GUITAR: Piecing Together Android App GUIs from Memory Images

Brendan Saltaformaggio, Rohit Bhatia, Zhongshu Gu, Xiangyu Zhang, Dongyan Xu
Department of Computer Science and CERIAS
Purdue University, West Lafayette, IN 47907
{bsaltafo, bhatia13, gu16, xyzhang, dxu}@cs.purdue.edu

ABSTRACT

An Android app's graphical user interface (GUI) displays rich semantic and contextual information about the smartphone's owner and app's execution. Such information provides vital clues to the investigation of crimes in both cyber and physical spaces. In real-world digital forensics however, once an electronic device becomes evidence most manual interactions with it are prohibited by criminal investigation protocols. Hence investigators must resort to "image-and-analyze" memory forensics (instead of browsing through the subject phone) to recover the apps' GUIs. Unfortunately, GUI reconstruction is still largely impossible with state-of-the-art memory forensics techniques, which tend to focus only on *individual in-memory data structures*. An Android GUI, however, displays diverse visual elements each built from numerous data structure instances. Furthermore, whenever an app is sent to the background, its GUI structure will be explicitly deallocated and disintegrated by the Android framework. In this paper, we present GUITAR, an app-independent technique which automatically reassembles and redraws all apps' GUIs from the multitude of GUI data elements found in a smartphone's memory image. To do so, GUITAR involves the reconstruction of (1) GUI tree topology, (2) drawing operation mapping, and (3) runtime environment for redrawing. Our evaluation shows that GUITAR is highly accurate (80-95% similar to original screenshots) at reconstructing GUIs from memory images taken from a variety of Android apps on popular phones. Moreover, GUITAR is robust in reconstructing meaningful GUIs even when facing GUI data loss.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection

General Terms

Security

Keywords

Memory Forensics; Android; Digital Forensics

CCS'15, October 12–16, 2015, Denver, Colorado, USA.

DOI: http://dx.doi.org/10.1145/2810103.2813650.

1. INTRODUCTION

The graphical user interface (GUI) of an application renders semantic information (e.g., text, images, and graphics) for human users to interact with. Further, GUIs often reflect our only perception of an application's execution state. This is even more true for the GUIs of Android apps, which users interact with — one at a time — on the smartphone's screen while numerous other apps run in the background. Moreover, smartphone apps are long-running (compared with their desktop counterparts) as users seldom terminate an app explicitly, and the apps keep running even with the screen turned off or in "airplane" mode. Now imagine the following digital forensics scenario: Law enforcement agents obtain a suspect's smartphone which they believe can reveal vital evidence for their investigation. Ideally, investigators would inspect the GUIs of the apps, specifically those *not* currently on screen, for evidence to review, catalog, and later present in court.

It turns out that this is far more difficult than it appears *for both policy and technical reasons.* Due to strict legal interpretations of "digital evidence preservation" in US court proceedings [2–10], once an electronic device becomes a piece of raw evidence, most manual interaction with it (e.g., browsing through a smartphone's screen) is prohibited by US DOJ, American Law Reports, and other's investigation protocols [1,14,16,20,27]. Moreover, if the app requires a password login every time it is brought to the foreground (see the case study in Section 4.2.2), then its earlier GUI could not be restored even if operating the phone were allowed.

To overcome this, modern digital investigators now rely on memory forensics. With a search warrant, investigators can capture the phone's memory image, using certified minimally intrusive tools [21, 44], which will be analyzed in the forensics lab without fear of jeopardizing the investigation. Therefore, the most desirable outcome of this analysis would be the recovery of the GUIs that the suspect was interacting with — revealing the evidence stored on the device.

Despite recent advances in computer memory forensics, GUI recovery remains largely impossible. Specifically, nearly all state-of-the-art memory forensics techniques [19, 24, 29, 30, 37, 39–41, 45] focus on the recovery of *individual data structures.* Given a memory image and a data structure of interest, existing techniques (e.g., [24, 29, 30, 37, 38, 41, 45]) rely on signature-based scanning of the memory image to locate raw in-memory instances of that data structure. To render a discovered data structure instance in human per-

ceivable format, a recent solution [40] derives that structure's rendering logic from the application it belongs to.

Unfortunately, an Android app GUI is much more complex than an individual data structure — it is a virtual "billboard" of many diverse, application-specific data objects with geometric and semantic dependencies defined by each individual app. As detailed in Section 2, a GUI is internally represented as a tree whose structure and nodes change dynamically at runtime. More significantly, whenever an app is backgrounded (i.e., replaced on the phone's screen by a newly in-focus app), Android will *explicitly nullify* many key pointers in the tree, effectively disintegrating the GUI. As such, existing data structure-oriented memory forensics techniques can only identify the GUI's "element" data structures from the memory image (i.e., "identifying the puzzle pieces"). But they cannot reassemble the elements (hundreds or even thousands of them) into the original GUI or further visually redraw the GUI (i.e., "putting the puzzle pieces together").

Here the new challenge is analogous to that faced by an archaeologist who tries to piece together an ancient fresco or pottery (the GUI) from its unearthed fragments (data structures) [25]. To address this challenge, we present GUITAR[1], a system which automatically reconstructs app GUIs from Android phone memory images and redraws them as they originally appeared. Interestingly, GUITAR does *not* require app-specific knowledge and hence can *reconstruct any Android app's GUI generically*. Unlike existing techniques, GUITAR presents investigators with the "same view" of the suspect's app(s) rather than individual data structure instances. For example, for an instant messaging app, GUITAR will reconstruct its GUI with contents (e.g., contacts, messages, timestamps, etc.) all in their original layout.

GUITAR targets the low-level GUI framework defined by the Android graphical windowing system library (analogous to X11 commonly used with Linux), which is common to all apps' implementation. Given the GUI element data structure instances, GUITAR employs a *depth-first topology recovery algorithm* to reconstruct the app's graphical layout hierarchy. Next, graphical GUI contents are remapped to the geometric layout using a *bipartite graph weighted assignment solver* and corresponding *drawing-content based fitness function*. Finally, GUITAR recreates the runtime environment to redraw the GUI using an *unmodified* Android windowing system binary, and outputs the app's redrawn GUI as it would have appeared *had it been displayed on-screen when the memory image was taken*. If present in a memory image, GUITAR can recover *previous GUI constructs*, allowing investigators to see some previous GUI state of the same app.

Our evaluation, performed with memory images taken from a number of popular Android apps on three new Android smartphones, shows that GUITAR is able to reconstruct and redraw entire app GUIs with very high accuracy. We use Content Based Image Recognition (CBIR) to measure the visual similarity between GUITAR-reconstructed GUIs and screenshots taken from the original app, and GUITAR scores 80-95% (high similarity) in all cases. Further, our evaluation shows that GUITAR is adaptive and robust for reconstructing partial, meaningful GUIs when faced with GUI data loss over time.

[1]GUITAR stands for "GUI Tree ARchaeology."

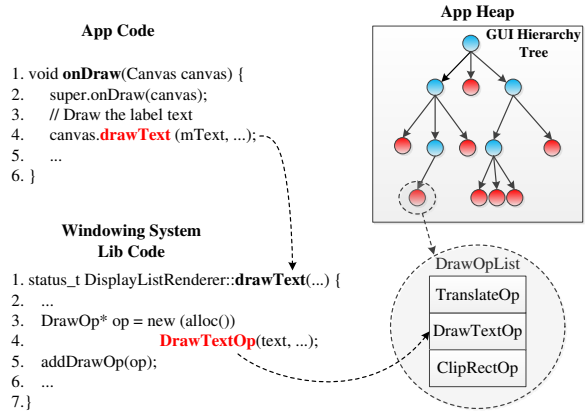


Figure 1: Overview of a Windowing System Library. Each app maintains a GUI tree, with each leaf node containing Drawing Operations. Changes to the app's GUI are reflected by changes to the GUI tree.

2. THE ANDROID GUI FRAMEWORK

The Android platform exhibits many features which inherently pose challenges to GUI reconstruction, and these motivate many of our design decisions in Section 3. For example, we originally considered recovering the pixel buffer to which the Android windowing system projects the entire GUI for on-screen display. However, this approach turned out to be infeasible because that buffer is located in the graphics card driver's memory which is quickly deallocated and reused when an app is backgrounded. Thus, recovering this buffer yields only the *currently* visible app's GUI.

Seeking an alternative solution, we instead target a much more robust in-memory artifact of the windowing system: *the GUI hierarchy tree* ("GUI tree" for short) with *drawing operations* ("draw ops" for short). Figure 1 illustrates how draw ops are organized in a GUI tree. The Android windowing system library included in each graphical app maintains a GUI tree to represent the GUI's current geometric layout and graphical content. Further, despite the vast variety of visually different apps, such a tree generically represents each app's visual presentation and display.

The GUI tree resides in each app's heap. Each node in the GUI tree (called a "TreeNode") contains a pointer to a list of draw ops (a "DrawOpList") which describes a portion of the screen space. A parent TreeNode points to a DrawOpList which contains pointers to child TreeNodes; whereas a leaf TreeNode points to a DrawOpList which contains actual draw ops with graphical content.

