12-2016

Convicted by memory: Automatically recovering spatial-temporal evidence from memory images

Brendan D. Saltaformaggio
Purdue University

Graduate School Form
30 Updated 12/26/2015

PURDUE UNIVERSITY
GRADUATE SCHOOL
Thesis/Dissertation Acceptance

This is to certify that the thesis/dissertation prepared

By Saltaformaggio, Brendan Dominic

Entitled

CONVICTED BY MEMORY: AUTOMATICALLY RECOVERING
SPATIAL-TEMPORAL EVIDENCE FROM MEMORY IMAGES

For the degree of Doctor of Philosophy

Is approved by the final examining committee:

Dr. Dongyan Xu Chair	Dr. Mikhail J. Atallah
Dr. Xiangyu Zhang Co-chair	
Dr. Elisa Bertino	
Dr. Aniket Kate	

To the best of my knowledge and as understood by the student in the Thesis/Dissertation Agreement, Publication Delay, and Certification Disclaimer (Graduate School Form 32), this thesis/dissertation adheres to the provisions of Purdue University's "Policy of Integrity in Research" and the use of copyright material.

Approved by Major Professor(s): Dr. Dongyan Xu and Dr. Xiangyu Zhang

Approved by: Dr. Sunil Prabhakar / Dr. William J. Gorman 12/7/2016

Head of the Departmental Graduate Program Date

CONVICTED BY MEMORY: AUTOMATICALLY RECOVERING SPATIAL-TEMPORAL EVIDENCE FROM MEMORY IMAGES

A Dissertation

Submitted to the Faculty

of

Purdue University

by

Brendan D. Saltaformaggio

In Partial Fulfillment of the

Requirements for the Degree

of

Doctor of Philosophy

December 2016

Purdue University

West Lafayette, Indiana

To my mother, Jody Juneau.

ACKNOWLEDGMENTS

From my first day at Purdue, my advisor, Dr. Dongyan Xu, has stood as a pillar of mentorship, teamwork, and limitless opportunity. From his example, Professor Xu has shown me the type of professor, advisor, and mentor that I want to be. I cannot imagine that many graduate students enjoy the degree of freedom in developing their own research agenda that Professor Xu has granted me, and I will forever reserve my deepest gratitude for his guidance, understanding, and support toward my success.

I also wish to express my sincerest appreciation and thanks to my co-advisor, Dr. Xiangyu Zhang. This dissertation is the culmination of many ideas, debates, and laughs shared with Professor Zhang. He truly is a colleague like no other.

I dedicate this dissertation to my mother, Jody Juneau, as it could not exist without her. The value of her endless emotional support, limitless words of encouragement, and staggering monetary commitment to my education cannot be conveyed in words alone. It is to her that I owe every implemented idea, published paper, and conquered challenge. No person alive more deserves her self-given title, the "president of my fan club."

This dissertation is built upon the bedrock of support, happiness, and patience I share with the love of my life, Kristen Johnson. From eating packaged dinners at midnight in the lab to weekend escapes in Chicago, together we got through our Ph.D.s; nothing can stop us now.

Lastly, an integral part of my time in graduate school was the collaboration and camaraderie among my lab mates in the, aptly named, FRIENDS lab. Foremost among them, I owe most of this dissertation to Rohit Bhatia's tireless effort, invaluable knowledge, and witty humor. I count myself very lucky to be among great colleagues, past and present, such as Chung Hwan Kim, Hui Lu, Yonghwi Kwon, Zhui Deng, Cong Xu, Taegyu Kim, Shiqing Ma, Kexin Pei, Praseem Banzal, and Zhongshu Gu.

TABLE OF CONTENTS

Page

LIST OF TABLES vi

LIST OF FIGURES vii

ABSTRACT ix

1 INTRODUCTION 1
1.1 Dissertation Statement 1
1.2 Thesis and Contributions 1
1.2.1 DSCRETE 3
1.2.2 VCR 4
1.2.3 GUITAR 4
1.2.4 RetroScope 5
1.3 Dissertation Organization 6

2 DSCRETE: CONTENT REVERSE ENGINEERING 8
2.1 Overview 10
2.1.1 Key Idea: Executable Code Reuse 10
2.1.2 Overview of DSCRETE Workflow 11
2.1.3 Assumptions and Setup 13
2.2 Design 13
2.2.1 Dynamic Data Dependence Tracing 13
2.2.2 Identifying Functional Closure 16
2.2.3 Finding the Scanner's Entry Point 19
2.2.4 Memory Image Scanning 21
2.2.5 Cross-State Execution 22
2.3 Evaluation 25
2.3.1 Experimental Setup 25
2.3.2 Function Identification Effectiveness 26
2.3.3 Memory Scanner Effectiveness 29
2.4 Future Expansion of DSCRETE 35

3 VCR: VISUAL CONTENT RECOVERY 39
3.1 Motivation 41
3.1.1 Centralized Photographic Evidence 44
3.1.2 Assumptions and Setup 45
3.2 Design 45
3.2.1 Recovering Evidentiary Data 46

