfortunately, GUITAR is *not* able to reconstruct the app's previous screens, which we call *Screens -1, -2, -3...* to reflect their reverse temporal order. For example, if the user has logged out of an app before the phone's memory image is captured, GUITAR will only be able to recover the "log out" screen, which is far less informative than the previous screens showing the actual app activities and their progression.

To address this limitation, we present a novel spatial-temporal solution, called RetroScope, to reconstruct an Android app's previous GUI screens (i.e., Screens 0, -1, -2... -N, $N > 0$). RetroScope is *app-agnostic* and does not require any app-specific knowledge (i.e., data structure definitions and rendering logic). More importantly, RetroScope achieves near perfect accuracy in terms of (1) reconstructed screen display and (2) temporal order of the reconstructed screens. To achieve these properties, RetroScope overcomes significant challenges. As indicated in [35], GUI data structures created for previous screens get overwritten almost completely, as soon as a new screen is rendered. This is exactly why GUITAR is unable to reconstruct Screen -i ($i > 0$), as it cannot find GUI data structures belonging to the previous screens. In other words, GUITAR is capable of "spatial" — but not "spatial-temporal" — GUI reconstruction. This limitation motivated us to seek a fundamentally different ap-

proach for RetroScope.

During our research, we noticed that although the GUI data structures for app screens dissolve quickly, the actual *app-internal data* displayed on those screens (e.g., chat texts, account balances, photos) have a much longer lifespan. Section 2 presents our profiling results to demonstrate this observation. However, if we follow the traditional memory forensics methodology of searching for [16,25,26,41] and rendering [35–37] instances of those app data, our solution would require app-specific data structure definitions and rendering logic, breaking the highly desirable app-agnostic property.

We then turned our attention to the (app-agnostic) display mechanism supplied by the Android framework, which revealed the most critical (and interesting) idea behind RetroScope. A smartphone displays the screen of one app at a time; hence the apps' screens are frequently switched in and out of the device's display, following the user's actions. Further, when the app is brought back to the foreground, its entire screen must be redrawn from scratch: by first "repackaging" the app's internal data to be displayed into GUI data structures, and then rendering the GUI data structures according to their layout on the screen. Now, recall that the "old" app-internal data (displayed on previous screens) are still in memory. Therefore, we propose redirecting Android's "draw-from-scratch" mechanism to those old app data. Intuitively, this would cause the previous screens to be rebuilt and rendered. This turns out to be both feasible and highly effective, thus enabling the development of RetroScope.

Based on the observations above, RetroScope is designed to trigger the re-execution of an app's screen-drawing code in-place within a memory image — a process we call *selective reanimation*. During selective reanimation, the app's data and drawing code from the memory image are logically interleaved with a live *symbiont app*, using our *interleaved re-execution engine* and *state interleaving finite automata* (Section 3.2). This allows RetroScope (within a live Android environment) to issue standard GUI redrawing commands to the interleaved execution of the target app, until the app has redrawn all different (previous) screens that its internal data can support. In this way, RetroScope acts as a "puppeteer," steering the app's code and data (the "puppet") to reproduce its previous screens.

We have performed extensive evaluation of RetroScope, using memory snapshots from 15 widely used Android apps on three commercially available phones. For each of these apps, RetroScope accurately recovered multiple (ranging from 3 to 11) previous screens. Our results show that RetroScope-recovered app screens provide clear spatial-temporal evidence of a phone's activities with high accuracy (only missing 2 of 256 recoverable screens) and efficiency (10 minutes on average to recover all screens for an app). We have open-sourced RetroScope[1] to encourage reproduction of our results and further research into this new memory forensics paradigm.

2 Problem and Opportunity

Different from typical desktop applications, frequent user interactions with Android apps require their screen display to be highly dynamic. For example, nearly all user interactions (e.g., clicking the "Compose Email" button on the Inbox screen) and asynchronous notifications (e.g., a pop-up for a newly received text message) lead to drawing an entirely new screen. Despite such frequent screen changes, an earlier study [35] shows that every newly rendered app screen destroys and overwrites the GUI data structures of the previous screen.

This observation however, seems counter-intuitive as Android apps are able to very quickly render a screen that is similar or identical to a previous screen. For example, consider how seamlessly a messenger app returns to the "Recent Conversations" screen after sending a new message. Given that the previous screen's data structures have been destroyed, the app must be able to *recreate* GUI data structures for the new screen. More importantly, we conjecture that the raw, app-internal data (e.g., chat texts, dates/times, and photos) displayed on previous screens must exist in memory long after their corresponding GUI data structures are lost.

To confirm our conjecture about the life spans of (1) GUI data structures (short) and (2) app-internal data (long), we performed a profiling study on a variety of popular Android apps (those in Section 4). Via instrumentation, we tracked the allocation and destruction (i.e., overwriting) of the two types of data following multiple screen changes of each app. Figure 1 presents our findings for TextSecure (also known as Signal Messenger). It is evident that the creation of every new screen causes the destruction of the previous screen's GUI data, whereas the app-internal data not only persists but accumulates with every new screen. We observed this trend across all evaluated apps.

Considering that a memory image reflects the memory's content at one time instance, Figure 1 illustrates a limitation of existing memory forensics techniques (background on memory image acquisition can be found in Appendix A). Specifically, given the memory image taken after Screen 0 is rendered (as marked in Figure 1), our GUITAR technique [35] will only have access to the GUI data for Screen 0. Meanwhile, the app's internal

[1] RetroScope is available online, along with a demo video, at: `https://github.com/ProjectRetroScope/RetroScope`.

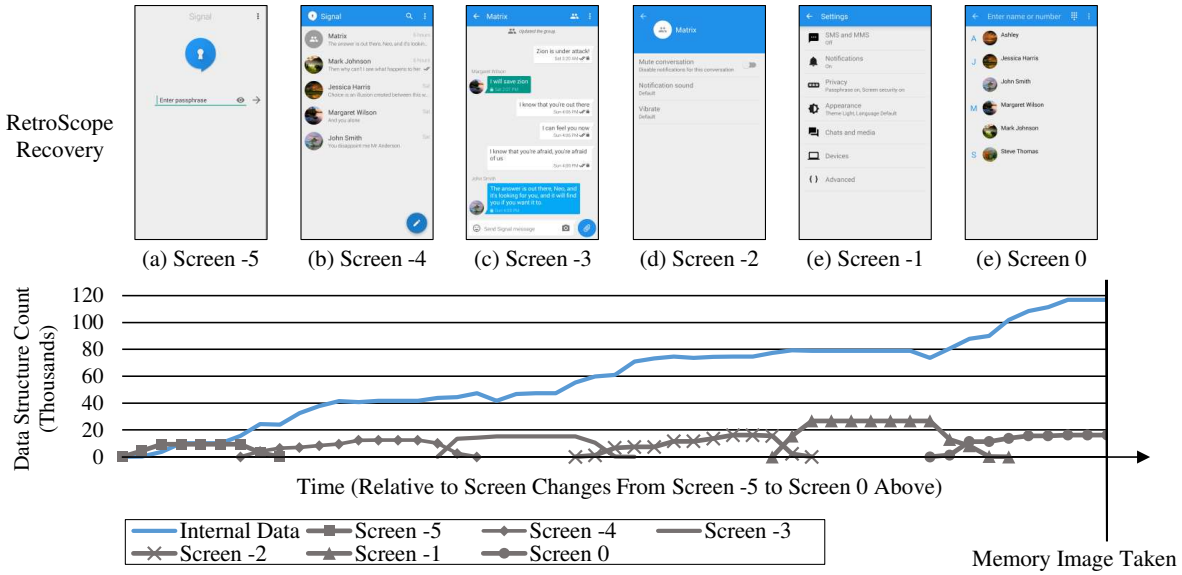


Figure 1: Life Cycles of GUI Data Structures Versus App-Internal Data Across Multiple Screen Changes.

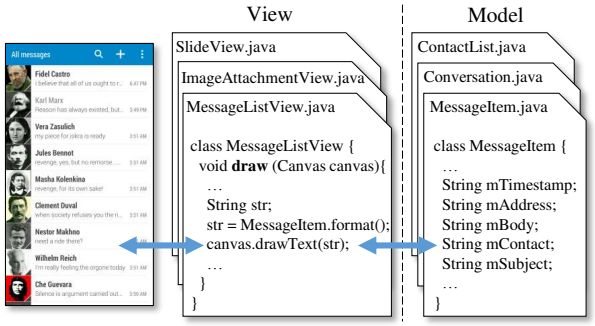


Figure 2: The Typical Model/View Implementation Split of Android Apps.

data are maintained by the app itself for as long as the app's implementation allows (e.g., we never observed TextSecure deallocating its messages because they may be needed again). However, without app-specific data definitions or rendering logic, it is impossible for existing app-agnostic techniques [6, 36, 41] to meaningfully recover and redisplay the app's internal data on Screens -1 to -5 in Figure 1.