When an app invokes the windowing system's drawing functions (e.g., `drawText(...)` shown in Figure 1), the GUI modifications will be converted into an array of draw ops (`TranslateOp`, `DrawTextOp`, `ClipRectOp`) and stored in a leaf TreeNode. A single drawing function may create multiple draw ops and store them in one or more leaves. Thus, whenever the app is visible, a GUI tree of parent TreeNodes (describing relative geometric positions and screen layout hierarchy) and leaf TreeNodes (containing actual graphical draw ops) will be created in the process' heap memory.

However, Android always tries to save memory, and when an app is backgrounded its GUI tree will be *deallocated* and critical pointers within it (in particular the ones from TreeNodes to their DrawOpLists) will be set to `NULL` (a good programming practice, but bad for memory forensics). This

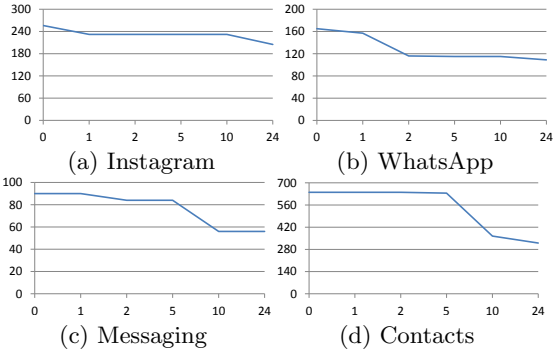


Figure 2: Number of recoverable GUI data structures of backgrounded apps over 24 hours.

effectively disintegrates the tree and, by doing so, makes its reconstruction challenging.

We profiled several Android apps' memory use before and after backgrounding and found that the GUI tree deallocation is among the last operations an app will perform, because background apps cannot receive user input. In fact, many nodes of the old GUI tree remain in the app's free heap space until the app is returned to the foreground, when an entirely new GUI tree will be built. Further, if the app is *not* returned to the foreground for some time, we observe that a non-trivial portion of the GUI data is still recoverable (i.e., their heap space is not reallocated and overwritten). Figure 2 shows measurements of 4 apps' GUI tree data structures after those apps were left in the background for a period of 24 hours. For this experiment, we took an initial measurement at time $t0$, backgrounded the app, then took measurements at times $t0+1$ (hour), $t0+2$, $t0+5$, $t0+10$, and $t0+24$. The smartphone (LG G3) belongs to one of the authors, with all other apps (except the one profiled) heavily used during that period.

From Figure 2 we can make a few key observations: First, although some apps have background activities (which reallocated the free heap space), a large amount of GUI data is recoverable even after 24 hours. In fact our evaluation in Section 4 shows that once an app is backgrounded 43% to 98% of its GUI data remains intact, which is sufficient to redraw the GUI — either completely or partially. Another key observation is that, since each node describes a small portion of the screen, any missing nodes do not affect the redrawing of the remaining GUI. In Section 4 we present a number of case studies demonstrating how missing internal nodes or leaf nodes may cause slight visual variations to reconstructed GUIs, which still retain reasonable appearance.

2.1 Challenges and Solution Overview

Firstly, because key pointers in the GUI tree are explicitly nullified, the GUI's original layout needs to be pieced together from the many disconnected nodes. GUITAR defines a *depth-first topology recovery algorithm* (Section 3.1) to reconnect *internal* parent nodes to their DrawOpLists and hence to their children nodes. Complicating the recovery, GUITAR often encounters old or partially destroyed nodes which appear to be valid children of the parent nodes, and GUITAR must automatically identify (and later remove) such *conflicting branches* in the tree.

Secondly, the GUI's graphical contents need to be restored by geometrically re-mapping the *leaf* TreeNodes back to their DrawOpLists. GUITAR leverages semantic hints in the *drawable graphical content* described by each DrawOpList. More formally, such leaf mapping can be reduced to a *bipartite graph weighted assignment problem*. GUITAR uses the drawable GUI content to build a *drawing-content-based fitness function* (Section 3.2) which computes the likelihood that each leaf matches to some graphical content.

Finally, several key data structures' functional inheritance, which is necessary for GUI redrawing, is lost in the memory image. GUITAR employs a technique called *forced polymorphism* (Section 3.3) to patch the lost inheritance information. Then, GUITAR recreates the GUI redrawing runtime using an *unmodified* Android windowing system library binary, which will redraw the reassembled GUI tree, as it would have appeared in the foreground of the original phone.

3. GUITAR DESIGN

The input to the GUITAR technique is a set of data structure instances corresponding to draw ops, TreeNodes, and graphical content elements, recovered from the subject app's memory image. For self-containedness, GUITAR's implementation includes a linear brute-force memory image scanner with 248 distinct signatures of data structures, defined by Android's windowing system and stable across all Android versions we tested. This signature set can be easily updated with any future changes to those data structures. Alternatively, the recovery can be done using any existing or future memory forensics techniques (e.g., those cited in Section 1). Note also, that because many of the target data structures have been deallocated, it is possible that some instances are partially corrupted (overwritten). To avoid complications from corrupted structures, GUITAR's data structure signature matching entails checks on every field needed to reconstruct the GUI, and if an object is partially broken then it will be conservatively discarded. Interested readers are directed to Appendix A for more information about GUITAR's data structure signatures.

3.1 Reconstructing GUI Tree Topology

Once the GUI "elements" (data structures) are recovered, GUITAR will reconstruct the GUI tree. This section details how GUITAR reconnects the tree's parent TreeNodes to their children. However, recall that key pointers are set to `NULL` when the app is backgrounded, causing each TreeNode to lose connection to its DrawOpList. As Figure 3 shows, losing the node's connection to its DrawOpList also breaks any connection to its children. Further, these parent-to-child links are the only ones that the windowing system binary follows to redraw the GUI. Thus, GUITAR must first recover the GUI tree's parent-to-child structure from two sets of disconnected TreeNodes and DrawOpLists (Figure 3).

The GUI's layout semantics provide a valuable hint toward solving this problem. We observe that, in addition to the GUI tree, several Java objects encode a "reverse" GUI layout (i.e., which layers are drawn in front of other layers). By traversing these Java objects, the *parent* of each TreeNode can be reached. Unfortunately, these additional structures are unusable during GUI redrawing (which only uses TreeNodes and DrawOpLists), but GUITAR can leverage these "child to parent" paths and the "DrawOpList to child" pointers (shown in Figure 3) to recover the "parent to DrawOpList" pointers.

However, an issue arises during topology recovery: *conflicting branches* may be introduced to the reconstructed

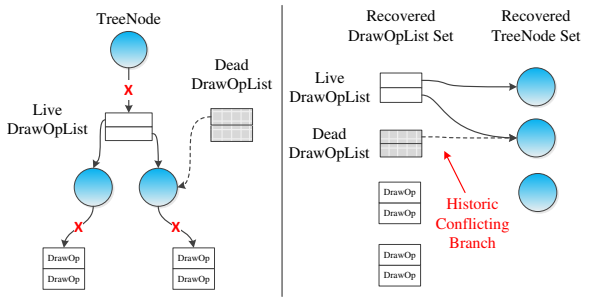


Figure 3: Broken tree structure due to nullified pointers (marked with a red X). GUITAR must rebuild the tree from the recovered TreeNodes and DrawOpLists. Note the conflicting branch introduced by historic data structures.

GUI tree when DrawOpLists or TreeNodes from *previously drawn* GUI screens remain (not overwritten) in memory after those portions of the GUI have been modified. Essentially, these old structures correspond to historical portions of the GUI which were rendered, changed, and replaced with new DrawOpLists and TreeNodes before the app was backgrounded. Figure 3 shows an example: a historic DrawOpList is recovered and introduces a conflict into an otherwise valid parent TreeNode. At this point, GUITAR cannot distinguish between the most recent versus older TreeNodes and DrawOpLists, but GUITAR will mark the conflict while mapping both DrawOpLists to the parent TreeNode in Figure 3. Similarly, old TreeNodes can cause conflicts with a parent TreeNode (via a historic Java object's encoding) which has already been updated with new children. Conflicting branches will be removed later by leveraging characteristics of the visual GUI content (Section 3.2). Our evaluation in Section 4 shows that conflicting branches occur for only a small number of nodes. Notably, we did find one case where a full conflicting branch (all parent and child TreeNodes and DrawOpLists) was recovered, leading to two drawable GUI versions: one shows the most recent view and the other shows elements of a prior view.