Page

3.2.2 Vendor-Generic Signature Derivation . . . 48
3.2.3 Memory Image Scanning . . . 51
3.2.4 Rendering Evidence . . . 53
3.3 Evaluation . . . 55
3.3.1 App-Agnostic Evidence Recovery . . . 56
3.3.2 Analysis Across Android Frameworks . . . 62
3.3.3 Recovering Temporal Evidence . . . 65
3.3.4 Privacy Concerns . . . 68

4 GUITAR: GUI TREE ARCHAEOLOGY . . . 70
4.1 The Android GUI Framework . . . 73
4.1.1 Challenges and Solution Overview . . . 76
4.2 Design . . . 76
4.2.1 Reconstructing GUI Tree Topology . . . 77
4.2.2 Remapping Drawing Operations . . . 80
4.2.3 Runtime Recreation for GUI Redraw . . . 85
4.2.4 Data Structure Signatures . . . 88
4.3 Evaluation . . . 89
4.3.1 GUI Data Elements (Puzzle Pieces) . . . 90
4.3.2 Reconstructed GUIs (Finished Puzzles) . . . 92
4.3.3 GUI Reconstruction Over Time . . . 99

5 RETROSCOPE: SCREEN AFTER PREVIOUS SCREEN . . . 104
5.1 Problem and Opportunity . . . 106
5.2 Design of RetroScope . . . 109
5.2.1 Selective Reanimation . . . 110
5.2.2 Interleaved Re-Execution Engine . . . 113
5.2.3 Escaping Execution and Data Accesses . . . 118
5.3 Evaluation . . . 119
5.3.1 Spatial-Temporal Evidence Recovery . . . 124
5.3.2 Case Study I: Behind the Logout . . . 126
5.3.3 Case Study II: Background Updates . . . 127
5.3.4 Case Study III: Deleted Messages . . . 128
5.4 RetroScope and Privacy Implications . . . 129

6 RELATED WORKS . . . 136

7 CONCLUSION . . . 140

REFERENCES . . . 142

VITA . . . 148

LIST OF TABLES

Table		Page
2.1	Results from identifying applications' P functions.	37
2.2	Results from DSCRETE-generated scanner+renderer tools.	38
3.1	VCR recovery from apps on commodity Android smartphones.	57
3.2	VCR recovery from current and future Android versions.	63
3.3	Time-lapse evaluation.	66
4.1	Recovery of backgrounded GUI data structures.	89
4.2	Reconstruction of GUI trees of various apps from different phones.	102
4.3	Reconstruction of background apps' GUI trees over a 24 hour period.	103
5.1	Samsung S4 results of RetroScope evaluation.	121
5.2	LG G3 results of RetroScope evaluation.	122
5.3	HTC One results of RetroScope evaluation.	123

LIST OF FIGURES

Figure		Page
1.1	Four components of my spatial-temporal memory forensics framework .	2
2.1	Illustration of content reverse engineering challenge.	9
2.2	Overview of DSCRETE workflow.	12
2.3	Example for cross-state execution.	23
2.4	Normalized size of P vs. entire binary code.	28
2.5	Observed throughput of each scanner.	31
2.6	Candidate testing output.	33
3.1	Time-lapse effect in recovered preview frames.	41
3.2	Intermediate service architecture.	44
3.3	AOSP vs. vendor customized structure.	47
3.4	Illustration of a partial CameraClient signature.	49
3.5	Matching a candidate CameraClient instance.	52
3.6	Buffers under different decoding algorithms.	54
3.7	Timelapse frames recovered from the Skype case study.	59
3.8	Measurement of temporal evidence.	66
3.9	Recovered check image left behind in a memory image.	68
4.1	Overview of a windowing system library.	74
4.2	Recoverable GUI data structures of backgrounded apps over 24 hours.	75
4.3	Broken GUI tree structures.	78
4.4	Example of drawing-content based bipartite graph matching.	80
4.5	Illustration of forced polymorphism.	87
4.6	DrawTextOp class definition and resulting data structure signature.	88
4.7	Samsung Contacts app with redrawn full conflict branch.	93
4.8	HTC Messaging.	93

Figure Page

4.9 LG WhatsApp Contacts. 93
4.10 Samsung Facebook. 94
4.11 LG Contacts app. 94
4.12 Reconstructed Chase Banking GUIs. 99
4.13 Contacts and WhatsApp GUIs. 99
5.1 Life cycles of GUI data structures. 107
5.2 Model/View implementation split of Android apps. 108
5.3 State interleaving finite automata. 113
5.4 Example of interleaved re-execution. 133
5.5 LG G3 Facebook recovery. 134
5.6 HTC One Chase Banking recovery. 134
5.7 Samsung S4 WhatsApp recovery. 135
5.8 LG G3 WeChat recovery. 135

ABSTRACT

Saltaformaggio, Brendan D. Ph.D., Purdue University, December 2016. Convicted by Memory: Automatically Recovering Spatial-Temporal Evidence from Memory Images. Major Professors: Dongyan Xu and Xiangyu Zhang.

Memory forensics can reveal "up to the minute" evidence of a device's usage, often without requiring a suspect's password to unlock the device, and it is oblivious to any persistent storage encryption schemes, e.g., whole disk encryption. Prior to my work, researchers and investigators alike considered data-structure recovery the ultimate goal of memory image forensics. This, however, was far from sufficient, as investigators were still largely unable to understand the content of the recovered evidence, and hence efficiently locating and accurately analyzing such evidence locked in memory images remained an open research challenge.