It turns out that the Android framework instills the "short-lived GUI structures and long-lived app-internal data" properties in all Android apps. Specifically, Android apps must follow a "Model/View" design pattern which intentionally separates the app's logic into *Model* and *View* components. As shown in Figure 2, an app's Model stores its internal runtime data; whereas its View is responsible for building and rendering the GUI screens that present the data. For example, the MessageItem, Conversation, and ContactList (Model) classes in Figure 2 store raw, app-internal data, which are then formatted into GUI data structures, and drawn on screen by the MessageListView class. This design allows the app's View screens to respond quickly to the highly dynamic user-phone interactions, while delegating slower operations (e.g., fetching data updates from a remote server) to the background Model threads.

Further, the Android framework provides a Java class (aptly named `View`) which apps must extend in order to implement their own GUI screens. As illustrated by Figure 2's MessageListView class, each of the app's screens correspond to an app-customized View object and possibly many sub-Views drawn within the top-level View. Most importantly, each View object defines a `draw` function. `draw` functions are prohibited from performing blocking operations and may be invoked by the Android framework *whenever that specific screen needs to be redrawn*. This makes any screen's GUI data (e.g., formatted text, graphics buffers, and drawing operations which build the screen) easily disposable, because the Android framework can quickly recreate them by issuing a *redraw command* to an app at any time. This design pattern provides an interesting opportunity for RetroScope, which will intercept and reuse the context of a live redraw command to support the reanimation of `draw` functions in a memory image.

3 Design of RetroScope

RetroScope's operation is fully automated and only requires a memory image from the Android app being investigated (referred to as the *target app*) as input. From this memory image, RetroScope will recreate as many

previous screens as the app's internal data (in the memory image) can support. However, without app-specific data definitions, RetroScope is unable to locate or understand such internal data. But recall from Section 2 that the Android framework can cause the app to draw its screen by issuing a redraw command, without handling the app-internal data directly. This is possible because the app's `draw` functions are invoked in a *context-free* manner: The Android framework only supplies a buffer (called a `Canvas`) to draw the screen into, and the `draw` function obtains the app's internal data via previously stored, global, or static variables — analogous to starting a car with a key (the redraw command) versus manually cranking the engine (app internals). Thus, RetroScope is able to leverage such commands, avoiding the low-level "dirty work" as in previous forensics/reverse engineering approaches [36, 37].

RetroScope mimics this process *within* the target app's memory image by selectively reanimating the app's screen drawing functions via an *interleaved re-execution engine* (IRE). RetroScope can then inject redraw commands to goad the target app into recreating its previous screens. An app's `draw` functions are ideal for reanimation because they are (1) functionally closed, (2) defined by the Android framework (thus we know their interface definition), and (3) prevented from performing I/O or other blocking operations which would otherwise require patching system dependencies. Finally, RetroScope saves the redrawn screens in the temporal order that they were previously displayed, unless the `draw` function crashes — indicating the app-internal data could not support that screen.

To support selective reanimation, RetroScope leverages the open-source Android emulator to start, control, and modify the execution of a *symbiont app*, a minimal implementation of an Android app which will serve as a "shell" for selective reanimation.

3.1 Selective Reanimation

Before selective reanimation can begin, RetroScope must first set up enough of the target app's runtime environment for re-executing the app's `draw` functions. Therefore RetroScope first starts a new process in the Android emulator, which will later become the symbiont app and the IRE (Section 3.2). RetroScope then synthetically recreates a subset of the target app's memory space from the subject memory image. Specifically, RetroScope loads the target app's data segments (native and Java) and code segments (native C/C++ and Java code segments) back to their original addresses (Lines 1-4 of Algorithm 1) — this would allow pointers within those segments to remain valid in the symbiont app's memory space. RetroScope then starts the symbiont app which will initialize its native execution environment and Java runtime. Note that the IRE will not be activated until later when state interleaving (Section 3.2) is needed.

Isolating Different Runtime States. The majority of an Android app's runtime state is maintained by its Java runtime environment[2]. For RetroScope, it is not sufficient to simply reload the target app's memory segments. Instead the symbiont app's Java runtime must also be made aware of the added (target app's) runtime data prior to selective reanimation. Later, the IRE will need to dynamically switch between the target app's runtime state and that of the symbiont app to present each piece of interleaved execution with the proper runtime environment.

RetroScope traverses a number of global Java runtime data structures from the subject memory image with information such as known/loaded Java classes, app-specific class definitions, and garbage collection trackers (Lines 5–9 of Algorithm 1). Such data are then copied and isolated into the symbiont app's Java runtime by inserting them (via the built in Android class-loading logic) into duplicates of the Java runtime structures in the symbiont app. Note that, at this point, the duplicate runtime data structures will not affect the execution of the symbiont app, but they must be set up during the symbiont app's initialization so that any app-specific classes and object allocations *from the memory image* can be handled later by the IRE.

At this point, the symbiont app's memory space contains (nearly) two full applications (shown in Figure 4). The symbiont app has been initialized naturally by the Android system with its own execution environment. In addition, RetroScope has reserved and loaded a subset of the target app's memory segments (those required for selective reanimation) and isolated the necessary old (target app's) Java runtime data into the new (symbiont app's) Java runtime. The remainder of RetroScope's operation is to (1) mark the target app's View `draw` functions so that they can receive redraw commands and (2) reanimate those drawing functions inside the symbiont app via the IRE.

Marking Top-Level Draw Functions. RetroScope traverses the target app's loaded classes to find top-level Views (Lines 10–17 in Algorithm 1). Top-level Views are identified as those which inherit from Android's parent View class `ViewParent` and are not drawn inside any other Views. As described in Section 2, top-level Views are default Android classes which *contain* app-customized sub-Views. Further, we know that all Views must implement a `draw` function (which invokes the sub-Views' `draw` functions). Thus RetroScope marks each top-level `draw` function as a reanimation starting point.

[2]Please see Section 5 regarding Dalvik JVM versus ART runtimes.

Algorithm 1 RetroScope Selective Reanimation.

Input: Target App Memory Image M
Output: GUI Screen Ordered Set S

▷ Rebuild the Target App runtime environment.
1: **for** Segment $S \in M$ **do** ▷ Remap memory segments.
2: **if** isNeededForReanimation(S) **then**
3: Map($S.startAddress$, $S.length$, $S.content$)
4: $SymbiontApp$.initialize() ▷ Set up Symbiont App.
5: JavaGlobalStructs $G \leftarrow \varnothing$ ▷ Isolate the Target App runtime state.
6: **for** Segment $S \in M$ **do** ▷ Find Java control data.
7: **if** containsJavaGlobals(S) **then**
8: $G \leftarrow$ getJavaGlobals(S)
9: **break**
▷ Register reanimation points with the IRE.
10: InterleavedReexecutionEngine IRE
11: View Set $V \leftarrow \varnothing$ ▷ Top-level Views.
12: **for** Class $C \in G \rightsquigarrow Classes$ **do** ▷ Find top-level Views.
13: **if** $C <: ViewParent$ **then** ▷ '$<:$' denotes subtype.
14: **if not** isSubView(C) **then**
15: IRE.beginOn($C.draw$) ▷ Register drawing function.
16: View Set $views \leftarrow C.instances$
17: $V \leftarrow V \cup views$
18: View $T \leftarrow SymbiontApp$.getTopLevelView()
19: T.invalidate() ▷ Cause screen redraw command to be issued.
20: **procedure** CATCHREDRAWCOMMAND
▷ Invoked when redraw command is issued.
21: **for** View $view \in V$ **do**
22: $T \leftarrow view$ ▷ Override the Symbiont App's top-level View.
▷ Record largest subView ID.
23: $largestID \leftarrow \max_{v \in view.subViews} v$.getField($ID$)
24: deliverRedrawCommand()
▷ IRE handles re-execution of redrawing code.
25: Screen $s \leftarrow T$.copyGUIBuffer()
26: S.insert($largestID$, s)
27: **end procedure**

Selective Reanimation. Once all top-level `draw` functions are identified, RetroScope can begin selective reanimation of each. First, RetroScope invalidates the symbiont app's current View (Line 19 of Algorithm 1). This will cause Android to set up and issue a redraw command to the symbiont app along with a buffer to draw into. However, RetroScope first intercepts this command and replaces the symbiont app's top-level View with one of the target app's top-level Views identified previously (Lines 20–27 in Algorithm 1). Note that RetroScope does not distinguish between different instances of top-level Views, it simply reissues redraw commands for every previously identified top-level View instance, even if duplicates exist.