GUITAR's depth-first tree topology recovery algorithm (Algorithm 1) uses a pre-order depth-first traversal of the recovered TreeNodes to rebuild the GUI hierarchy. The recursive algorithm starts at the tree's root. Given a parent TreeNode, GUITAR first locates all TreeNodes which have that TreeNode as a parent. Then GUITAR searches

Algorithm 1 Depth-First Tree Topology Recovery

Input: TreeNode Set N, DrawOpList Set D
Output: GUI Tree $T = (V, E)$

```
procedure MapNode(node)
    for other ∈ N do
        if other ⇝ node then          ▷ ⇝: child-to-parent path exists
            for list ∈ D do  ▷ Find DrawOpLists pointing to other
                if other ∈ list.points_to then
                                                  ▷ Map list to node
                    node.children ← node.children ∪ other
                    node.opsLists ← node.opsLists ∪ list
                    if |node.opsLists| > 1 then
                        node.conflict ← True      ▷ Mark the conflict
    for child ∈ node.children do
        MapNode(child)                            ▷ Continue recursion
end procedure

for node ∈ N do
    if node.parent = ∅ then                  ▷ Start at the root nodes
        MapNode(node)
```

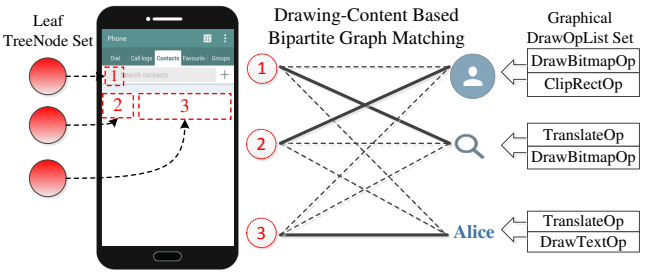


Figure 4: Example of drawing-content based bipartite graph matching.

each DrawOpList for those which point to any child of the parent. If any are located, then these DrawOpLists must belong to this parent, because they point to the parent's children. A conflicting branch is identified if this search returns more than one DrawOpList. The algorithm matches the located DrawOpLists to the parent TreeNode, and continues the recursion with only those children pointed to by the DrawOpLists. The recursion will stop when a leaf (i.e., a node that is not any other node's parent) is reached.

3.2 Remapping Drawing Operations

Having reconstructed the GUI tree's internal structure, GUITAR must now map the *leaf* TreeNodes to the remaining DrawOpLists. Note that these DrawOpLists contain only graphical content (such as created by `drawText` in Figure 1), unlike those pointed to by non-leaf nodes. Figure 4 illustrates our key intuition of matching leaf TreeNodes to the DrawOpLists' drawable GUI content. First, GUITAR computes the geometric screen area described by each leaf TreeNode. Based on this, GUITAR finds a best global match to the drawable GUI content that fits in that screen area. Formally, we define a *drawing-content-based fitness function* to compute the fit between any leaf TreeNode and DrawOpList's graphical content. We then reduce the problem of mapping DrawOpLists to leaf TreeNodes to a *weighted assignment problem*. Problems of this class can be solved in polynomial time when modeled as a weighted bipartite graph matching problem. Thus, GUITAR must first set up a multi-source multi-sink bipartite graph with the graphical DrawOpLists as a source vertex set and the leaf TreeNodes as the sink vertex set.

Building a Weighted Bipartite Graph Algorithm 2 shows how GUITAR builds the weighted bipartite graph. For each DrawOpList, GUITAR computes the maximum dimensions $(ops_{width}, ops_{height})$ of the graphical output produced by those draw ops (i.e., the pixels to be drawn on screen). Next, GUITAR must compute the screen area described by each leaf TreeNode, by subtracting the leaf's screen coordinates (x_{leaf}, y_{leaf}) from the closest neighboring leaves' coordinates. However, each leaf only describes its coordinates relative to its parent TreeNode. Thus, to find the neighboring leaves and compute each leaf's true (full screen) coordinates, GUITAR must look backwards through the tree's hierarchy. This is performed by a recursive function summarized by the $getNeighbors$ and $getFullCoord$ functions in Algorithm 2. After finding the two closest neighboring leaves' coordinates (x_{below}, y_{below}) and (x_{right}, y_{right}), the current leaf's dimensions are computed as shown in the second loop of Algorithm 2.

Algorithm 2 Building Draw-Content-Weighted Bi-graph

Input: Leaf Set L, DrawOpList Set D, ScalingFactor f
Output: Graph $G = (V_{leaves}, V_{opLists}, E)$, $maxWeight$

```
V_leaves ← ∅
V_opLists ← ∅
E ← ∅
maxWeight ← 0
for ops ∈ D do                                  ▷ Compute DrawOpList dimensions
    (ops_width, ops_height) ← computeDrawSize(ops)
    ops.width ← ops_width
    ops.height ← ops_height
    V_opLists ← V_opLists ∪ ops                 ▷ Insert DrawOpList vertex

for leaf ∈ L do                                 ▷ Compute leaf dimensions
    right, below ← getNeighbors(leaf)
    (x_right, y_right) ← getFullCoord(right)
    (x_below, y_below) ← getFullCoord(below)
    (x_leaf, y_leaf) ← getFullCoord(leaf)
    (leaf_width, leaf_height) = (x_right − x_leaf, y_below − y_leaf)
    leaf.width ← leaf_width
    leaf.height ← leaf_height
    V_leaves ← V_leaves ∪ leaf                  ▷ Insert leaf vertex

for ops ∈ V_opLists do
    for leaf ∈ V_leaves do                      ▷ Compute edge weights
        d_width ← leaf.width − ops.width
        d_height ← leaf.height − ops.height
        if d_width < 0 or d_height < 0 then
            scale ← f                           ▷ Scale factor for over-drawing
        else
            scale ← 1.0
        weight ← scale * (√((d_width)^2 + (d_height)^2))
        E(ops, leaf) ← weight                   ▷ Insert edge weight
        if weight > maxWeight then
            maxWeight ← weight                  ▷ Update max weight
```

Note that leaves and their DrawOpLists often *do not have the same dimensions.* It is possible that a DrawOpList draws graphics smaller or larger than its leaf's dimensions. To account for this, the dimensions of each leaf and each DrawOpList are compared using Euclidean distance. A scaling factor is used to make the comparisons favor under-drawing to over-drawing (i.e., it is more likely that the draw ops draw something smaller than the leaf rather than larger). The scaling factor is configurable (input f in Algorithm 2), and in our evaluation a scale of 1.3 resulted in the best mappings. The resulting weights are assigned to the bipartite graph edges in the final loop of Algorithm 2, and we update a maximum weight variable to be used later.

Solving the Assignment Unfortunately, the resulting graph is not suitable for assignment solving because GUITAR will likely recover an unequal number of DrawOpLists and leaf TreeNodes. However, weighted assignment solving algorithms require the bipartite graph to be balanced (i.e., |source vertices| = |sink vertices|) and complete (i.e., edge set = source vertices × sink vertices). Typically, this is solved by adding fake vertices to the smaller half of the bipartite graph, but this would allow GUI elements to go unmatched or be matched to fake leaves. Instead, GUITAR aims to redraw the most complete GUI possible by finding the most valid matches.

To overcome this, we build upon two key observations: In the case that GUITAR recovers more DrawOpLists than leaf TreeNodes, we can be sure that at least one leaf has a conflict (like before, a conflict is a TreeNode with two or more DrawOpLists). In this case, we want to allow some TreeNodes *to map to multiple DrawOpLists* to preserve as much graphical data as possible.

In the case that GUITAR recovers more leaf TreeNodes than DrawOpLists, we observe that adding fake DrawOpList vertices will not harm the resulting GUI because they will represent "empty space" where no leaf mapping could be found. However, GUITAR must only consider mapping a fake DrawOpList if no real DrawOpList remains mappable (all real DrawOpLists have been assigned), simply put: try to draw as many DrawOpLists as possible even if some are over-drawn.

Algorithm 3 Correcting Bipartite Graph and Mapping

Input: Graph $G = (V_{leaves}, V_{opLists}, E)$, $maxWeight$
Output: Matched Graph G

```
while |V_leaves| < |V_opLists| do
    for leaf ∈ unique(V_leaves) do
                                    ▷ Duplicate all the unique leaf vertices
        newLeaf = copy(leaf)
        V_leaves ← V_leaves ∪ newLeaf
        for ops ∈ V_opLists do
            E(ops, newLeaf) ← E(ops, leaf)
                                    ▷ Duplicate edge weights

while not|V_opLists| = |V_leaves| do
    fakeOps ← new FakeOpList
                                    ▷ Add fake DrawOpLists to balance G
    V_opLists ← V_opLists ∪ fakeOps
    for leaf ∈ V_leaves do
        E(fakeOps, leaf) ← maxWeight + 1

KuhnMunkresAlgo(G)
```

Algorithm 3 builds the balanced and complete bipartite graph and performs the weighted assignment. In the first loop, GUITAR checks if we have recovered fewer leaf TreeNodes and, if so, repeatedly duplicates the leaf vertices until there are more leaf vertices than DrawOpList vertices. Note that GUITAR also copies the corresponding edge weights so that each duplicate of a TreeNode vertex has an equal likelihood of mapping to the same DrawOpList. Next, in the second loop, GUITAR adds fake DrawOpList vertices until the graph is balanced. For each fake DrawOpList vertex, GUITAR adds edges to every leaf vertex with weight equal to $maxWeight + 1$ (calculated in Algorithm 2). Using $maxWeight + 1$ edge weights ensures that the fake DrawOpLists are only considered for mapping after all real DrawOpLists (with lower, more favorable edge weights) have been mapped. After this step, the bipartite graph is balanced and complete — allowing any leaf TreeNode to map to any DrawOpList per their minimal edge weights.