In this dissertation, I propose breaking from traditional data-recovery-oriented forensics, and instead I present a memory forensics framework which leverages program analysis to automatically recover spatial-temporal evidence from memory images by understanding the programs that generated it. This framework consists of four techniques, each of which builds upon the discoveries of the previous, that represent this new paradigm of program-analysis-driven memory forensics. First, I present DSCRETE, a technique which reuses a program's own interpretation and rendering logic to recover and present in-memory data structure contents. Following that, VCR developed vendor-generic data structure identification for the recovery of in-memory photographic evidence produced by an Android device's cameras. GUITAR then realized an app-independent technique which automatically reassembles and redraws an app's GUI from the multitude of GUI data elements found in a smartphone's memory image. Finally, different from any traditional memory forensics technique,

RetroScope introduced the vision of spatial-temporal memory forensics by retargeting an Android app's execution to recover sequences of previous GUI screens, in their original temporal order, from a memory image. This framework, and the new program analysis techniques which enable it, have introduced encryption-oblivious forensics capabilities far exceeding traditional data-structure recovery.

1 INTRODUCTION

1.1 Dissertation Statement

Since 2008 the U.S. federal government has filed an unprecedented 70 orders under the All Writs Act to compel Apple or Google to provide assistance in on-going criminal investigations [1]. This dangerous new practice reveals an unnerving truth about the current state of cyber forensics: Authorities lack the techniques necessary to investigate cyber crimes without the explicit introduction of backdoors by which to obtain evidence. My cyber forensics research directly addresses this emergent problem by developing next-generation techniques for the investigation of advanced cyber crimes.

After years of exclusively investigating persistent storage (e.g., hard disk drives), cyber forensics investigators have only recently begun turning their attention to the wealth of forensic evidence stored in a device's volatile memory (RAM). Today, memory forensics is becoming an essential capability in cyber crime investigations, as it can reveal "up to the minute" evidence of a device's activities, often *without* requiring a suspect's password to unlock the device, and it is oblivious to any persistent storage encryption schemes, e.g., whole disk encryption. Typically, an investigator only needs to obtain an image of a device's RAM (using minimally invasive hardware or software techniques) for offline analysis. However, efficiently locating and accurately analyzing such evidence locked in memory images remained an open research challenge.

1.2 Thesis and Contributions

Prior to my work, researchers and investigators alike considered data-structure recovery the ultimate goal of memory image forensics. This, however, was far from

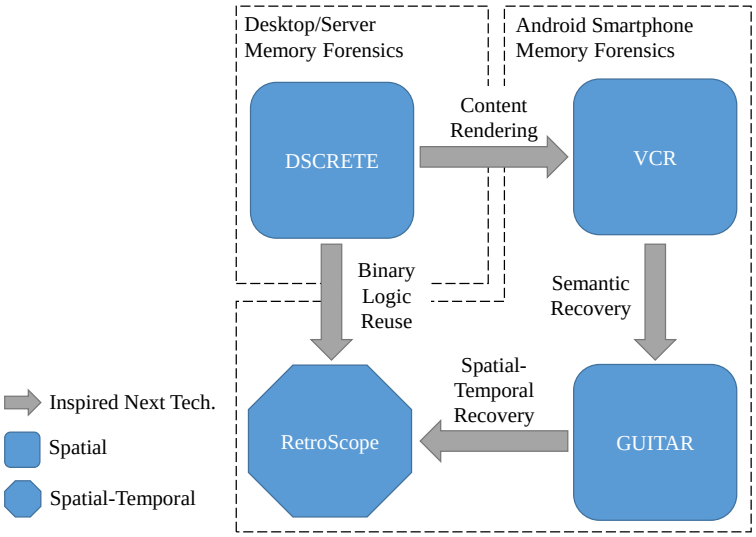


Figure 1.1.: Interconnection of the four components of my spatial-temporal memory forensics framework.

sufficient, as investigators were still largely unable to understand the *content* of the recovered data structures; a problem that I would later define as the *content reverse engineering challenge* in memory forensics. For example, consider the fragmented and encoded contents of a PDF document in memory: being unable to view, catalog, and present the PDF as evidence entirely defeats the purpose of performing memory forensics.

My research aims to break from this traditional data-recovery-oriented forensics and instead develop innovative techniques, based on retargeting program executions, for recovering spatial-temporal evidence. In this dissertation, I will present a memory forensics framework which leverages program analysis to automatically understand the artifacts that applications leave in a memory image and in doing so enable the recovery of spatial-temporal evidence from these artifacts. Specifically, this framework consists of four technologies that have driven this paradigm shift in memory image forensics capabilities. Figure 1.1 presents these four components in relation to their investigation subject (desktop/server memory forensics vs. Android smartphone memory forensics) and how the technologies revealed by each influenced the

development of their successors. Below we will briefly introduce these techniques, the technical contributions made by each, and the unique challenges that they overcome.