Since the top-level Views of the symbiont app and the target app are both default instances of (or inherit from) the same Android View class, they are interchangeable as far as the Android framework is concerned (both with the same functionality). Now RetroScope can inject the redraw command into the symbiont app which, upon receiving this command, will naturally invoke the target app's top-level `draw` function (previously marked for reanimation).

This will trigger the IRE to begin logically interleaving the `draw` function execution with the symbiont app's GUI drawing environment. Most importantly, this will direct input code/data accesses (i.e., queries to the target app's Model) to the appropriate target app functions and output code/data accesses (i.e., drawing of screens) to the symbiont app's running GUI framework. Upon successful completion of each `draw` function reanimation, RetroScope retrieves and stores the symbiont app's (now filled) screen buffer, switches the top-level View to another marked target app View, and re-injects the redraw command — reloading the memory image in between to avoid side effects.

Finally, RetroScope reorders the redrawn screens to match the temporal order in which they were displayed. This is done via comparison of View ID fields in the target app's Views (recovered from the memory image). A View's ID is an integer that identifies a View. The ID may not be unique, as some Views may alias others, but it is always set from a monotonically increasing counter. This yields the property that app screens can be ordered temporally by comparing the largest ID among their subViews. Intuitively, the most recently modified portion of the screen (sub-View) will yield an increasingly large ID.

3.2 Interleaved Re-Execution Engine

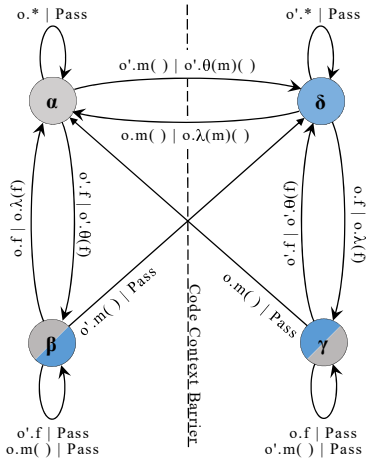


Figure 3: State Interleaving Finite Automata.

The key enabling technique behind RetroScope is its IRE which logically interleaves the state of the target app into the symbiont app just before it is needed by the execution. To monitor and interleave the execution contexts, the IRE intercepts the execution of Java byte-code instructions corresponding to function invocations, returns, and data accesses (i.e., instance/static field reads/writes). The IRE's operation is similar to parsing a lexical context-free grammar: The current byte-code instruction (i.e., token) and the context of its operands (e.g., new/old data) are matched to a *state interleaving*

finite automata (Figure 3), where each state transition defines which runtime environment the IRE should present to that instruction.

In RetroScope, state interleaving begins at the invocation of one of the marked top-level `draw` functions within the target app. As a running example, Figure 4 shows a snippet of a `draw` function's code along with the live memory space (containing both the symbiont app and the target app's execution environment).

IRE State Tracking. For each byte code instruction, the IRE tracks two pieces of information: (1) if the code being executed is from the memory image (*old code*) or from the symbiont app (*new code*) and (2) if the current runtime information (i.e., loaded classes, object layouts, etc.) originates from the memory image (*old runtime*) or the symbiont app (*new runtime*). Based on that, the execution context may be in any of four possible states:

$$\begin{aligned}(\text{new code}, \text{new runtime}) &= \textcircled{\alpha} \\ (\text{new code}, \text{old runtime}) &= \textcircled{\beta} \\ (\text{old code}, \text{new runtime}) &= \textcircled{\gamma} \\ (\text{old code}, \text{old runtime}) &= \textcircled{\delta}\end{aligned} \tag{1}$$

In Figure 4, we have denoted which state the IRE is in before and after executing each line of code. For ease of explanation, Figure 4 presents source code, but RetroScope operates on byte-code instructions only. For example, before executing Line 1, the IRE is in $\textcircled{\alpha}$ because no old code or data has been introduced yet. Likewise, after Line 1, the IRE is in $\textcircled{\delta}$ as the IRE is then executing the target app's `draw` function (old code) within the target app's top-level View object (old runtime). However, note that the context of runtime data may not (and often does not) match the context of the code: For example, in Line 4, fetching the `mDensity` field from the new `Canvas` requires using the new runtime data but is being performed by old code (resulting in state $\textcircled{\gamma}$).

Modeling State-Transitions. In Figure 3, we generalize the state-transition rule matching to two primitive operations: Given an object o, state transitions may occur when accessing a field f within o ($o.f$) or when invoking a method m defined by o ($o.m()$). Further, o may be an object loaded from the target app's memory image or allocated by the target app's code (i.e., interacting with this object requires the old runtime data), thus we denote such *old objects* as o' in Figure 3. Note that our discussion will follow Java's object-oriented design, but the transitions in Figure 3 are equally applicable to static (i.e., o == NULL) execution.

The state transitions in Figure 3 are modeled as a Mealy machine [29] with the input of each state-transition being a matched operation and the output being the corresponding state correction performed by the IRE. These state corrections (i.e., transition outputs) fall into three categories: (1) a transition from the new runtime data to the old runtime data (the function θ), (2) a transition from old to new runtime data (the function λ), and (3) no change in runtime data ("Pass"). For example, the transition from $\textcircled{\alpha}$ to $\textcircled{\delta}$ is represented as:

$$\textcircled{\alpha} \rightarrow \textcircled{\delta} : o'.m() \mid o'.\theta(m)() \tag{2}$$

where the input to this transition is a match on $o'.m()$ (invoking an old object's method) and the output state correction is to switch to the old runtime prior to invoking the method ($o'.\theta(m)()$). This is exactly the IRE's transition before executing Line 1 in Figure 4 as the IRE must switch to the old runtime prior to invoking the old View object's `draw` function to look up the method's implementation. Conversely, the transition from $\textcircled{\gamma}$ to $\textcircled{\alpha}$ is represented as:

$$\textcircled{\gamma} \rightarrow \textcircled{\alpha} : o.m() \mid \text{Pass} \tag{3}$$

because this transition occurs when a new object's method is invoked ($o.m$) but the IRE is already using the new runtime data, thus no runtime data correction is needed (i.e., "Pass"). This case is observed in Line 11 of Figure 4. At the beginning of Line 11, the IRE is in state $\textcircled{\gamma}$ due to the lookup of the new `Canvas`'s `mDensity` field on Line 4. Thus, the invocation of `getClipBounds` on Line 11 does not require the runtime to change (a "Pass" transition), but does change from old code to new.

Another important corrective action in Figure 3 is whether or not a transition crosses the code context barrier (i.e., a horizontal transition). Crossing the code context barrier signifies a switch between fetching new code (from the symbiont app) to old code (from the memory image) or vice versa. Although crossing the context barrier alone does not require active correction by the IRE (e.g., the old runtime's method definitions will naturally direct the execution to the old code), the IRE must note that the change occurred.

Monitoring which context the code is fetched from is essential for a number of runtime checks and corrections that the IRE must perform. Firstly, objects allocated while executing old code should use the class definitions from the target app (as the Android framework classes may be vendor-customized or the class may be defined by the target app itself). Secondly, type comparisons (e.g., the Java `instanceof` operator) executed by old code must consider both new and old classes but prefer old classes. This is because new objects (which are instances of classes loaded by the symbiont app's runtime) will be passed into old code functions — which use the target app's loaded classes that contain "old duplicates" of classes common to both executions (e.g., system classes). The reverse is true for new code type comparisons. Lastly, exceptions thrown during interleaved execution should be catchable by both old and new code.

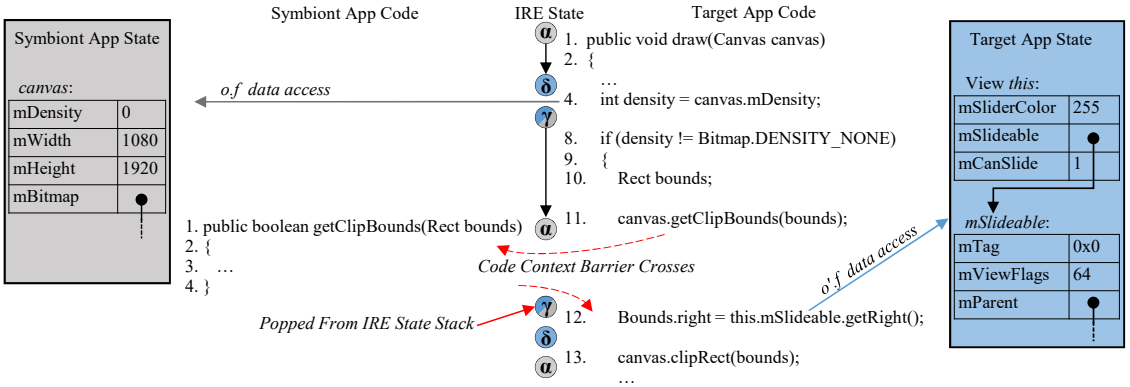


Figure 4: Example of Interleaved Re-Execution.