The weighed bipartite graph assignment solving algorithm is represented in Algorithm 3 as the $KuhnMunkresAlgo$ function. The Kuhn-Munkres algorithm (also known as the Hungarian method) solves the weighted assignment problem in polynomial time. Our implementation uses an open-source version of the algorithm with time complexity $O(n^3)$ [11]. The Kuhn-Munkres algorithm takes the bipartite graph (i.e., two disjoint, balanced vertices sets and the complete edge set) as input. Internally, the algorithm maintains an adjacency matrix representing the weights of the complete edge set. The matrix values are iteratively reduced by the balanced cost (i.e., edge weight) of the minimum weight edges. Thus, at the end of each iteration the lowest weight edges will have cost 0. The iteration continues (balancing by the minimum weights and reducing) until: for each source vertex, the weight of one edge to a distinct sink vertex reduces to 0 (i.e., at least one 0 value in each row and column). The algorithm outputs the edge set which matches every source vertex to a distinct sink vertex with the minimum possible combined edge weights. To GUITAR, this edge set represents the global best mapping of the DrawOpLists' visual content to the leaf TreeNodes' geometric area on screen.

At this point, any mappings to fake DrawOpList vertices are removed and those leaf TreeNodes are marked as empty space on the resulting GUI. Now, GUITAR can remove conflicting branches based on two criteria leveraging the mapped visual GUI content: If a branch 1) has a dead end (i.e., TreeNodes without mapped DrawOpLists) or 2) describes a visual portion of the screen that is covered by a more complete branch with overlapping DrawOpList mappings. To ensure visual GUI data is not removed, GUITAR ensures that the DrawOpLists of leaf TreeNodes marked for removal are mapped to a different branch of the tree (even if that requires adding a new branch). In practice a conflicting branch is rarely mapped to any valid (not fake) DrawOpList.

3.3 Runtime Recreation for GUI Redraw

Once the GUI tree has been reconstructed, GUITAR has everything needed to redraw the app GUI, but a few challenges still remain. First, the majority of the GUI drawing functionality is invoked via *inherited methods* in the C++ GUI objects. Since these objects are recovered from a static memory image, the functional inheritance has been broken. GUITAR recreates this inheritance via a technique called *forced polymorphism*. Second, after recreating the polymorphism, the GUI tree needs to be grafted into a live "host tree" which will be redrawn by Android's windowing system.

Runtime Setup for Redrawing To preserve the interconnection between the recovered GUI data, GUITAR first maps the recovered data structures back to their original locations (i.e., the addresses they occupied when the memory image was taken) in the memory of the Android emulator[2]. This ensures that Android's windowing system — without modification — can follow any *data pointers* needed to redraw the GUI. Note that we map neither any additional data nor code segments from the memory image into the live memory. This makes GUITAR applicable to memory images from any Android device without concern about vendor-customizations.

Forced Polymorphism Many of the GUI data structures are polymorphic, and when inherited methods are invoked against these objects, dynamic function pointer tables are consulted to determine which implementation of an inherited function should be invoked. Unfortunately for recovered objects from a memory image, these function dispatch tables are unusable because the values in those tables are highly sensitive to each execution of the application. This situation is further confounded by ASLR present on modern Android devices (i.e., functions pointed to by the dispatch tables will be at random addresses in the memory image). Further, recovering both the object's *code and data* from the memory image would require GUITAR to handle a significant number of inconsistencies between the old (frozen) execution environment and the new one — making GUITAR a less portable and more heavy-weight solution.

To overcome this, we have developed a technique called *forced polymorphism* to force the "recovered objects" to inherit from newly allocated "live objects." GUITAR must rebuild the recovered objects' function dispatch tables to allow the windowing system to invoke any inherited drawing functions. However, due to lack of type and symbol information[3] in the memory image and the multiple-inheritance used by these objects, GUITAR must first determine the true runtime type of each recovered object.

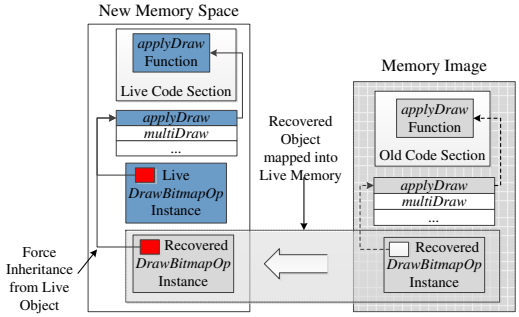


Figure 5: Illustration of forced polymorphism.

GUITAR leverages the GUI data structure signatures to guide the forced polymorphism. For each recovered object, GUITAR recalls the object-type recognition performed during memory image scanning. During scanning, many objects are recovered based on their common superclass. To identify the recovered object's true type inheritance, GUITAR compares the object to signatures from every object along that object's inheritance tree. The *deepest* matching subclass is then marked as the object's previous runtime-type, which GUITAR uses to reconnect the function dispatch table.

Based on the recovered object's true inheritance, GUITAR allocates a new instance of the matching type (a live object). GUITAR then redirects the recovered object's function dispatch table to that of the live object. Now, when the Android windowing system attempts to invoke an inherited function from one of the recovered objects, it will be redirected to the correct function in the current address space. This also avoids any complications from ASLR present in the memory image, because the function's old location is abandoned and corrected to the live location. Figure 5 shows an example of a recovered `DrawBitmapOp` being forced to inherit a live `DrawBitmapOp` function dispatch table. Notice that when the inherited `applyDraw` function is invoked, the lookup consults the live function dispatch table but the recovered object's data (e.g. GUI content) is preserved.

GUI Redraw Once the recovered structures' functional inheritance has been recreated, the reconstructed GUI is ready to be redrawn. Because the recovered objects have been mapped back to their original memory locations, the windowing system code can interact with them seamlessly, without any instrumentation for address translation.

GUITAR is prepackaged with unmodified Android windowing system binary code and a minimal Android app GUI, used as a "host" for grafting the recovered GUI tree. When redrawing the GUI tree, GUITAR inserts the entire recovered GUI tree as a subtree within the running host app's GUI. GUITAR then marks the tree as "dirty," causing the windowing system to redraw the GUI content. At this point the windowing system executes unsuspectingly, accessing the recovered GUI data as if it had naturally been allocated and initialized in the new process. The GUI content is displayed as it would have appeared on the original device's screen the last time that app was in focus. The newly drawn GUI then replaces the host app's GUI.

[2]This mapping is done using a newly started "stub" process in the emulator, before any heap or data segments are allocated, to avoid conflicts with new "live" memory usage.

[3]Android devices are shipped with stripped versions of all system binaries, including the windowing system library.

Device	App	Foreground Instances	Background Instances	% Persists	Recovered by GUITAR
Samsung S4	Calendar	546	507	92.86	507
	Chase Banking	221	168	76.01	168
	Contacts	511	476	93.15	476
	Facebook	655	634	96.79	634
	Instagram	262	240	91.60	240
	Messaging	120	102	85.00	102
	WhatsApp	172	148	86.05	148
LG G3	Calendar	753	738	98.00	738
	Chase Banking	220	172	78.18	172
	Contacts	731	640	87.55	640
	Facebook	926	884	95.46	884
	Instagram	301	259	86.05	259
	Messaging	101	90	89.11	90
	WhatsApp	214	165	78.97	165
HTC One	Calendar	276	259	93.84	259
	Chase Banking	191	170	89.01	170
	Contacts	358	285	79.60	285
	Facebook	608	593	97.53	593
	Instagram	355	319	89.86	319
	Messaging	392	371	94.64	371
	WhatsApp	130	123	94.61	123

Table 1: Recovery of backgrounded GUI data structures.

4. EVALUATION

We have implemented GUITAR as a plug-in for the Android emulator (~2000 lines of C++ code). GUITAR takes a subject Android device's memory image as input and redraws the recovered app GUIs on the emulator's screen. GUITAR requires no modification to the Android framework code but leverages the (open-source) data structure definitions of its windowing system.

Experimental Setup We used three Android smartphones as "suspect devices": an HTC One, Samsung Galaxy S4, and LG G3. The devices are all different OEM customized versions of Android 4.4[4]. We first installed a variety of apps on all 3 devices, and one of the authors interacted with each to cause several GUIs to be displayed and changed. Among these were 2 of the most popular social networking apps: Facebook and Instagram, whose GUIs reveal significant personal information about the device's owner, friends, and activities. We also evaluated WhatsApp, a widely used chat and instant messaging app, to reveal a suspect's recent conversations and contact list. Also 3 *vendor-specific* apps (Calendar, Contacts, and Messaging) were tested, each of which is implemented by smartphone vendors specifically for their devices with vastly different GUI constructs.

4.1 GUI Data Elements (Puzzle Pieces)

As stated in Section 2, most GUI data structures are freed and key pointers nullified when an app is backgrounded. In this section, we evaluate how many of these data structure instances persist in the app's free heap space after being backgrounded. For these tests, we interacted with each app, backgrounded it, and waited 15 minutes while interacting with another app, before capturing memory images from the app *while it was in the background*. Our results will be leveraged in the next subsections to connect the quantity of recovered data structures to the quality of the redrawn GUIs.