1.2.1 DSCRETE

State-of-the-art memory forensics involves signature-based scanning of memory images to uncover data structure instances of interest to investigators. A largely unaddressed challenge is that investigators may not be able to *interpret the content of data structure fields*, even with a deep understanding of the data structure's syntax and semantics. This is very common for data structures with application-specific encoding, such as those representing images, figures, passwords, and formatted file contents. For example, an investigator may know that a `buffer` field is holding a photo image, but still cannot display (and hence understand) the image. I call this the *data structure content reverse engineering* challenge. First, I will present DSCRETE [2], a system that enables automatic interpretation and rendering of in-memory data structure contents. DSCRETE is based on the observation that the application in which a data structure is defined usually contains interpretation and rendering logic to generate human-understandable output for that data structure. Hence DSCRETE aims to *identify and reuse* such logic in the program's *binary* and create a "scanner+renderer" tool for scanning and rendering instances of the data structure in a memory image. Different from signature-based approaches, DSCRETE avoids reverse engineering data structure signatures. Our evaluation with a wide range of real-world application binaries shows that DSCRETE is able to recover a variety of application data — e.g., images, figures, screenshots, user accounts, and formatted files and messages — with high accuracy. The raw contents of such data would otherwise be unfathomable to human investigators.

1.2.2 VCR

The ubiquity of modern smartphones means that nearly everyone has easy access to a camera at all times. In the event of a crime, the photographic evidence that these cameras leave in a smartphone's memory becomes vital pieces of digital evidence, and forensic investigators are tasked with recovering and analyzing this evidence. Unfortunately, few existing forensics tools are capable of systematically recovering and inspecting such in-memory photographic evidence produced by smartphone cameras. Next, I will present VCR [3], a memory forensics technique which aims to fill this void by enabling the recovery of all photographic evidence produced by an Android device's cameras. By leveraging key aspects of the Android framework, VCR extends existing memory forensics techniques to improve vendor-customized Android memory image analysis. Based on this, VCR targets *application-generic* artifacts in an input memory image which allow photographic evidence to be collected *no matter which application produced it*. Further, VCR builds upon the Android framework's existing image decoding logic to both automatically recover and render any located evidence. Our evaluation with commercially available smartphones shows that VCR is highly effective at recovering all forms of photographic evidence produced by a variety of applications across several different Android platforms.

1.2.3 GUITAR

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 was largely impossible with previous 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. Thirdly, I will present GUITAR [4], 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.

1.2.4 RetroScope

Finally, I will demonstrate a powerful smartphone memory forensics technique, called RetroScope [5], 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.

1.3 Dissertation Organization

This dissertation will present the evolution of this body of work. I will highlight the progression of and the shifts in memory image forensics capabilities proposed by each subsequent technology. The overall organization of this dissertation is as follows:

- Chapter 1 has introduced the forensic benefits and unique challenges of memory image investigation. I have presented an overview of my contributions to this area, with a specific focus on moving the research field away from traditional data-recovery-oriented recovery and instead developing the concepts of spatial-temporal memory forensics. For each component within this body of work, I have demonstrated the research problems they target solve and the fundamental principles behind each technique.

- Chapter 2 explains in detail the motivation, design, implementation, and evaluation of DSCRETE. I will present the landscape of memory forensics research before the development of DSCRETE, and the many investigation scenarios which benefit from DSCRETE's powerful new capabilities.

- Chapter 3 focuses on the shift from the recovery of low-level raw data to smartphone contextual evidence made available by VCR. I will explore the unique properties of Android smartphone memory forensics and explain the procedures used by VCR to recover photographs, videos, and camera previews (i.e., the on-screen camera view shown before taking a photo) *even if the criminal did not explicitly take a photo.*

- Chapter 4 presents GUITAR, my most direct effort to move memory forensics research away from individual pieces of evidence (e.g., a PDF recovered

by DSCRETE or video recovered by VCR) toward evidence which holistically reveals how a suspect *used their device in the commission of a crime.* I will discuss the significant challenges facing the rebuilding of previously displayed GUIs which remain in a smartphone's memory image and GUITAR's puzzle-piecing methodology which accomplishes this task.

- Chapter 5 details RetroScope, a technique capable of recreating entire sequences of previously displayed screens, in their original temporal order, for all apps in an Android device's memory image. RetroScope most accurately embodies the vision of spatial-temporal evidence recovery, and I will present the execution retargeting techniques which have made RetroScope possible.

- Chapter 6 describes related research efforts which serve as motivation, background, and technical complements to my work in this dissertation.

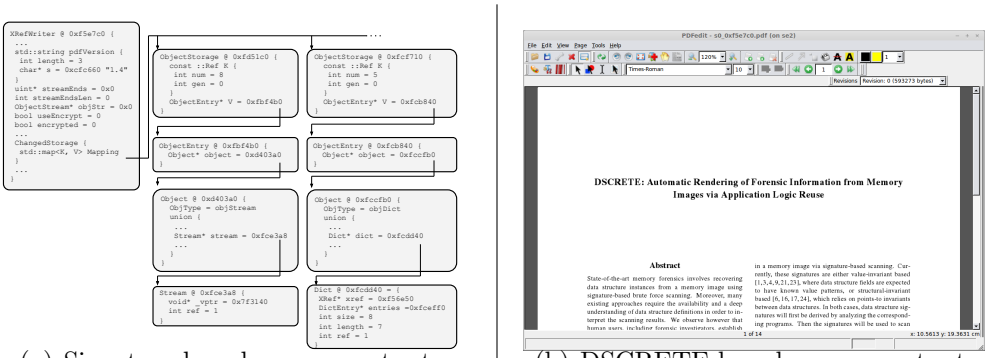
- Chapter 7 concludes this dissertation.