Interestingly, we find a number of test cases in Section 4 *purposely* throw exceptions inside their inner drawing functions, and allowing new code to catch old code exceptions (or vice versa) requires patching type lookups (as before) and stack walking.

Return Transitions. Although Figure 3 does not illustrate state transitions for return instructions, the IRE does perform state correction for them. Unlike the transitions in Figure 3 (which rely on the current IRE state to determine a new state), method returns simply restore the IRE state from before the matching invocation. This is tracked by a stack implemented in the IRE which pushes the current IRE state before invoking a method and pops/restores that IRE state upon the method's return. This behavior is seen in Line 12 in Figure 4. Before the invocation of `getClipBounds` (Line 11), the IRE is in state (γ). Function `getClipBounds` executes in state (α), and upon its return the IRE pops state (γ) from the stack and restores that state prior to executing Line 12.

Another notable simplification of the IRE's design is that it is sufficient to only perform state correction at function invocations, returns, and field accesses. Intuitively, this is because other "self-contained" instructions (e.g., mathematical operations) do not require support from the runtime. But another advantage is that state-interleaving tends to occur after bunches of instructions. Our evaluation shows that on average 10.24 instructions in a row will cause loop-back transitions before a state correction is needed. Further, many functions execute entirely in state (α) or (δ) because no data from the other environment enter those functions.

Native Execution. The IRE operates on the Java bytecode instructions of the functions marked for selective reanimation. However, it is possible that app developers utilize the Java Native Interface (JNI) to implement some of their app's functionality in native C/C++ code. Further, the Android framework heavily uses JNI functions. When the IRE observes an invocation of a C/C++ function, it follows the same state transitions defined in Figure 3 (i.e., new code only invokes new C/C++ functions and vice versa).

Luckily, due to the tightly controlled interaction between C/C++ functions and the Java runtime data, the IRE's state correction can be further simplified. To access data or invoke methods from the Java runtime, C/C++ functions must use a set of helper functions defined by the Java runtime. The IRE hooks these functions and checks if the data or method being requested is in the old or new context. The IRE can then properly patch the helper function's return value and allow the C/C++ function to execute as intended. Note that, because all the target app's native code and data segments have been mapped back to their original addresses, all pointers (code and data) in those segments remain valid.

Lastly, although the IRE executes app-specific code, it does so on a *syntactic* basis *without* understanding the code's semantics, hence maintaining RetroScope's app-agnostic property.

3.3 Escaping Execution and Data Accesses

To monitor and interleave the target app's reanimation, the IRE must accurately track the current state of the execution environment. However, due to the relative complexity of Android apps, it is possible that the target app's control flow causes the IRE to miss a state transition, potentially failing to correct the execution environment despite the actual execution being in a different state. We call such missed state transitions *escaping execution* or *escaping data accesses*.

Escaping Execution. This occurs when the target app's reanimation invokes a function but the IRE is unable to determine which context to transition to. This is primarily due to the invocation of a static method which exists in both the old and new environments — leading to an *ambiguous state-transition*, where the IRE does not have

sufficient information at the function invocation site to determine which state ($\textcircled{\alpha}$ or $\textcircled{\delta}$) to transition to. Simply put, the IRE must discover if the execution intended to invoke the old or new method. To decide that, the IRE performs a simple data flow analysis on each version. If the method writes data to a static variable, then the IRE always invokes the method in state $\textcircled{\alpha}$, otherwise the IRE keeps the same state that the method was invoked by (to avoid an unnecessary transition). This ensures that any accesses to static values which exist in both old and new environments are always directed to the new one. Note that app-defined static variables will only exist in the old environment, and thus their accesses do not lead to ambiguous transitions.

Escaping Data Accesses. This occurs when an app implements a non-standard means of accessing an object's fields. For example, the two most common causes of escaping data accesses we observed are: (1) C/C++ code using a hard-coded Java object layout to access an object's fields and (2) old Java code which has cached an old version of an object which RetroScope is trying to replace with a new version (e.g., some Views will save and reuse a reference to the previously drawn on `Canvas`). Although escaping data accesses are caused by app implementation differences, they can be handled uniformly by the IRE.

Escaping data accesses caused by Java code can be identified automatically when the fields of the object are accessed incorrectly. For example, there should not exist any old `Canvas` objects during selective reanimation and thus the IRE will identify its field accesses and replace the object with the new instance. Escaping data accesses caused by C/C++ code are handled by preventing C/C++ code from directly accessing Java objects. Instead, the IRE requires all pointers to Java objects to be encoded before they are given to C/C++ code. These pointers can be decoded when they are used in the standard JNI field access helper functions, but will cause a segmentation fault when dereferenced erroneously. This segmentation fault can then be handled by RetroScope to patch the field access with the appropriate JNI helper function. In fact, support for encoded/decoded JNI pointers already exists but may be avoided in Android, so the IRE only needs to require that all JNI pointers are encoded/decoded and handle the segmentation fault for those that previously avoided this functionality.

4 Evaluation

Evaluation Setup. Our evaluation of RetroScope involved three Android phones (a Samsung Galaxy S4, HTC One, and LG G3)[3] as evidentiary devices. On each phone, we installed and interacted with 15 different apps to cause the generation, modification, and deletion of as many screens as possible. The interactions took an average of 16 minutes per app, and we installed and interacted with the apps on each phone at random times over a 4-day period. Then, for each phone, we waited 60 minutes for any background activity of the 15 apps to complete, after which we took a memory image from the phone (as described in Appendix A).

The set of 15 apps was chosen to represent both typical app categories (to highlight RetroScope's generic applicability) and diverse app implementation (to evaluate the robustness of RetroScope's selective reanimation). Based on the importance of personal communication in criminal investigations, we included Gmail, Skype, WeChat, WhatsApp, TextSecure (also known as Signal, notable for its privacy-oriented design which limits evidence recovery [4]), Telegram (whose encrypted broadcast channels are popular with terrorist organizations [3]), and each device's default MMS app (implemented by the device vendor). We also included the two most popular social networking apps: Facebook (known for its highly complex/obfuscated implementation) and Instagram. Finally we consider several apps which, by nature, display sensitive personal information: Chase Banking, IRS2Go (the official IRS mobile app), MyChart (the most popular medical record portfolio app), Microsoft Word for Android, and the vendor-specific Calendar and Contacts/Recent Calls apps.

We then used RetroScope to recreate as many previous app screens as still exist in the memory images of the 45 (15×3) apps. The recovery results are reported in Table 1. Table 1 presents the device and app name in Columns 1 and 2, respectively. Column 3 shows the ground-truth number of screens that RetroScope should recover, and Column 4 reports the number of screens recovered. Columns 5 through 9 present several metrics recorded over the selective reanimation of all screen redrawing functions for each app: Column 5 shows the number of reanimated Java byte-code instructions, Column 6 reports the number of JNI invocations (i.e., C/C++ functions invoked from Java code) observed, and Columns 7 and 8 report the total number of newly allocated Java objects and C/C++ structures that made up the new screens. Column 9 shows RetroScope's runtime for each case.

Selective Reanimation Metrics. Table 1 provides interesting insights into the complexity and scale of screen redrawing via selective reanimation. From Table 1, we learn that an average of 231,867 byte-code instructions

[3]These devices all run vendor-customized versions of Android Kitkat (the most widely used Android version [17]).