To establish ground truth, we instrumented each app to log allocations and deallocations of the GUI data structures[5]. When a data structure instance was deallocated, we also logged its contents, allowing us to verify which freed instances had been overwritten (fully or partially). We then analyzed the log to identify how many GUI data structures existed *before* and *after* the app was backgrounded. Finally, we tested GUITAR with each memory image to ensure that all remaining valid data structures could be located. Note that GUITAR has no knowledge of our profiling results and relies only on signature-based scanning for data structure recovery.

Table 1 presents the results for all 7 apps on each of the 3 devices. The devices and app names are listed in Columns 1 and 2, respectively. Column 3 shows the count of GUI data structure instances that were in the app's heap when the app was in the foreground. Column 4 shows those which remained in the app's heap after the app was backgrounded, and Column 5 shows this as a percentage. Lastly, Column 6 presents the number of data structure instances which GUITAR recovered from the memory image.

From Table 1, we make several observations: First, GUIs are built from a significant number of data structure instances. This may seem intuitive, but it confirms our earlier claim that *focusing on individual data structures is insufficient*. Many apps require more than 500 data structure instances for their GUIs. Notably, the LG G3 Facebook app reports the most data structures: 926. Overall, 11 of the 21 test cases have more than 300 data structure instances each. Table 1 also shows that *vendor-specific* apps show very different results on each smartphone. For example, each vendor's Calendar app has very different GUI construction and thus contains a disparate number of data structures: 546 for Samsung, 753 for LG, and 276 for HTC. In contrast, *vendor-generic* apps tend to have very similar results (e.g., 262, 301, and 355 for Instagram).

[4]To handle different Android versions, GUITAR only needs to update its data structure signatures for memory image scanning (if those versions change any GUI object definitions).

[5]This was done via in-place binary instrumentation of the windowing system library and, by design, neither interacts with any memory management components nor changes how the structures are used by the library.

Device	App	Ground Truth Tree Size	Recovered Tree Size	Edit Distance	% Original	CBIR Similarity
Samsung S4	Calendar	62	57	22	64.51	85.05
	Chase Banking	33	33	4	87.88	N/A
	Contacts	50	57	11	92.00	94.64
	Facebook	85	113	48	77.65	95.47
	Instagram	54	57	6	94.44	89.14
	Messaging	22	22	2	90.91	85.75
	WhatsApp	30	30	4	86.67	82.17
LG G3	Calendar	79	76	18	77.22	94.62
	Chase Banking	33	33	6	81.82	N/A
	Contacts	98	101	19	83.67	80.16
	Facebook	116	128	41	76.72	85.52
	Instagram	58	58	9	84.48	87.75
	Messaging	23	23	3	86.96	80.78
	WhatsApp	37	37	8	76.30	80.85
HTC One	Calendar	38	40	8	84.21	80.33
	Chase Banking	33	34	5	87.88	N/A
	Contacts	73	73	12	83.56	84.16
	Facebook	79	78	23	70.89	90.52
	Instagram	58	58	5	91.38	93.25
	Messaging	85	83	34	60.00	86.02
	WhatsApp	25	25	1	96.00	92.70

Table 2: Reconstruction of GUI trees of various apps from different phones.

Table 1 shows that a large percentage of these data structures persist after the application is backgrounded. As presented in Section 2, this percentage will drop over time if the app remains in the background — Section 4.3 expands on this by evaluating GUITAR's GUI recovery capability over a period of 24 hours. For all 21 cases, an average 89.23% of the data structures persist. We only see 4 cases where less than 80% persist. Further, recall that GUITAR can reconstruct an app's remaining GUI even if some of the data structures are missing — the missing pieces might simply be blank spaces on the screen.

Lastly, these results show that GUITAR's memory image scanner is robust enough to recover 100% of the data structure instances in the backgrounded apps' memory images. Although we point out that individual data structure recovery is *not* GUITAR's primary capability and may be performed by other existing memory forensics techniques.

4.2 Reconstructed GUIs (Finished Puzzles)

Using the recovered GUI data "pieces," we now evaluate how accurately GUITAR reconstructs each GUI tree and the quality of each redrawn GUI.

We first need to compare a GUITAR-reconstructed GUI tree with a *Ground Truth Tree* (i.e., the app's true GUI tree as it was in the memory image). To obtain the Ground Truth Tree, we instrumented each app to log the structure and content of its GUI tree in the foreground. From that log, we subtracted the elements of the tree that were lost when the app was backgrounded (recall that the tree's structure is explicitly destroyed when the app is backgrounded). This yielded the ideal tree which GUITAR could reconstruct with the remaining data structures.

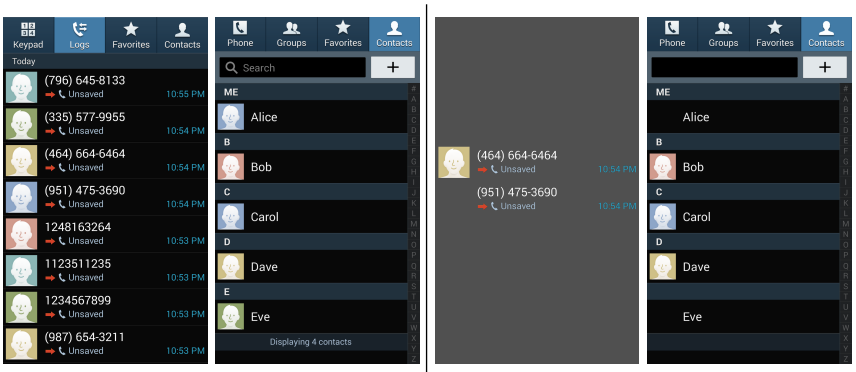
However, the most important (and interesting) test for GUITAR is: How does the reconstructed GUI look? To reliably compare each GUITAR-reconstructed GUI to the app's original GUI, we used Content Based Image Recognition (CBIR) to score the similarity between the GUI reconstructed by GUITAR and a screenshot of the original app (from Android's `screencap` program). Note that CBIR is used instead of a naive per-pixel comparison because GUITAR may rearrange a few GUI elements — causing many pixels to change though the overall image's content remains the same. For this, we employed the widely used LIRE open source CBIR library [32,33] and the default CEDD indexing feature [23]. Notice that this comparison is actually *unfavorable to GUITAR* because the screenshot is taken when the app is in the foreground, and GUITAR has no control over what data is overwritten when the app is backgrounded.

Table 2 presents the devices and app names in Columns 1 and 2. Columns 3 and 4 show the size (number of nodes) of the Ground Truth Tree and GUITAR-reconstructed tree, respectively[6]. Note that the number of edges is always the number of nodes minus 1. Column 5 presents the edit distance (number of node additions and deletions) between the Ground Truth Tree and reconstructed tree. The percentage of the GUITAR-rebuilt tree that is *strictly identical* (content, structure, and position in the tree) to the Ground Truth Tree is shown in Column 6. Lastly, Column 7 lists the CBIR score between the GUITAR-reconstructed GUI and a screenshot of the foreground app.

Table 2 shows that GUITAR-reconstructed GUI trees are very similar to the apps' original GUI trees. Column 6 shows that most of the reconstructed trees (14 out of 21) are more than 80% strictly identical to the Ground Truth Trees (with an average of 82.63%). Moreover, the edit distances in Column 5 show that many rebuilt trees only differ from the (ideal) Ground Truth Trees by less than 10 modifications (node additions or deletions). Further, as described in Section 3, often the reconstructed tree branches that are not identical to the Ground Truth Tree are simple permutations of the tree's structure. In the following section, we will highlight LG G3's WhatsApp test to demonstrate how reconstructed GUIs are often a slightly "rearranged" form of the original GUIs.

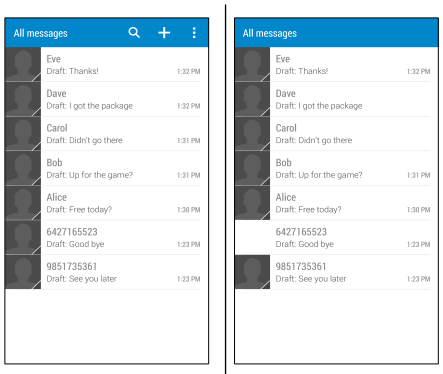
Table 2 also shows that 7 reconstructed trees are slightly larger than their Ground Truth Trees. A larger tree is always caused by *historic GUI data*. For instance, the Samsung S4 Contacts GUI is reconstructed with several elements

[6]The number of nodes is smaller than the number of data structure instances because each TreeNode includes the node plus all elements in its DrawOpList and graphical contents.



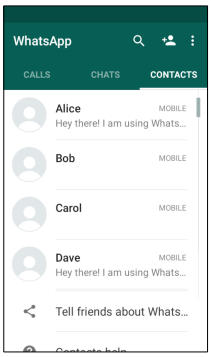
(a) Earlier Screen. (b) Latest Screen. (c) Conflict Branch. (d) Recovered GUI.

Figure 6: Samsung Contacts app with redrawn full conflict branch.

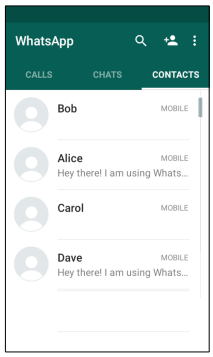


(a) App Screen. (b) Recovered GUI.

Figure 7: HTC Messaging.