2 DSCRETE: CONTENT REVERSE ENGINEERING

Traditionally, digital investigations have aimed to recover evidence of a cyber-crime or perform incident response via analysis of non-volatile storage. This routine involves powering down a workstation, preserving images of any storage devices (e.g., hard disks, thumb drive, etc.), and later analyzing those images to recover evidentiary files. However, this procedure results in a significant loss of *live evidence* stored in the system's RAM — executing processes, open network connections, volatile IPC data, and OS and application data structures.

Increasingly, forensic investigators are looking to access the wealth of actionable evidence stored in a system's memory. Typically, this requires that an investigator have access to a suspected machine, prior to it being powered down, to capture an image of its volatile memory. Further, memory acquisition (both hardware [6] and software [7] based) must be as minimally invasive as possible since they operate directly on the machine under investigation. The resulting memory image is then analyzed offline using memory analysis tools. Therefore, the goal of memory analysis tools (like the work presented in this dissertation) is to recreate, in the forensics lab, the system's previously observable state based on the memory image.

Uncovering evidence from memory images is now an essential capability in modern computer forensics. Most state-of-the-art solutions locate data structure instances in a memory image via signature-based scanning. Currently these signatures are either value-invariant based [8–13], where data structure fields are expected to have known value patterns, or structural-invariant based [14–17], which rely on points-to invariants between data structures. In both cases, data structure signatures will first be derived by analyzing the corresponding programs. Then the signatures will be used to scan memory images and identify instances of the data structures. Contents of the identified instances will be presented to forensic investigators as potential evidence.



(a) Signature-based scanner output. (b) DSCRETE-based scanner output.

Figure 2.1.: Illustration of content reverse engineering challenge. (a) Raw content of an in-memory data structure instance representing a PDF file. (b) The same data structure after applying DSCRETE's scanner based on content reverse engineering.

A significant challenge, not addressed in existing memory forensics techniques, is that investigators may not be able to *interpret the content of data structure fields*, even with the data structure's syntax and semantics. This is very common for data structures with application-specific encoding, such as those representing images, passwords, messages, or formatted file contents (e.g., PDF), all of which are potential evidence in a forensic investigation. For example, an investigator may know that a `buffer` field is holding a photo image (through existing data structure reverse engineering and scanning techniques [8, 11, 15–18]), but still cannot display (and hence understand) the image. Similarly, a `message_body` field may hold an instant message, but the message is encoded, and hence it cannot be readily interpreted. We call this the *data structure content reverse engineering* challenge.

To enable automatic data structure content reverse engineering, I will present DSCRETE[1], a technique that automatically *interprets and renders* contents of data structures within a memory image. DSCRETE is based on the following observation: The application, in which a data structure is defined, usually contains interpretation and rendering logic to generate human-understandable output for that data structure.

[1]DSCRETE stands for "Data Structure Content Reverse Engineering via execuTable rEuse," pronounced as "discrete."

Hence the key idea behind DSCRETE is to *identify and reuse* such interpretation and rendering logic in a binary program — without source code — to create a "scanner+renderer" tool. This tool can then identify instances of the data structure in a memory image and, most importantly, render them in the application's *original output format* to facilitate human perception and avoid the overhead of reverse engineering data structure signatures required by signature-based memory image scanners.

To illustrate the challenge of data structure content reverse engineering, we present a concrete example (from Section 2.3). Figure 2.1(a) shows the raw content of an in-memory data structure graph representing a PDF file. This is the output produced by existing signature-based scanners. For comparison, Figure 2.1(b) shows the same data structure content *after* applying DSCRETE's scanner with content reverse engineering capability. It is quite obvious that the reverse-engineered content would be far more helpful to investigators than the raw data structure content.

We have performed extensive experimentation with DSCRETE using a wide range of real-world commodity application binaries. Our results show that DSCRETE is able to recover a variety of application data — e.g., images, figures, screenshots, user accounts, and formatted files and messages — with very high accuracy. The raw contents of such data would otherwise be unfathomable to human investigators.

2.1 Overview

2.1.1 Key Idea: Executable Code Reuse

DSCRETE is based on reusing the existing data structure interpretation and rendering logic in the original application binary. As a simple example, consider the Linux `gnome-paint` application. At the high-level, `gnome-paint` works as follows: An input image file is processed into various internal application data structures. The user then performs edits to and saves the image. To save the image, `gnome-paint` will reconstruct a formatted image from its internal data structures and write this image to the output file.

Later, if a cyber forensic investigator wanted to recover the edited image left by `gnome-paint` in a memory snapshot, DSCRETE would be used to identify and automatically reuse `gnome-paint`'s own data structure rendering logic. First, DSCRETE will identify and isolate the corresponding data structure printing functionality within the application binary. For brevity, let us refer to this printing/rendering component as the function P. P should take as input a data structure instance and produce the human readable application output which is expected for the given data structure. In the case of `gnome-paint`, this component is the `file_save` function. It takes as input a `GdkPixbuf` structure and outputs a formatted image to a file. Note that P may not be a function in the programming language sense, but instead a *subsection of the application's code* responsible for converting instances of a certain data structure into some human-understandable form (e.g., output to the screen, file, socket, etc.).