Device	App	Expected # of Screens	RetroScope Recovery	Metrics for Evaluating Selective Reanimation				
				Byte-Code Instructions	JNI Invocations	Allocated Java Objects	New C/C++ Structures	Runtime (seconds)
Samsung S4	Calendar	8	8	259196	4699	930	79119	502
	Chase Banking	9	9	424336	9318	1905	106168	1610
	Contacts	5	5	199755	4606	928	49322	369
	Facebook	6	6	338195	7928	1432	45420	1059
	Gmail	5	5	188463	4185	826	80808	487
	Instagram	7	7	240139	5191	482	86319	672
	IRS2Go	5	5	195413	4450	790	21027	674
	MMS	3	3	96856	2004	333	25311	276
	Microsoft Word	3	3	211762	4273	460	58291	637
	MyChart	4	4	74213	1632	367	18902	259
	Skype	6	6	236213	5256	1072	30753	486
	Telegram	6	7	177973	3488	314	41815	664
	TextSecure	4	4	145436	3461	763	27450	450
	WeChat	3	3	121630	2823	638	24730	831
	WhatsApp	7	8	402536	8186	1373	65818	1390
LG G3	Calendar	7	7	199290	4193	665	72944	478
	Chase Banking	8	8	360607	8436	1843	127337	1731
	Contacts	5	5	313068	6289	1184	105004	430
	Facebook	7	7	448535	10038	1892	88949	1413
	Gmail	6	6	263850	6148	1353	239711	1248
	Instagram	5	5	245094	5097	489	104391	446
	IRS2Go	6	6	335323	7599	1458	82077	709
	MMS	6	6	147428	3077	422	61210	303
	Microsoft Word	4	4	175394	4189	652	51769	375
	MyChart	3	3	59284	1291	202	24995	335
	Skype	6	5	238227	4914	914	63007	382
	Telegram	6	6	125085	2452	183	48496	297
	TextSecure	6	6	206146	4388	860	80672	381
	WeChat	4	5	225245	5296	1293	72310	632
	WhatsApp	7	8	205661	4548	884	67789	466
HTC One	Calendar	6	6	197316	3675	732	102642	749
	Chase Banking	11	11	584587	12591	2091	266965	850
	Contacts	3	3	190847	4023	723	71578	380
	Facebook	6	5	382522	8629	1451	95516	1128
	Gmail	6	6	235973	5366	929	129804	1128
	Instagram	3	3	86829	2078	433	42037	399
	IRS2Go	5	5	200196	4510	832	52097	547
	MMS	4	4	93971	1950	287	45085	493
	Microsoft Word	3	3	137978	3249	562	43209	456
	MyChart	6	6	131876	2599	353	65377	403
	Skype	9	9	468258	9817	1232	149372	890
	Telegram	4	4	98662	1989	185	49902	291
	TextSecure	7	8	231891	5268	924	98571	488
	WeChat	5	5	211518	4836	901	69587	723
	WhatsApp	6	6	321229	7075	1571	104216	573

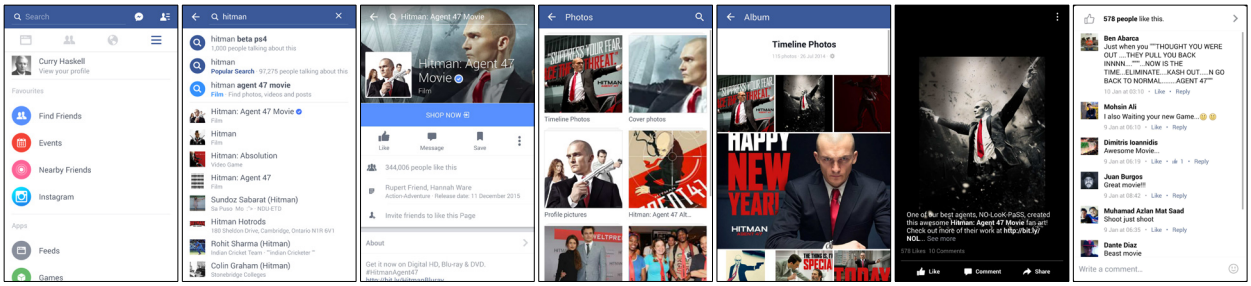
Table 1: Overall Results of RetroScope Evaluation.

and 5,047 JNI function invocations are required to redraw all of the screens for a single app. This yields an average of 41,078 byte-code instructions and 894 JNI function invocations *per screen*. Higher than our initial expectations, these numbers attest to the complexity of the screen drawing implementation and robustness of RetroScope's IRE.

Another metric above our expectation was the number of data structures that had to be *newly allocated* to redraw each screen. While redrawing all previous screens of each app, the reanimated code allocated an average of 891 Java objects and 76,397 C/C++ structures per app, and an average of 158 Java objects and 13,535 C/C++ structures *per screen*. These numbers confirm the claim in GUITAR [35] that each screen is made of "thousands of GUI data structures." Most importantly, as also shown in [35], only the structures for Screen 0 may still exist in a memory image, whereas RetroScope actively triggers the *rebuilding* of the lost data for Screens 0, -1, -2, ... -N.

4.1 Spatial-Temporal Evidence Recovery

Ground Truth. We now evaluate how accurately RetroScope recreates the screens displayed during our last



(a) Screen -6. (b) Screen -5. (c) Screen -4. (d) Screen -3. (e) Screen -2. (f) Screen -1. (g) Screen 0.

Figure 5: LG G3 Facebook Recovery.

interaction session with each app. However, obtaining the ground truth (how many previous screens RetroScope *should* recover) is not straightforward because the screens' recoverability is decided by the availability of the app's internal data in the memory image. Therefore, to identify the recoverable previous screens, we instrumented each app to log any *non-GUI-related* data allocations and accesses performed by each screen-drawing function. We then compared this log to the content of the final memory image to identify which screens' *entire* app-internal data still existed[4]. This gives us a strict lower bound on the number of screens that RetroScope should recover (i.e., all the internal data for those screens exist in the memory image). *Without* app-specific reverse engineering efforts, it is impossible to know the upper bound that the app's internal data could support. But as we discuss later, screen redrawing is often "all or nothing" and adheres closely to this lower bound.

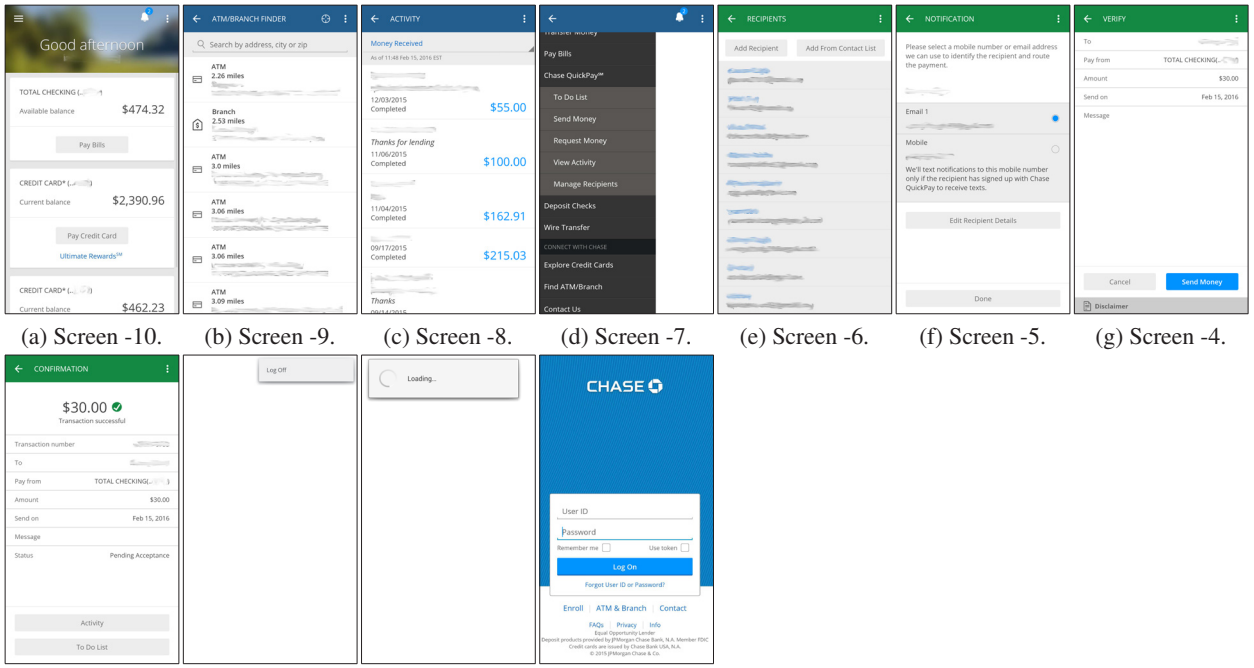
Highlights of Results. RetroScope recovered a total of 254 screens for the 45 apps, from a low of 3 to a high of 11 screens — ironically for the privacy sensitive Chase Banking app on the HTC One phone (Figure 6). Overall, Table 1 shows that RetroScope recovers an average of 5.64 screens per app, with the majority of the test cases (33 out of 45) having 5 or more screens.