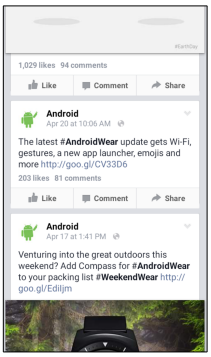


(a) App Screen.

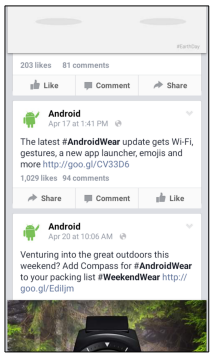


(b) Recovered GUI.

Figure 8: LG WhatsApp Contacts.

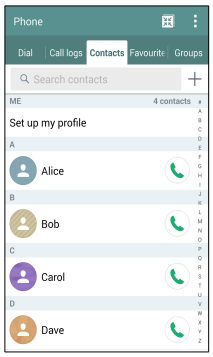


(a) App Screen.

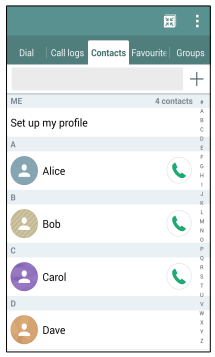


(b) Recovered GUI.

Figure 9: Samsung Facebook.



(a) App Screen.



(b) Recovered GUI.

Figure 10: LG Contacts app.

that were present on an even earlier screen. Figure 6 shows the two previous GUI screens which contributed to the resulting GUITAR-reconstructed GUIs: one from the most recently viewed screen (Figure 6(b)), and a portion of the earlier screen (a full conflicting branch, Figure 6(c)). Of the 21 test cases, only 4 have reconstructed trees smaller than the Ground Truth Trees. Smaller trees are caused when too few data structures with visual GUI contents are recovered. As detailed in Section 3.2, GUITAR removes empty branches of the reconstructed tree, yielding a tree smaller than the Ground Truth Tree (which does not remove empty branches). Empty branches result in blank areas in the redrawn GUI. As Figure 7 shows, the HTC One's Messaging app GUI loses three of the icons on the top of the screen and one thumbnail image when the app is backgrounded.

Most importantly, the CBIR results summarize how *visually similar* the re-drawn GUIs are to the original app's screens. Column 7 of Table 2 shows that all test cases score between 80.16% and 95.47%, with an overall average of 87.16% similarity. To illustrate this measurement Figure 9 shows the *best case*: Samsung S4's Facebook, and Figure 10 shows the *worst case*: LG G3's Contacts. Even in the worst case (80.16%) the GUITAR-reconstructed GUI is quite similar to the original GUI.

From the CBIR similarity scores, we make a few observations: First, certain apps have consistent GUI reconstruction results. For instance, Instagram has good scores for all devices (89.14% for Samsung, 87.75% for LG, and 93.25% for HTC). Again, the vendor-specific apps do not show any similarity across devices. For example, Samsung's Contacts is among the best cases at 95.47%, but the LG G3 Contacts GUI is 80.16%. We also find that no device outperforms the others by a significant margin. The device-specific averages are all very similar: 87.50% for Samsung, 84.25% for LG, and 86.77% for HTC.

Also note that the GUI tree reconstruction metrics are somewhat misrepresentative, which prompted us to perform the CBIR similarity comparison. Several apps have reconstructed trees that seem fairly different from their Ground Truth Trees, but the displayed GUIs are very similar to the original apps' GUIs. One such example is the Samsung S4's Facebook app. In this case, the reconstructed tree is 77.65% identical to the Ground Truth Tree with an edit distance of 48 (i.e., it would take 48 additions or deletions to make the trees fully identical). However, the GUITAR-redrawn GUI scores 95.47% similarity to the app's screenshot (as highlighted in Figure 9). This is due to many small GUI elements being "best fit" matches for the same GUI tree nodes. Therefore, GUITAR reconstructed a GUI tree which has many nodes mapped to alternate locations, but in fact the visual elements are nearly interchangeable.

Lastly, we found that several test cases had similar data structure destruction patterns caused by backgrounding the app. Manual investigation revealed that textual glyphs and UI colors are often the first data structures to be deallocated and overwritten. This turns out to be favorable for investigators because glyphs and colors can be reconstructed by analyzing the app's APK from a forensic image of the smartphone's SD card (acquired alongside a memory image). For these cases, we used a python script to extract the glyph icons and colors from the APKs and patch them into the overwritten data structures.

GUI Reconstruction Time We measured the GUI reconstruction time for each case in Table 2. GUITAR's running time ranges from 5 minutes to 10 minutes, *including* the scanning of the memory image for GUI element data structure recovery. If such recovery time is excluded (because it is not the main capability of GUITAR), the GUI reconstruction time alone ranges from 3 to 5 minutes, which is very acceptable for (off-line) digital forensics investigations.

4.2.1 Case Study: WhatsApp on LG G3

In several test cases presented in Table 2 we found that GUITAR-reconstructed GUIs are slightly "rearranged" forms of the original apps' GUIs. In this case study, we examine one such case in detail where this effect is most obvious. When we performed the LG G3's WhatsApp experiment, the last GUI we viewed was the app's Friends List window. Thus, this is the GUI we aimed to redraw using GUITAR.

As shown in Row 14 of Table 1, 78.97% of WhatsApp's GUI data structures persisted in the backgrounded app's memory. From those recovered structures, GUITAR was able to reconstruct the app's GUI tree of 37 nodes. Table 2 shows that the reconstructed tree has the same size as the Ground Truth Tree but is only 76.30% identical. Curiously, when we looked at the GUITAR-redrawn GUI everything appeared to be drawn correctly.

Through further investigation we found that 4 of the major GUI elements were virtually interchangeable: each subtree having the same geometric on-screen dimensions and identical tree structure. Correlating these 4 GUI elements to the redrawn GUI revealed that these were the 4 rows for each friend in our Friends List. While rebuilding the GUI tree, GUITAR could not determine the order of these subtrees (given their similarity) and thus broke the tie randomly — swapping 2 of the friends in the list.

Figure 8 presents the app's foreground screenshot and the GUITAR-reconstructed GUI. Notice how the first and second friend in the list have swapped positions in the rebuilt GUI. The only other difference between the GUIs is the missing icons at the bottom of the screen which were lost when the app was backgrounded. To correct this, GUITAR could leverage heuristics (e.g., the structures' offset in the heap) to help break such "best match" ties more accurately. We leave this as future work.

4.2.2 Bypassing the Password Check

In this section, we highlight another interesting feature of GUITAR: It helps bypass an app's password protection. Many Android apps (particularly those handling highly sensitive data) require users to log in when they bring the app back to the foreground after a certain (short) period of time. One such example is the Chase Banking app. Like many other highly secure apps, the Chase app requires users to log into their Chase account every time the app is brought to the foreground. This login is cached and a timer is used to automatically log the user out after some time of inactivity. Thus, if someone later opened the app it would again ask for login credentials.

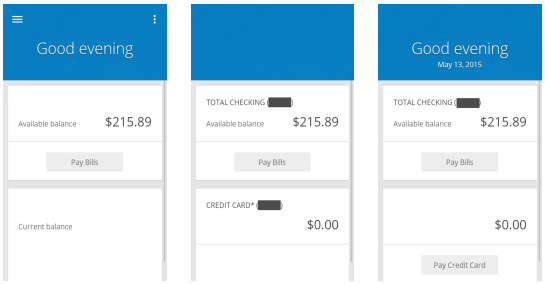
Importantly however, when such a secure app is being used, the last screen the user views before backgrounding the app is always some *internal* screen of the app after the user has already logged in. Further, this most recent internal screen will be the one present in the app's heap *even after the app has logged the user out.* For these apps, GUITAR can recover confidential personal information frozen in the memory image long after the app's session has expired.

We have evaluated GUITAR with memory images from the Chase Banking app on all three test smartphones. In each case, we logged into our personal Chase Bank account, checked our account balances, and backgrounded the app. We then waited for the app's session timer to expire (thus requiring us to log in if we brought up the app again) and then took the memory image of the backgrounded app.

Note that Table 2 shows "N/A" for the Chase Banking app's CBIR scores. This is because the Chase Banking app, like many other secure apps, has *explicitly disabled* screenshots from being taken when the app is in the foreground. This however cannot prevent GUITAR from reconstructing the app's GUI from the memory image.

Table 2 shows that GUITAR was able to rebuild the Chase Banking app's GUI tree with very high accuracy: 87.88% identical to the Ground Truth Tree for the Samsung S4 and HTC One devices and 81.82% identical for the LG G3. For visual comparison, Figure 11 shows the reconstructed app GUIs for all 3 devices (and one of the authors' graduate-student-size account balance).

We point out that a broader impact of this case study is the user privacy concerns it raises for running highly sensitive apps on smartphones. Interestingly, even apps focusing on privacy (such as TextSecure [36]) cannot disrupt GUITAR's recovery. This is because GUITAR operates on the lowest-level GUI objects (defined by Android, not by the apps). Such GUI data is used directly by the system for GUI display. Thus any app which displays a GUI will have to use these objects, leaving behind the GUI-related data that GUITAR will (later) use for GUI recovery. In our targeted application scenario (digital investigation), we assume that the privacy issues are addressed by legal protocols and policies (e.g., requirement of a search warrant).