Once P is identified, DSCRETE will reuse this function to create a *memory scanner+renderer* (or "scanner" for short) to identify all instances of the subject data structure in a memory image. If P is well defined for the input data structure, then one can expect P to behave erroneously when given input which is not a valid instance of that data structure. Under this assumption, we can present each possible location in the memory image to P and see if P renders valid output for the data structure of interest. We note that should an investigator alternatively choose to use a signature-based memory scanner to locate data structure instances, the DSCRETE-generated scanner *is still able to render* any located instances.

2.1.2 Overview of DSCRETE Workflow

Figure 2.2 presents the key phases and operations of DSCRETE. The first input is a binary application for which an investigator wishes to recover application data of interest from a memory image. To avoid compatibility issues (such as different data structure field layouts), this binary should be the same as the one that has contributed to the memory image.

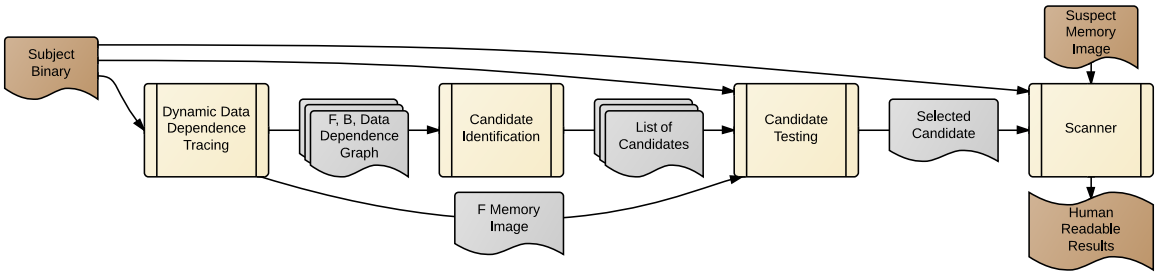


Figure 2.2.: Overview of DSCRETE workflow.

The subject binary is then executed under instrumentation to identify the code region responsible for converting a specific data structure into application output (the function P defined earlier). We refer to this phase of DSCRETE as "tracing," and the details of this step are presented in Section 2.2.1. In the next phase, "identification" (Section 2.2.2), a *graph closure algorithm* is used to formulate a list of possible candidates for P. Each candidate is tested, by the "tester" component (Section 2.2.3), with a ground truth data structure instance to determine if it can serve as a viable memory scanner.

Once the specific application logic (P) is identified, DSCRETE packages this logic as a *context-free memory scanner* (Section 2.2.4), which will be presented to forensic investigators to scan and interpret memory images in this and future investigations involving the same application. We point out that the first three phases (tracing, identification, and tester) are a one-time procedure internal to DSCRETE and do *not* involve field investigators who will be using the DSCRETE scanners in their practice.

It is important to note that, unlike signature-based memory scanning techniques, we do not attempt to find and return the raw contents (bytes) of identified data structure instances in a memory image. Instead, we aim to present the investigator a set of *application-defined outputs* that would naturally be generated by the subject application, had it executed P with the data identified in the memory image. We emphasize that DSCRETE does *not* infer data structure definitions (unlike [15, 16]), nor does it derive data structure signatures (unlike [17]).

2.1.3 Assumptions and Setup

Firstly, we assume that when producing DSCRETE-based memory scanners (typically the task of a central lab of a law enforcement agency), the subject binary can be executed. This includes recreating any execution environment (i.e., operating system and application version, required libraries, directory configurations, etc.) which the application requires. We believe that this assumption is not overly difficult to realize. In a real forensic investigation, such runtime configuration information can be collected via preliminary examination of suspect or victim machines. Additionally, our dynamic instrumentation requires that address space layout randomization (ASLR) be turned off during the production of the DSCRETE memory scanners (i.e., only the investigator's personal workstation, *not* the suspect machine under study). The reason for this will become clear in Section 2.2.3.

Secondly, we assume that the OS kernel's paging data structures in the subject memory image are intact. This is a similar assumption made by many previous memory forensics projects [8, 10, 17]. We require this because DSCRETE takes as input only the subject application's memory session from the suspect machine. For our evaluation, we extracted the memory pages directly from running applications — which is preferred when an investigator has physical access to a suspect's machine. However in many forensic investigations only the memory snapshot and hard disk image are available. In this case the Volatility [8] `linux_proc_maps` and `linux_dump_map` plug-ins (or `memmap` and `memdump` for Windows) can be used to identify and extract process pages and mapping information from a whole system memory image.

2.2 Design

2.2.1 Dynamic Data Dependence Tracing

The first phase of DSCRETE, "tracing," collects a dynamic data dependence trace from the subject application binary. This trace must contain some portion of

the future scanner's code (i.e., the code responsible for rendering a data structure of interest as human-understandable output). To collect this trace, we (as the central lab staff producing the scanners for field investigators) interact with the application to perform the following actions:

1) Create and populate an instance of the data structure used to store the data of interest. However, we make no assumptions on the knowledge about this data structure. We only assume that *some* data structure exists in the application which holds forensically interesting information in its fields.

2) The data structure of interest must be emitted as observable outputs (e.g., to files, network packets, or displayed on screen). This is to allow the scanner production staff to express their forensic interest by marking (part of) the output.