Table 1 highlights the depth of *temporal* evidence that RetroScope makes available to forensic investigators, but even more intriguing is the clear progression of user-app interaction portrayed by the recovered screens. Figure 5 shows the 7 screens recovered for the Facebook app on the LG G3 phone. From these screens we can infer the "suspect's" progression: from his own profile (Screen -6), to search results for "hitman" (Screen -5), to the Facebook profile (Screen -4), Photos screen (Screen -3), a photo album (Screen -2) of the Hitman movie, to a single photo (Screen -1), and lastly to that photo's comments (Screen 0). Such powerful spatial-temporal recovery — from a single memory image — is not possible via any existing memory forensics technique.

Another interesting observation from Table 1 is that, although RetroScope's recovery is app-agnostic, the apps' diverse implementations lead to very different redrawing procedures. For example, for both Skype and Facebook apps on the Samsung S4, RetroScope reproduced all 6 screens from each app. However, Facebook's redrawing implementation appears much more complex, requiring 338,195 byte-code instructions and 7,928 JNI invocations, compared to Skype's 236,213 byte-code instructions and 5,256 JNI invocations. This also leads to varied RetroScope run times: from the shortest, Samsung S4's MyChart, at 259 seconds to the longest, LG G3's Chase Banking, at 1731 seconds. The average runtime across all apps is 655 seconds (10 minutes, 55 seconds).

Lastly, Table 1 shows that in two cases (Rows 26 and 34), RetroScope missed a single screen. Manual investigation of these cases revealed that the app-specific drawing functions for the missed screens had thrown unhandled Java exceptions. For the HTC One device's Facebook case, we found that the app had stored a pointer to the Thread object which handled its user interface and during redrawing the app failed on a check that the current Thread (handled by RetroScope during reanimation) is the same as the previously stored Thread (from the memory image). For the LG G3 Skype case, when drawing the "video call" screen, a saved timer value (in the memory image) was compared against the system's current time, which also failed during reanimation. These were addressed by reverse engineering to determine which field/condition in the app caused the fault, and RetroScope can be instructed to set/avoid them during interleaved execution. Also of note, several cases required recovering on-screen elements (e.g., user avatars) which were cached on persistent storage until they are loaded on the screen. Currently, RetroScope attempts to detect (e.g., via the unhandled exception) but can not automatically correct such implementation-specific semantic constraints. We leave this as future work.

[4]Note that RetroScope did not have access to nor could benefit from this ground truth information. Further, we utilized in-place binary instrumentation (which does not interact nor interfere with the app's execution or memory management) to ensure the accuracy of our experiments.



(a) Screen -10. (b) Screen -9. (c) Screen -8. (d) Screen -7. (e) Screen -6. (f) Screen -5. (g) Screen -4.

(h) Screen -3. (i) Screen -2. (j) Screen -1. (k) Screen 0.

Figure 6: HTC One Chase Banking Recovery.

4.2 Case Study I: Behind the Logout

We now elaborate on the Chase Banking app case and highlight RetroScope's ability to recreate an app's previous screens *even after the user has logged out*. Table 1 Row 32 shows that RetroScope recovered 11 out of 11 screens (the highest of all cases). Not surprisingly, the recovery required the most reanimated byte-code instructions (584,587) and JNI function invocations (12,591), as well as the most re-allocated Java objects (2,091) and C/C++ structures (266,965).

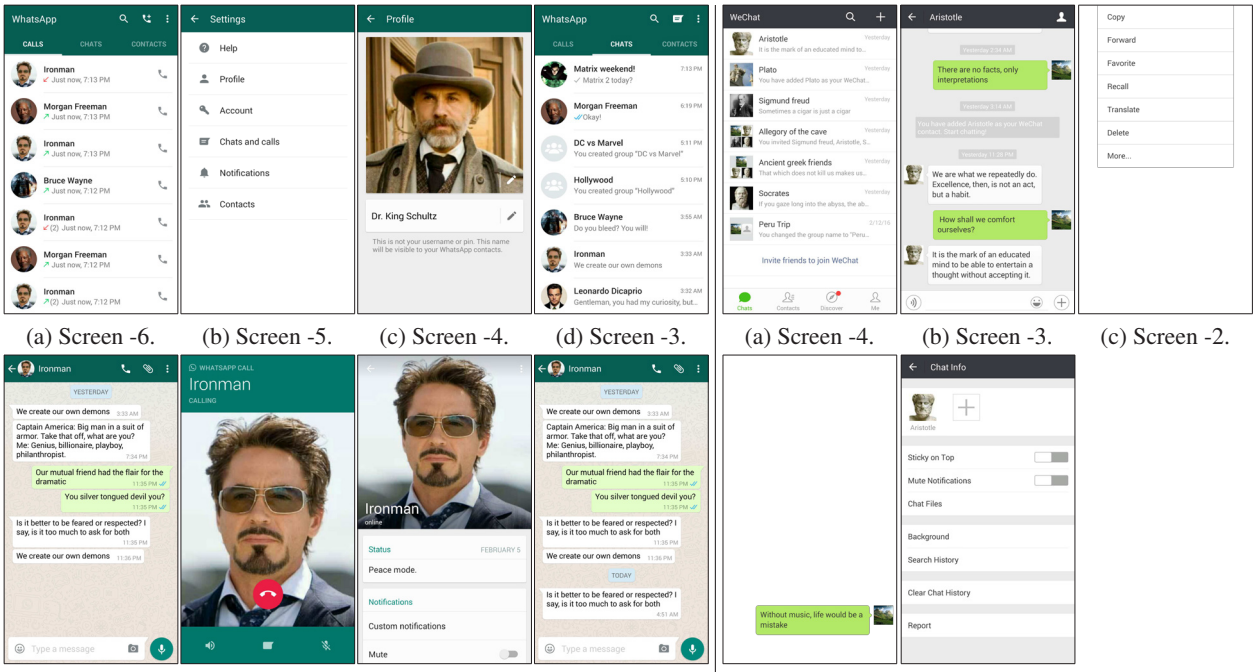
The recovered screens are shown in Figure 6. Starting from the Account screen (Screen -10), the "suspect" looks up a nearby ATM (Screen -9). He then reviews his recent money transfers (Screen -8) and begins a new transfer to a friend via the app's options menu (Screen -7). Screens -6 to -4 fill in the transfer's recipient and amount. Screen -3 asks the user to confirm the transfer. Screen -2 shows the app's "Log Out" menu, Screen -1 presents a loading screen while the app logs out, and Screen 0 is (as expected) the app's log in screen.

This case yields some interesting observations: First, it highlights the robustness of RetroScope to recover a large number of screens when an app's internal data continues to accumulate. More importantly, the case shows that, after logging out, the Chase app (as well as many others we have tested) does not clear its internal data. This is not surprising because programmers usually consider their app's *memory* to be private (compared to network communications or files on persistent storage). This is further evidenced by the TextSecure app, which also allows for a significant post-logout recovery (of pre-logout screens), despite the app's message database being locked in the device's storage.

4.3 Case Study II: Background Updates

Another interesting case is WhatsApp Messenger on the Samsung S4. Table 1 Row 15 shows that RetroScope reanimated 402,536 byte-code instructions and 8,186 JNI functions in 23 minutes, 10 seconds, yielding an average of 50,317 instructions and 1,023 JNI functions per screen. What was unexpected however is that RetroScope recovered an *extra* screen (8 out of the 7 expected screens) from the memory image.

Our investigation into this extra screen found that it was not a screen we had previously seen during our phone usage. Instead, after we had finished interacting with WhatsApp, the app received a new chat message *while it was in the background* and, to our surprise, this prompted the app to prepare a new chat screen that appended the newly received message to the chat. Figure 7 presents the screens recovered by RetroScope, and again we see a clear temporal progression through the app by the "suspect." First, Screen -6 shows the call log screen. The app's Settings screen is seen in Screen -5 followed by a screen that is only accessible through the Settings: the device owner's profile (our fictitious device owner is Dr. King Schultz) in Screen -4. Screen -3 shows the recent chats; Screen -2 shows the "suspect's" chat with a friend; then Dr. Schultz places a call to that friend



(a) Screen -6. (b) Screen -5. (c) Screen -4. (d) Screen -3.

(e) Screen -2. (f) Screen -1. (g) Screen 0. (h) Screen +1.

Figure 7: Samsung S4 WhatsApp Recovery.

(a) Screen -4. (b) Screen -3. (c) Screen -2.

(d) Screen -1. (e) Screen 0.

Figure 8: LG G3 WeChat Recovery.

in Screen -1. Lastly, Screen 0 shows the friend's profile. Then, the extra *Screen +1* shows the chat screen as prepared by the app while in the background. Indeed it shows the newly received message, even time-stamped ("TODAY" and "4:51 AM" in Figure 7(h)) after the previous chat had taken place.