(a) LG G3 (b) Samsung S4 (c) HTC One

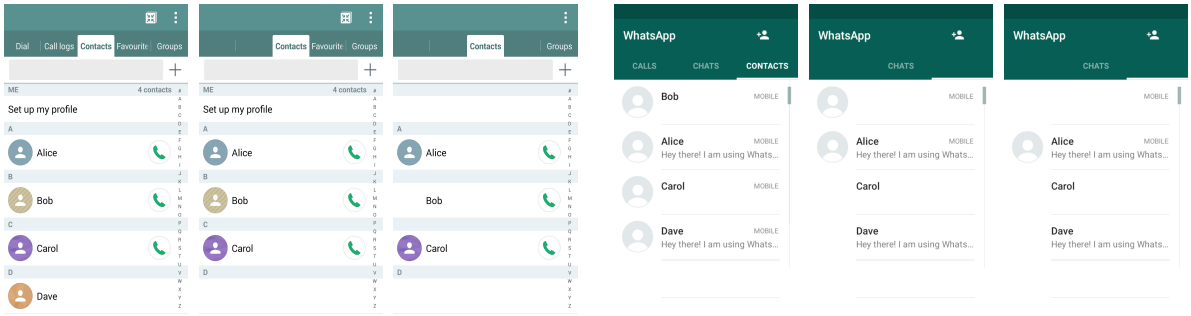
Figure 11: Reconstructed Chase Banking GUIs. Although the user was logged out, recovered GUIs still reveal sensitive information. Note: we manually blocked out the account number.

4.3 GUI Reconstruction Over Time

In this section, we evaluate GUITAR's GUI reconstruction capability for memory images captured over a longer period of time since the app was backgrounded. As described in Section 2, Android rebuilds an app's GUI from scratch (i.e., allocates and builds a new GUI tree) every time it is brought to the foreground; as such, data of its previous GUI are freed and risk being overwritten if the app performs background processing. However, as shown in Figure 2, a non-trivial amount of the GUI data persist in the app's free heap space over a period of 24 hours.

App	Fore/Backgound	Backgrounded Time	Instances	% Persists	Tree Size	Edit Distance	% Original
Contacts	Foreground		731		98		
	Background	1 hour	640	87.55	101	19	83.67
		2 hours	640	87.55	101	19	83.67
		5 hours	635	86.87	97	24	79.06
		10 hours	364	49.79	65	43	54.23
		24 hours	320	43.78	63	47	51.56
Instagram	Foreground		301		58		
	Background	1 hour	232	77.08	58	11	82.03
		2 hours	232	77.08	58	11	82.03
		5 hours	232	77.08	58	11	82.03
		10 hours	232	77.08	58	11	82.03
		24 hours	205	68.11	51	18	74.51
Messaging	Foreground		101		23		
	Background	1 hour	90	89.11	23	3	86.96
		2 hours	84	83.17	21	6	82.52
		5 hours	84	83.17	21	6	82.52
		10 hours	56	55.45	18	11	57.01
		24 hours	56	55.45	18	11	57.01
WhatsApp	Foreground		214		37		
	Background	1 hour	157	73.36	37	10	71.94
		2 hours	116	54.21	35	13	65.02
		5 hours	115	53.74	35	13	64.13
		10 hours	115	53.74	35	13	64.13
		24 hours	109	50.93	32	18	50.71

Table 3: Reconstruction of background apps' GUI trees over a 24 hour period.



(a) 1 hour (b) 5 hours (c) 24 hours (d) 1 hour (e) 5 hours (f) 24 hours

Figure 12: Contacts and WhatsApp GUIs with varying degree of loss over time.

Using the memory images of 4 apps taken in Section 4.1 as a baseline (i.e., time $t0$), we *left the apps in the background*, untouched, and took additional memory images at times $t0+1$ (hour), $t0+2$, $t0+5$, $t0+10$, and $t0+24$. During this time period, the other apps on the smartphone were heavily used. We employed the same ground truth collection as in the previous sections, and then applied GUITAR on these memory images.

Table 3 presents our results for the LG G3 phone. For comparison, we include each app's foreground data (with 100% of the intact GUI tree in memory). Note that each app's GUI reconstruction results for the memory image captured at $t0$ are already presented in Tables 1 and 2. Column 1 of Table 3 shows each app's name. Column 2 shows if the app was in the foreground or background, and Column 3 lists the time each app had been in the background when the memory image was taken. The number of data structures in the memory images is listed in Column 4 (like before, GUITAR located all *recoverable* data structures). Column 5 presents the percentage of the foreground data structures which persist in the memory. Columns 5, 6, and 7 present the reconstructed GUI tree's size, edit distance, and percentage that is identical to the Ground Truth Tree, respectively.

Table 3 presents several interesting results: First, as expected, after 1 hour in the background the GUI recovery results are similar to those reported in the previous sections (i.e., 15 minutes in the background). On average, 81.78% of the data structures are recoverable — fairly close to the average in Section 4.1: 89.23%. Further, GUITAR reconstructs GUI trees that are all more than 71% identical to their Ground Truth Trees.

Notably, Table 3 shows that loss of GUI data is *non-linear* over time. For example, the Instagram GUI data had no loss until 9% of the data structures were overwritten in the 24 hour-memory image. Intuitively, this is because those data structures remain intact, until one or more bursts of background computation have requested enough memory to overwrite the GUI data. Because of this, the apps tend to exhibit "stepwise" GUI data loss. The Messaging app in Table 3 shows this trend: 6% of the data were lost after 2 hours, and then no data were lost until 28% more data were lost after 10 hours.

To visually compare the reconstructed GUIs, Figure 12 shows the gradual (but graceful) degradation of GUITAR-reconstructed GUIs over the 24-hour period. Again, the GUIs reconstructed from the 1-hour-memory images are very similar to those reconstructed in the previous subsections. After 24 hours, the GUIs will be missing some non-trivial content. But GUITAR is robust enough to reconstruct the partial GUIs showing the graphical content of the remaining GUI data, which (as shown in Figure 12) are still of forensic value.

5. RELATED WORK

By and large, memory forensics tools have focused on recovering individual data structure instances as evidence. They can be roughly divided into two categories based on the data structure signatures they employ. Value-invariant signatures leverage known in-memory value patterns or invariants to locate data structure instances via brute-force scanning [17, 18, 24, 37, 41, 45]. Structural-invariant (or "points-to") signatures rely on the interconnection of data structure networks [30]. To date, most forensic tools and reverse engineering systems rely on traversing data structures (making use of structural-invariant assumptions) [19, 22, 35, 48]. Many techniques, such as HOWARD [42], REWARDS [31], and TIE [28], leveraged program analysis to automatically infer data structure definitions for deriving their signatures. DIMSUM [29] used hybrid-signatures along with probabilistic inference to identify data structure instances without memory mapping information (e.g., page tables). GUITAR, however, addresses a different problem: reconstructing GUIs by piecing together data structures already recovered. As such, the input of GUITAR is the output of existing data structure recovery tools.

Recently, DSCRETE [40] was proposed to automatically render application output for recovered data structures. It still focuses on individual structure instances and involves an application-specific dynamic analysis step. DSCRETE requires the data structure it renders to be "live" and intact. By comparison, GUITAR is app-independent and reconstructs entire GUIs, each containing hundreds of inter-dependent *deallocated* data structures, with critical pointers between them nullified by Android.

Forensics research has only recently started to focus on mobile devices. DEC0DE [47] employed probabilistic finite state machines to recover plain-text call logs and address book entries from phone storage. Spurred by the release of Android memory acquisition tools [13, 43], several efforts began recovering app-specific data from memory images. VCR [39] recovers photographic evidence from the camera service's memory. Thing et al. [46] aimed at recovering low-level inter-app communications from memory images. Other works [12, 34] have investigated Dalvik-JVM control structures and raw Java object content. Apostolopoulos et al. [15] later found several login credentials in app memory images. Lastly, Hilgers et al. [26] proposed using memory analysis on cold-booted Android phones. GUITAR shares the same analysis subjects with these efforts: Android memory images. However, GUITAR is unique in focusing on reconstructing GUIs (instead of specific types of program data) and redrawing the GUIs in the same, full-screen view as seen by smartphone users.

6. CONCLUSION

To address a real-world smartphone forensics challenge, we have presented GUITAR, a memory forensics technique which automatically reconstructs and redraws Android app GUIs frozen in a memory image. Instead of focusing on recovering individual data structures (as most existing techniques do), GUITAR takes recovered GUI data structures — already deallocated by Android — as input and pieces them back together to recreate the original GUI. Our evaluation results show that GUITAR achieves high accuracy in GUI tree reconstruction and redrawing, and tolerates loss of GUI data elements over time by reconstructing partial yet meaningful GUIs.

7. ACKNOWLEDGMENTS

We thank the anonymous reviewers for their insightful comments and suggestions. We also thank Dr. Golden G. Richard III for his valuable input on the legal and technical aspects of memory forensics. This work was supported in part by NSF under Award 1409668.