Again let us use `gnome-paint` to illustrate this procedure. To accomplish Step 1, we only need to execute `gnome-paint` with some input image. This will cause `gnome-paint` to create and populate numerous internal data structures to store the image's content. To accomplish Step 2, we only need to save the image to an output file. `gnome-paint` will process the image for output and call the GDK library's `gdk_pixbuf_save` function with the image's content as a parameter. While this may seem like a highly simplified example, the case studies in Section 2.3 show that in general we do not need to perform lengthy or in-depth interaction with an application to accomplish these two requirements.

Meanwhile, DSCRETE will be collecting each instruction's data dependence and recording any library functions or system calls invoked by the application as well as their input parameters. This is used to later identify which known external functions, specifically those which emit data to external devices, were invoked with the forensically interesting content as a parameter (`gdk_pixbuf_save` from our `gnome-paint` example). Note that since our analysis is at the binary level without symbolic information, we consider a parameter to be any memory read inside a function that depends on a value defined prior to the function's invocation. The memory may be accessed inside the function, subsequent functions, or as an argument to a system

call[2], and the content read is logged as parameters. We exclude any memory not previously written to by the application or a previous library function, allowing us to ignore any memory which is private to the library function and not related to the parameter (i.e., the transformed data structure). This logging results in an output file containing the list of invoked external functions and parameters to each (similar to the Linux `strace` utility).

It is important to note here that DSCRETE saves a snapshot of the process's stack and heap memory at the invocation of any external library function which leads to an output-specific system call (i.e., `sys_write`, `sys_writev`, etc.). We (as the forensics lab staff) may, optionally, further specify individual library functions for which a snapshot should be taken. For example, if we know that the forensic evidence will be rendered on the application's GUI, then we may choose to only log visual-output related library calls in the GTK library. These snapshots will later be used to test possible *closure points* (defined in Section 2.2.2).

Once Steps 1 and 2 are accomplished, we may terminate the subject binary and search the log of external function calls for one in which the forensically interesting data is seen as parameters. Once suitable functions are chosen, DSCRETE only needs to identify which bytes of the parameters for those function invocations are of forensic value.

The chosen function invocations and set of parameter bytes will be important for two reasons: First, the parameter bytes will serve as the source nodes in our data dependence graph. Second, the function(s) will be used as the termination point for our scanner and the corresponding bytes will be the *output* of the scanner. For brevity, these functions will be referred to as F and the selected forensically interesting bytes of F's parameters as the set B. For our running `gnome-paint` example, consider `gdk_pixbuf_save` as F and the image buffer it prints to the output file as B.

[2]We assume that system call interfaces are known and thus we can mark which parameters and memory ranges are read and which are written to.

Finally, a data dependence graph is generated using the trace gathered during dynamic instrumentation. The graph begins with the instructions responsible for computing the bytes of B as source nodes. Then in an iterative backwards fashion, any instruction which a graph node depends on is also added to the graph. Eventually, the graph will contain any instruction instance which led to the final value of B's bytes. This process is identical to that of typical dynamic slicing [19], we just chose to ignore control dependence as it is not required for identifying the functional closure (to be described next).

2.2.2 Identifying Functional Closure

Given F, B, and the data dependence graph, DSCRETE must find a *closure point* for the rendering function P. We define a closure point as an instruction in the data dependence graph which satisfies: 1) It directly handles a pointer to the forensically interesting data structure and 2) Any future instruction which reads a field of the data structure must be dependent on the closure point. This leads to the nice property that by changing the pointer handled by the closure point, we can change the data output by P. Returning to the `gnome-paint` example, the closure point is the instruction which moves a `GdkPixbuf` pointer into an argument register during `file_save`.

However, without source code or the effort of reverse engineering the subject binary, we cannot know the closure point for certain a priori. In fact, there may be multiple closure points in a program, any of which will satisfy our criteria above. To find a valid, usable closure point we use a combination of a graph closure algorithm and heuristics to output a list of *closure point candidates*. Each candidate is a tuple of the following: the address of an instruction which may satisfy the above criteria, the register or memory operand which it stores a pointer to, and the value of that pointer from the data dependence trace taken during tracing (Section 2.2.1).

Algorithm 1 Identifying *Closure Point* Candidates

Input: DataDepGraph(V, E), p

Output: Candidates[]

```
SubGraphs[] ← ∅
Previous_Candidate ← ∅
for node n ∈ V in Reverse Temporal Order do
    G(Vn, En) ← ∅                                  ▷ Build subgraph rooted at n
    Vn ← {n}
    for (n, t) ∈ E do                              ▷ Each t that depends on n (may be ∅)
        Gt(Vt, Et) ← SubGraphs[t]                  ▷ SubGraph rooted at t
        Vn ← Vn ∪ Vt
        En ← En ∪ Et ∪ (n, t)
    SubGraphs[n] ← G
    if Is_Store_Instruction(n) then                ▷ Apply heuristics to n
        val ← Stored_Value(n)
        loc ← Store_Location(n)
        if Is_Possible_Pointer(val) then
            if |SubGraph[n]|>|SubGraph[Previous_Candidate]| then
                Candidates ← Candidates ∪ (n, loc, val)
                Previous_Candidate ← n
    if |SubGraphs|> p%×|DataDepGraph| then
        break                                      ▷ Only consider p% of DataDepGraph
```

We call this phase "candidate identification." The algorithm to identify closure point candidates is given in Algorithm 1. Starting from each byte in B, the algorithm steps through the data dependence graph in reverse temporal order (i.e., from the last instructions executed to the first). For each node visited (n) the algorithm builds a graph containing all previously visited nodes which depend on n (G in Algorithm 1). Essentially, graph G will resemble a subgraph rooted at n with its leaves accessing some bytes of B.