To ensure that this result was not an accident, we repeated the experiment (receiving chat messages while the app was in the background) six more times (twice per device). In every test, we found that RetroScope recovered the additional pre-built chat screen containing the new message. Strangely, after testing the other apps which can receive background updates, we found that WhatsApp is the only app, among our 15 apps, that exhibited this behavior. We suspect that this is a WhatsApp-specific implementation feature to speed up displaying the chat screen (Screen +1) when the device user clicks the "New Message" pop-up notification.

4.4 Case Study III: Deleted Messages

In addition to the WhatsApp case above, RetroScope recovered *extra screens* for four other cases in Table 1: Telegram (Row 12), WeChat (Row 29), WhatsApp (Row 30), and TextSecure (Row 43). However, the extra screens here are for a different reason: RetroScope can recover *explicitly deleted* chat messages. In these tests, we began a chat in each app and then explicitly deleted one of the messages (as a suspect would do in an attempt to hide evidence), and then used RetroScope to recover the deleted message. Additionally, RetroScope also recovered proof of the suspect's intent to delete the message: For WeChat and WhatsApp, RetroScope recovered the app's pop-up menu (just prior to the deleted message) which displays the "Delete Message" option. For TextSecure, RetroScope recovered both the pop-up menu and a loading screen showing the text "Deleting Messages."

Figure 8 shows one example: RetroScope's recovery for the WeChat app on the LG G3. Screen -4 shows the "suspect's" recent chats followed by a chat conversation with a friend in Screen -3. Screen -2 is the pop-up menu displaying the "Delete" option. The deleted message (now disconnected from the previous chat window) is displayed in Screen -1, and the friend's profile page (which the "suspect" navigated to last) is shown in Screen 0.

This result, in particular, highlights one of the most powerful features of RetroScope, given that it works for many apps and even provides proof of the suspect's intent. Further, all four apps tout their *encrypted* communication and some (e.g., TextSecure) even encrypt the message database in the device. In light of this, law enforcement has routinely had trouble convincing developers of such apps to backdoor their encryption in support of investigations [4, 5]. Despite the few hardening measures discussed in Section 5, RetroScope can provide such alternative evidence which would otherwise be unavailable to investigators.

5 Privacy Implications and Discussion

RetroScope provides a powerful new capability to forensic investigators. But despite being developed to aid criminal investigations, RetroScope also raises privacy concerns. In digital forensics practice, the privacy of device users is protected by strict legal protocols and regulations [9,21], the most important of which is the requirement to obtain a search warrant prior to performing "invasive" digital forensics such as memory image analysis. Outside the forensics context, even some of the authors were surprised by the *temporal depth* of screens that RetroScope recovered for many privacy-sensitive apps (e.g., banking, tax, and healthcare). In light of this, we discuss possible mitigation techniques which, despite their significant drawbacks, might be considered worthwhile by privacy-conscientious users/developers.

RetroScope's recovery is based on two fundamental features of Android app design: (1) All apps which present a GUI must draw that GUI through the provided View class's `draw` function and (2) The Android framework calls drawing functions on-demand and prevents those drawing functions from performing blocking operations (file/network reads/writes, etc.). As such, an app that aims to disrupt RetroScope's recovery would need to hinder its own ability to draw screens.

Previous *anti-memory-forensics* schemes focused on encrypting in-memory data after its immediate use. This ensures that traditional memory scanning or data structure carving approaches (e.g., [25, 26, 37, 41]) would not find any useful evidence beyond the few pieces of decrypted in-use data. However, these solutions cannot hinder RetroScope's recovery because RetroScope recovers evidence via the app's existing `draw` functions, which would have to include decryption routines as part of building the app screen. App developers may add state-dependent conditions to their `draw` functions which would crash when executed by RetroScope, but as seen in Section 4 these can still be handled via additional debugging/reverse engineering efforts to skip/fix the conditions.

One approach that may disable RetroScope's recovery is to overwrite (i.e., zero) all app-internal data immediately after they are drawn on screen. By doing so, RetroScope would find that the app's internal state could not support the execution of any of its `draw` functions. Unfortunately, this approach would significantly degrade usability and increase implementation complexity: First, frequently overwriting app-internal data would incur execution overhead (especially during screen changes which are expected to be fast and dynamic). More importantly, this would require the app to download its internal data from a remote server *every time the app needs to draw a screen*. An app may attempt to amortize these overheads (e.g., only zeroing a prior session's memory upon logout) but this would require: (1) tracking used/freed memory throughout the session (to be zeroed later) and (2) users to regularly log out, which is uncommon and inconvenient for frequently used apps such as email, messengers, etc.

Current vs. Future Android Runtimes. It is worth noting that Google has begun shifting the Android framework's runtime from the Dalvik JVM to a Java-to-native compilation and native execution environment (named ART). Our implementation of RetroScope was based on the original (and still the most widely used by far [17]) Dalvik JVM runtime. However, during our development of RetroScope, specific care was taken to design RetroScope to utilize only features present in *both* runtimes. Specifically, ART still provides the same Java runtime tracking and support as Dalvik does (implemented now via C/C++ libraries) and all apps' implementations (e.g., their Views and `draw` functions) remain unchanged. Our study of ART revealed that the only engineering effort required to port RetroScope is the interception of state-changing instructions in the compiled byte-code, rather than the literal byte-code as it exists in Dalvik. We leave this as future work.

6 Related Work

RetroScope is most related to GUITAR [35] which, by recovering the remaining "puzzle pieces" (GUI data structures) from a memory image, is able to piece together an app's Screen 0. Motivated by GUITAR's "Screen 0-only" limitation (i.e., spatial recovery), RetroScope enables the fundamentally more powerful capability of recovering Screens 0, -1, -2, ... -N (i.e., spatial-temporal recovery). Technically, GUITAR is based on geometric matching of GUI pieces; whereas RetroScope is based on selective reanimation of GUI code and data.

A number of other (spatial) memory forensics tools have also been developed recently for Android. Many of these approaches recover raw instances of app-specific data structures to reveal evidence: App-specific login credentials were recovered by Apostolopoulos et al. [8]. Macht [28] followed by Dalvik Inspector [6] involved techniques to recover Dalvik-JVM control structures and raw Java object content. Earlier, Thing et al. [42] found that text-based message contents could be recovered from memory images. Most recently, our VCR [36] technique made it possible to recover images/video/preview frames from a phone's camera memory.

In a mobile device-agnostic effort, DEC0DE [44] involved an effective technique to carve plain-text call logs and address book entries from phone storage using probabilistic finite state machines.

RetroScope shares the philosophy of leveraging existing code for memory content rendering with our prior memory forensics technique DSCRETE [37]. However, DSCRETE renders a single application data structure, whereas RetroScope renders full app display screens in temporal order. More importantly, DSCRETE requires *application-specific* (actually, data structure-specific) identification and extraction of data rendering code, while RetroScope is totally *app-agnostic*, requiring no analysis of app-internal data or rendering logic. Finally, DSCRETE works on Linux/x86 whereas RetroScope works on the Android/ARM platform.

Many prior memory forensics techniques leverage memory image scanning and data structure signature generation approaches [11,12,16,26,32,34,38,41]. Data structure signatures can be content-based [16] or "points-to" structure-based [13, 15, 25, 26, 30]. For binary programs without source code, a number of reverse engineering techniques have been proposed to infer data structure definitions [24, 27, 39]. As a fundamentally new memory forensics technique, RetroScope requires neither data structure signature generation nor memory scanning.

7 Conclusion

We have presented RetroScope, a spatial-temporal memory forensics technique (and new paradigm) that recovers multiple previous screens of an app from an Android phone's memory image. RetroScope is based on a novel interleaved re-execution engine which selectively reanimates an app's screen redrawing functionality without requiring any app-specific knowledge. Our evaluation results show that RetroScope can recover visually accurate, temporally ordered screens (ranging from 3 to 11 screens) for a variety of apps on three different Android phones.

Acknowledgments

We thank the anonymous reviewers for their insightful comments and suggestions. This work was supported in part by NSF under Award 1409668.

References

[1] Advanced jtag mobile device forensics training. `http://www.teeltech.com/mobile-device-forensics-training/jtag-forensics/`, 2015.

[2] Forensics wiki - memory imaging tools. `http://forensicswiki.org/wiki/Tools:Memory_Imaging`, 2015.

[3] ISIS still using Telegram channels - Business Insider. `http://www.businessinsider.com/isis-telegram-channels-2015-11`, 2015.

[4] Signal, the Snowden-Approved Crypto App, Comes to Android. `http://www.wired.com/2015/11/signals-snowden-approved-phone-crypto-app-comes-to-android/`, 2015.