8. REFERENCES

[1] 7 American Law Reports. 4th, 8, 2b.
[2] Pearl Brewing Co. v. Jos. Schlitz Brewing Co. 415 F. Supp. 1122, (1976).
[3] US v. Scholle. 553 F. 2d 1109, (1977).
[4] Nat. Union Elec. Corp. v. Matsushita Elec. Indus. Co. 494 F. Supp. 1257, (1980).
[5] US v. Vela. 673 F. 2d 86, (1982).
[6] US v. Bonallo. 858 F. 2d 1427, (1988).
[7] Gates Rubber Co. v. Bando Chemical Industries, Ltd. 9 F. 3d 823, (1993).
[8] Illinois Tool Works v. Metro Mark Products, Ltd. 43 F. Supp. 2d 951, (1999).
[9] John Paul Mitchell Systems v. Quality King Distributors, Inc. 106 F. Supp. 2d 462, (2000).
[10] Schaghticoke Tribal Nation v. Kempthorne. 587 F. Supp. 2d 389, (2008).
[11] Hungarian method source. `https://github.com/maandree/hungarian-algorithm-n3/blob/master/hungarian.c`, 2014.
[12] 504ENSICS Labs. Dalvik Inspector (DI) Alpha. `http://www.504ensics.com/tools/dalvik-inspector-di-alpha`, 2013.
[13] 504ENSICS Labs. LiME Linux Memory Extractor. `https://github.com/504ensicsLabs/LiME`, 2013.
[14] F. Adelstein. Live forensics: diagnosing your system without killing it first. *Communications of the ACM*, 49(2), 2006.
[15] D. Apostolopoulos, G. Marinakis, C. Ntantogian, and C. Xenakis. Discovering authentication credentials in volatile memory of android mobile devices. In *Collaborative, Trusted and Privacy-Aware e/m-Services*. 2013.
[16] J. Ashcroft, D. J. Daniels, and S. V. Hart. 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.
[17] C. Betz. Memparser forensics tool. `http://www.dfrws.org/2005/challenge/memparser.shtml`, 2005.
[18] C. Bugcheck. Grepexec: Grepping executive objects from pool memory. In *Proc. Digital Forensic Research Workshop*, 2006.
[19] M. Carbone, W. Cui, L. Lu, W. Lee, M. Peinado, and X. Jiang. Mapping kernel objects to enable systematic integrity checking. In *Proc. CCS*, 2009.
[20] B. D. Carrier. Risks of live digital forensic analysis. *Communications of the ACM*, 49(2), 2006.
[21] B. D. Carrier and J. Grand. A hardware-based memory acquisition procedure for digital investigations. *Digital Investigation*, 1, 2004.
[22] A. Case, A. Cristina, L. Marziale, G. G. Richard, and V. Roussev. FACE: Automated digital evidence

discovery and correlation. *Digital Investigation*, 5, 2008.

[23] S. A. Chatzichristofis and Y. S. Boutalis. CEDD: color and edge directivity descriptor: a compact descriptor for image indexing and retrieval. In *Computer Vision Systems*. 2008.

[24] B. Dolan-Gavitt, A. Srivastava, P. Traynor, and J. Giffin. Robust signatures for kernel data structures. In *Proc. CCS*, 2009.

[25] M. Graves. *Digital Archaeology: The Art and Science of Digital Forensics*. Addison-Wesley, 2013.

[26] C. Hilgers, H. Macht, T. Muller, and M. Spreitzenbarth. Post-mortem memory analysis of cold-booted android devices. In *Proc. IT Security Incident Management & IT Forensics (IMF)*, 2014.

[27] H. M. Jarrett, M. W. Bailie, E. Hagen, and N. Judish. 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.

[28] J. Lee, T. Avgerinos, and D. Brumley. TIE: Principled reverse engineering of types in binary programs. In *Proc. NDSS*, 2011.

[29] Z. Lin, J. Rhee, C. Wu, X. Zhang, and D. Xu. DIMSUM: Discovering semantic data of interest from un-mappable memory with confidence. In *Proc. NDSS*, 2012.

[30] Z. Lin, J. Rhee, X. Zhang, D. Xu, and X. Jiang. SigGraph: Brute force scanning of kernel data structure instances using graph-based signatures. In *Proc. NDSS*, 2011.

[31] Z. Lin, X. Zhang, and D. Xu. Automatic reverse engineering of data structures from binary execution. In *Proc. NDSS*, 2010.

[32] M. Lux. Content based image retrieval with lire. In *Proc. ACM International Conference on Multimedia*, 2011.

[33] M. Lux and S. A. Chatzichristofis. Lire: lucene image retrieval: an extensible java cbir library. In *Proc. ACM International Conference on Multimedia*, 2008.

[34] H. Macht. Live memory forensics on android with volatility. *Friedrich-Alexander University Erlangen-Nuremberg*, 2013.

[35] P. Movall, W. Nelson, and S. Wetzstein. Linux physical memory analysis. In *Proc. USENIX Annual Technical Conference, FREENIX Track*, 2005.

[36] Open Whisper Systems. TextSecure Private Messenger. `https://play.google.com/store/apps/details?id=org.thoughtcrime.securesms`, 2015.

[37] N. L. Petroni Jr, A. Walters, T. Fraser, and W. A. Arbaugh. FATKit: A framework for the extraction and analysis of digital forensic data from volatile system memory. *Digital Investigation*, 3, 2006.

[38] B. Saltaformaggio. Forensic carving of wireless network information from the android linux kernel. *University of New Orleans*, 2012.

[39] B. Saltaformaggio, R. Bhatia, Z. Gu, X. Zhang, and D. Xu. VCR: App-agnostic recovery of photographic evidence from android device memory images. In *Proc. CCS*, 2015.

[40] B. Saltaformaggio, Z. Gu, X. Zhang, and D. Xu. DSCRETE: Automatic rendering of forensic information from memory images via application logic reuse. In *Proc. USENIX Security*, 2014.

[41] A. Schuster. Searching for processes and threads in microsoft windows memory dumps. *Digital Investigation*, 3, 2006.

[42] A. Slowinska, T. Stancescu, and H. Bos. Howard: A dynamic excavator for reverse engineering data structures. In *Proc. NDSS*, 2011.

[43] H. Sun, K. Sun, Y. Wang, J. Jing, and S. Jajodia. Trustdump: Reliable memory acquisition on smartphones. In *Proc. European Symposium on Research in Computer Security*. 2014.

[44] J. Sylve, A. Case, L. Marziale, and G. G. Richard. Acquisition and analysis of volatile memory from android devices. *Digital Investigation*, 8, 2012.

[45] The Volatility Framework. `https://www.volatilesystems.com/default/volatility`.

[46] V. L. Thing, K.-Y. Ng, and E.-C. Chang. Live memory forensics of mobile phones. *Digital Investigation*, 7, 2010.

[47] R. Walls, B. N. Levine, and E. G. Learned-Miller. Forensic triage for mobile phones with DEC0DE. In *Proc. USENIX Security*, 2011.

[48] J. Zeng, Y. Fu, K. A. Miller, Z. Lin, X. Zhang, and D. Xu. Obfuscation resilient binary code reuse through trace-oriented programming. In *Proc. CCS*, 2013.

APPENDIX

A. DATA STRUCTURE SIGNATURES

Here we present additional details on the data structure signatures used during GUITAR's memory image scanning. Though not GUITAR's main contribution, data structure recovery is a prerequisite for GUITAR's GUI reconstruction.

```
  class DrawTextOp {
[0x0 ]    void * vtable;
[0x4 ]    SkPaint* mPaint;
[0x8 ]    bool mQuickRejected;
[0xC ]    Rect mLocalBounds;
[0x1C]    const char* mText;
[0x20]    int mBytesCount;
[0x24]    int mCount;
[0x28]    float mX;
[0x2C]    float mY;
[0x30]    const float* mPositions;
[0x34]    float mTotalAdvance;
[0x38]    mat4 mPrecacheTransform;
  };
```

```
DrawTextOp(A) =
  vtable_ptr_value(A) &&
  data_ptr_value(A + 0x4) &&
  SkPaint(*(A + 0x4)) &&
  bool_value(A + 0x8) &&
  Rect(&A + 0xC) &&
  data_ptr_value(A + 0x1C) &&
  printable_text(*(A + 0x1C)) &&
  int_value(A + 0x20) &&
  int_value(A + 0x24) &&
  float_value(A + 0x28) &&
  float_value(A + 0x2C) &&
  data_ptr_value(A + 0x30) &&
  float_value(*(A + 0x30)) &&
  float_value(A + 0x34) &&
  mat4(&A+ 0x38);
```

Figure 13: DrawTextOp class definition and resulting data structure signature.

GUITAR uses a combination of structural and value invariant signatures for each structure it recovers. Figure 13 shows a representative example: The source code definition and signature for the DrawTextOp data structure. Note that each field which GUITAR relies on for GUI reconstruction is converted into boolean conditions. During memory image scanning, the value-invariant boolean conditions identify potential signature matches and the structural-invariant functions validate the interconnection between different objects. For instance, the second field of the DrawTextOp structure is first checked with a value-invariant (`data_ptr_value`) and then the interconnection is checked by validating the pointer target (`SkPaint`).