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

Brendan Saltaformaggio, Rohit Bhatia, Xiangyu Zhang, Dongyan Xu, Golden G. Richard III*

Purdue University

*University of New Orleans
A Crime To Investigate...

Before the investigation began, the suspect was interacting with their apps...

Without access to the suspect’s password or breaking Telegram’s fully encrypted storage!
Memory Forensics ... or Mission Impossible?
State of the Art: GUITAR - GUI Tree ARchaeology

[CCS ’15, Best Paper]

Remaining GUI Data Structures

Drawing-Content-Based Bipartite Graph Matching

GUI Tree

Alice

1

2

3
The “Screen 0” Limitation of GUITAR

In Memory GUI Data:

Time:
Are The Old Screens Really Gone?

App screen changes are **highly dynamic**

How can every screen be **fully rebuilt** so fast?

Some data must remain to bring the screens back
Are The Old Screens Really Gone? ... Yes and No

GUI Screen Data
GUITAR’s Target:
GUI Tree,
Draw Ops, ...

App Internal Data
Not for GUI drawing:
Raw Chat Strings,
Account Balance, ...

Are The Old Screens Really Gone?
... Yes and No
Android Asks The App To Draw A Screen

Android sends a **Redraw Command**

1) A **Canvas** is sent for the app to fill
   - Apps register `draw` routines with Android

2) The app builds **GUI structures** which “package” the internal data
   - Destroying the previous screen!

3) The filled canvas is **rendered** on the device’s screen
Idea: Ask The Memory Image To Draw A Screen

Challenges:

1) How to inject the Redraw Command?
   - Screen-specific draw routines

2) Need to understand the app internal data?
   - Data structure signature scanning
   - App-specific reverse engineering

3) Memory = Static Data
   - Execution context is gone

Our Goal: “Plug And Play” App-Agnostic Recovery
RetroScope: Spatial-Temporal Display Recreation

Performs app-agnostic screen reconstruction from an app’s internal data within a memory image.
Symbiont App: Two Apps In One

Step 1) Start the Symbiont App to host the memory image

Step 2) Move the memory image state into the Symbiont App
- Map memory segments
- Merge Java runtimes
- Register *draw* functions
Interleaved Re-Execution Engine

Step 3) Initialize the Interleaved Re-Execution Engine (IRE)

Formally modeled the interleaving of states as a finite automata

The Overly Simple Explanation:
Live Code outputs to Live Environment & Old Code reads from Old Environment

Transition rules guided by executing instruction semantics
Step 3) Initialize the Interleaved Re-Execution Engine (IRE)
Selective Reanimation

Step 4) **Redirect** a redraw command to the Target App

The IRE monitors the state transitions and corrects the execution
Selective Reanimation

Memory image app's `draw` routines naturally accesses its internal data.
Selective Reanimation

Memory image app’s draw routines naturally accesses its internal data.
IRE ensures that function calls to the new canvas are directed to the live GUI system.
Selective Reanimation

The newly filled Canvas is rendered by the live GUI system and saved.
Selective Reanimation

This process repeats for each registered draw routine
Breaking The Case Wide Open!
## Evaluation

15 Apps on 3 “Suspect” Devices: HTC One, LG G3, Samsung Galaxy S4

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Calendar</td>
<td>6</td>
<td>6</td>
<td>197316</td>
<td>732</td>
<td>102642</td>
</tr>
<tr>
<td>Chase</td>
<td></td>
<td></td>
<td>584587</td>
<td>2091</td>
<td>266965</td>
</tr>
<tr>
<td>Contacts</td>
<td></td>
<td></td>
<td>190847</td>
<td>723</td>
<td>71578</td>
</tr>
<tr>
<td>Facebook</td>
<td></td>
<td></td>
<td>382522</td>
<td>1451</td>
<td>95516</td>
</tr>
<tr>
<td>Gmail</td>
<td></td>
<td></td>
<td>235973</td>
<td>929</td>
<td>129804</td>
</tr>
<tr>
<td>Instagram</td>
<td>3</td>
<td>3</td>
<td>86829</td>
<td>433</td>
<td>42037</td>
</tr>
<tr>
<td>Messaging</td>
<td>4</td>
<td>4</td>
<td>93971</td>
<td>287</td>
<td>45085</td>
</tr>
<tr>
<td>TextSecure</td>
<td>7</td>
<td>8</td>
<td>231891</td>
<td>924</td>
<td>98571</td>
</tr>
<tr>
<td>WhatsApp</td>
<td>6</td>
<td>6</td>
<td>321229</td>
<td>1571</td>
<td>104216</td>
</tr>
</tbody>
</table>

Average of: 41,078 Byte-Code Instructions, 158 New Java Objects, and 13,535 New C/C++ Structures Per Screen.
Case 1: WeChat (And Others) Deleted Messages

From LG G3 Device
Case 2: WhatsApp Background Update

From Samsung Galaxy S4 Device
Related Works


Conclusion

RetroScope represents a new paradigm of **spatial-temporal** memory forensics for app GUI screens.

RetroScope’s novel IRE selectively reanimates an app’s screen redrawing functionality **without** any app-specific knowledge.

Recover visually accurate, temporally ordered screens (ranging from 3 to 11 screens) for a wide variety of **privacy-sensitive apps**.
Thank you!

Questions?

Brendan Saltaformaggio
bsaltafo@cs.purdue.edu
Privacy Implications of RetroScope?

The privacy-sensitive apps are not broken, per se
- Unlike disk or network, memory is assumed private
- Little incentive to “protect” memory
- E.g., Malware in your app’s memory = all bets are off

RetroScope is just emulating the standard behavior of Android
- To disrupt RetroScope would also hinder an app’s ability to draw screens
- Encrypting memory doesn’t work because RetroScope would reanimate the decryption logic
- Privacy vs. Usability
  - E.g., Zeroing data would require getting it back in order to redraw (slowing down the UI)

Citizens’ privacy is protected by strict legal protocols and regulations (see [9,21])
- Search warrants & strict chain of custody documentation prior to performing “invasive” forensics