[5] Apple vs. the FBI: Google, WhatsApp, John McAfee and more are taking sides - LA Times. `http://www.latimes.com/business/technology/la-fi-tn-tech-response-apple-20160218-snap-htmlstory.html`, 2016.

[6] 504ENSICS LABS. Dalvik Inspector. `http://www.504ensics.com/automated-volatility-plugin-generation-with-dalvik-inspector/`, 2013.

[7] 504ENSICS LABS. LiME Linux Memory Extractor. `https://github.com/504ensicsLabs/LiME`, 2013.

[8] APOSTOLOPOULOS, D., MARINAKIS, G., NTANTOGIAN, C., AND XENAKIS, C. Discovering authentication credentials in volatile memory of android mobile devices. In *Collaborative, Trusted and Privacy-Aware e/m-Services*. 2013.

[9] ASHCROFT, J., DANIELS, D. J., AND HART, S. V. Forensic examination of digital evidence: A guide for law enforcement. *U.S. National Institute of Justice, Office of Justice Programs, NIJ Special Report NCJ 199408* (2004).

[10] BECHER, M., DORNSEIF, M., AND KLEIN, C. Firewire: all your memory are belong to us. *CanSecWest* (2005).

[11] BETZ, C. Memparser forensics tool. `http://www.dfrws.org/2005/challenge/memparser.shtml`, 2005.

[12] BUGCHECK, C. Grepexec: Grepping executive objects from pool memory. In *Proc. Digital Forensic Research Workshop* (2006).

[13] CARBONE, M., CUI, W., LU, L., LEE, W., PEINADO, M., AND JIANG, X. Mapping kernel objects to enable systematic integrity checking. In *Proc. CCS* (2009).

[14] CARRIER, B. D., AND GRAND, J. A hardware-based memory acquisition procedure for digital investigations. *Digital Investigation 1* (2004).

[15] CASE, A., CRISTINA, A., MARZIALE, L., RICHARD, G. G., AND ROUSSEV, V. FACE: Automated digital evidence discovery and correlation. *Digital Investigation 5* (2008).

[16] DOLAN-GAVITT, B., SRIVASTAVA, A., TRAYNOR, P., AND GIFFIN, J. Robust signatures for kernel data structures. In *Proc. CCS* (2009).

[17] GOOGLE, INC. Android dashboards - platform versions. `https://developer.android.com/about/dashboards/index.html`, 2015.

[18] GRUHN, M. Windows nt pagefile. sys virtual memory analysis. In *Proc. IT Security Incident Management & IT Forensics (IMF)* (2015).

[19] HALDERMAN, J. A., SCHOEN, S. D., HENINGER, N., CLARKSON, W., PAUL, W., CALANDRINO, J. A., FELDMAN, A. J., APPELBAUM, J., AND FELTEN, E. W. Lest we remember: cold-boot attacks on encryption keys. In *Proc. USENIX Security* (2008).

[20] HILGERS, C., MACHT, H., MULLER, T., AND SPREITZENBARTH, M. Post-mortem memory analysis of cold-booted android devices. In *Proc. IT Security Incident Management & IT Forensics (IMF)* (2014).

[21] JARRETT, H. M., BAILIE, M. W., HAGEN, E., AND JUDISH, N. Searching and seizing computers and obtaining electronic evidence in criminal investigations. *U.S. Department of Justice, Computer Crime and Intellectual Property Section Criminal Division* (2009).

[22] KOLLÁR, I. Forensic ram dump image analyser. *Master's Thesis, Charles University in Prague* (2010).

[23] KORNBLUM, J. D. Using every part of the buffalo in windows memory analysis. *Digital Investigation 4* (2007).

[24] LEE, J., AVGERINOS, T., AND BRUMLEY, D. TIE: Principled reverse engineering of types in binary programs. In *Proc. NDSS* (2011).

[25] LIN, Z., RHEE, J., WU, C., ZHANG, X., AND XU, D. DIMSUM: Discovering semantic data of interest from un-mappable memory with confidence. In *Proc. NDSS* (2012).

[26] LIN, Z., RHEE, J., ZHANG, X., XU, D., AND JIANG, X. SigGraph: Brute force scanning of kernel data structure instances using graph-based signatures. In *Proc. NDSS* (2011).

[27] LIN, Z., ZHANG, X., AND XU, D. Automatic reverse engineering of data structures from binary execution. In *Proc. NDSS* (2010).

[28] MACHT, H. Live memory forensics on android with volatility. *Friedrich-Alexander University Erlangen-Nuremberg* (2013).

[29] MEALY, G. H. A Method for Synthesizing Sequential Circuits. *Bell System Technical Journal 34*, 5 (1955), 1045–1079.

[30] MOVALL, P., NELSON, W., AND WETZSTEIN, S. Linux physical memory analysis. In *Proc. USENIX Annual Technical Conference, FREENIX Track* (2005).

[31] PETRONI, N., FRASER, T., MOLINA, J., AND ARBAUGH, W. Copilot - a coprocessor-based kernel runtime integrity monitor. In *Proc. USENIX Security* (2004).

[32] PETRONI JR, N. L., WALTERS, A., FRASER, T., AND ARBAUGH, W. A. FATKit: A framework for the extraction and analysis of digital forensic data from volatile system memory. *Digital Investigation 3* (2006).

[33] RICHARD, G. G., AND CASE, A. In lieu of swap: Analyzing compressed ram in mac os x and linux. *Digital Investigation 11* (2014).

[34] SALTAFORMAGGIO, B. Forensic carving of wireless network information from the android linux kernel. *University of New Orleans* (2012).

[35] SALTAFORMAGGIO, B., BHATIA, R., GU, Z., ZHANG, X., AND XU, D. GUITAR: Piecing together android app GUIs from memory images. In *Proc. CCS* (2015).

[36] SALTAFORMAGGIO, B., BHATIA, R., GU, Z., ZHANG, X., AND XU, D. VCR: App-agnostic recovery of photographic evidence from android device memory images. In *Proc. CCS* (2015).

[37] SALTAFORMAGGIO, B., GU, Z., ZHANG, X., AND XU, D. DSCRETE: Automatic rendering of forensic information from memory images via application logic reuse. In *Proc. USENIX Security* (2014).

[38] SCHUSTER, A. Searching for processes and threads in microsoft windows memory dumps. *Digital Investigation 3* (2006).

[39] SLOWINSKA, A., STANCESCU, T., AND BOS, H. Howard: A dynamic excavator for reverse engineering data structures. In *Proc. NDSS* (2011).

[40] SUN, H., SUN, K., WANG, Y., JING, J., AND JAJODIA, S. Trustdump: Reliable memory acquisition on smartphones. In *Proc. European Symposium on Research in Computer Security*. 2014.

[41] THE VOLATILITY FRAMEWORK. `https://www.volatilesystems.com/default/volatility`.

[42] THING, V. L., NG, K.-Y., AND CHANG, E.-C. Live memory forensics of mobile phones. *Digital Investigation 7* (2010).

[43] VIDAS, T. Volatile memory acquisition via warm boot memory survivability. In *Proc. Hawaii International Conference on System Sciences* (2010).

[44] WALLS, R., LEVINE, B. N., AND LEARNED-MILLER, E. G. Forensic triage for mobile phones with DEC0DE. In *Proc. USENIX Security* (2011).

[45] YANG, S. J., CHOI, J. H., KIM, K. B., AND CHANG, T. New acquisition method based on firmware update protocols for android smartphones. *Digital Investigation 14* (2015).

Appendix

A. Memory Image Acquisition

A prerequisite of memory forensics is the timely acquisition of a memory image from the subject device. Memory images typically contain a byte-for-byte copy of the entire physical RAM of a device or the virtual memory of an operating system or specific process(es). Traditionally, acquisition is performed by investigators, before the subject device is powered down, using minimally invasive software (e.g., fmem [22], LiME [7]) or hardware (e.g., Tibble [14], CoPilot [31]) tools. Other notable techniques have used the DMA-capable Firewire port [10] to acquire memory images, existing hibernation or swap files [18, 23, 32, 33], or cold/warm booted devices [19,20,43], but such approaches are only employed for highly specialized investigations. A more comprehensive list of memory image acquisition tools can be found in [2].

Android memory forensics was initially proposed during the development of memory acquisition tools for the devices. Most known among these are the software-based LiME [7] and TrustDump [40] techniques. In an alternative approach, Hilgers et al. [20] proposed cold-booting Android phones to perform memory forensics. Our evaluation of RetroScope used both LiME and a ptrace-based tool we developed (also available with the open source RetroScope code). Meanwhile, hardware-based memory acquisition from a mobile device is often performed via the ARM processor's JTAG port [1,45].