For each node n added to these subgraphs, the algorithm performs the following heuristic checks; any node which passes these checks is considered a closure point candidate. First, n must store a value (either to a register or memory location) which could be a possible data structure pointer (any integer value that falls within a memory segment marked readable and writable). Second, the size of the dependence subgraph rooted at n must be larger than the previous candidate's subgraph. The intuition here is that a correct closure point will take as input a pointer to a data structure instance, and store this pointer to be reused by the rendering function P. Thus for the part of the data dependence graph responsible for rendering a data structure instance, the largest subgraph must have the closure point at its root. Consider a data dependence graph for the `file_save` function from `gnome-paint`: The largest subgraph of this data dependence graph should be rooted at the input `GdkPixbuf` pointer.

Another heuristic is to stop the algorithm after only a small percent of the data dependence graph is analyzed. Note that the data dependence graph contains instructions from F back to the application's `main` function. Further, P will be close to F in the graph and significantly smaller than the rest of the application's code. This percentage is taken as a configurable input (p in Algorithm 1) and is set via a forward iterative approach. In our evaluations in Section 2.3, we started with a p value of 1 and incremented p until a valid closure point was found. Even in the extreme case (`top`), p was never more than 10 and was often less than 5.

In all of our evaluations, the number of candidates never exceeded 102 and was often below 30. Additionally, as will be explained in the next section, we never need to verify (or even see) any of the candidates. The testing of candidates is done mostly automatically.

2.2.3 Finding the Scanner's Entry Point

To test each closure point candidate, DSCRETE will run a modified version of the memory scanner described in the next section. This modified scanner, named the candidate "tester," takes as input: 1) the known end point of the scanner (i.e., F), 2) the memory image taken when F was executed, 3) the list of candidates, and 4) the subject binary. The modified scanner will treat F's memory image as the "suspect" memory image to scan. We assume that this memory image contains a valid instance of the data structure which held the data seen in B because the application was in the process of rendering/emitting this data structure instance's fields when the memory image was captured.

The candidate tester will re-execute the subject binary from the beginning, but before the process is started the scanner maps the "suspect" memory image's segments into the address space. Each segment (a set of pages) is mapped back to the address from which it was originally taken[3]. This ensures that pointers in these memory segments will still be valid in the new process's address space. Note that ASLR is disabled during DSCRETE operations. At this point, the new process is unaware of the added memory segments and executes normally using only its new allocations. Later, we will intentionally force the new process to use a small portion of the old process's data session to test closure point candidates, a technique we call *cross-state execution* (discussed in Section 2.2.5).

[3]We have not seen any cases where critical segments overlapped. This is because the segments are being mapped into ranges usually reserved for heap and stack space. Since these segments are almost universally relocatable the new process is simply allocated pages around our memory image.

In the new execution, the forensically interesting data seen in this run of the application should be altered (e.g., executing `gnome-paint` with a different image). This will later allow the user to easily determine which candidate's output is correct (from the data structure in the memory image).

The application runs until a closure point candidate instruction is executed. Here, the tester forks an identical copy-on-write child of the subject application to perform the actual scan; the parent process will be paused until the child has completed. The scanner looks up which register or memory operand this candidate stores its pointer value into and overwrites this location with the pointer's value stored in the candidate. Note that if this candidate is a correct closure point, then the stored pointer value is a valid pointer to a data structure instance in the mapped memory image. This assumes that the data structure instance is not corrupted from the beginning of the rendering function P (for which this candidate may be an entry point) to the invocation of F. Since all candidates are reasonably close to the invocation of F (within $p\%$ of the total trace size), we find that this is never a problem in practice.

Further, if this candidate is a correct closure point, the child process will now execute P, access the old process's memory segments (via the changed pointer value), generate the same bytes for B, and invoke F with these bytes. Imagine that, for our `gnome-paint` example, this candidate is the instruction which moves a `GdkPixbuf` pointer into a register during `file_save` (P). Now `file_save` will execute in the child process with the `GdkPixbuf` structure inside the memory image and should call `gdk_pixbuf_save` (F) with an identical image as was previously rendered (B). Also, recall that the forensically interesting information seen in the new run is altered. This is to easily partition between output generated from the memory image and output from the new execution of the application.

During testing, if the child process crashes after the pointer replacement, then the candidate is assumed incorrect and thrown out. When the F function(s) execute to completion (recall that in our `gnome-paint` example F is `gdk_pixbuf_save`) then the content given as input to F is recorded as a result for this closure point candidate test.

An example of this recorded output is given later in Figure 2.6 (Section 2.3.3). The end of a scan is determined as follows: When F is a single function invocation, the child process is killed after F returns. If F consists of multiple invocations, the scan continues until the execution call stack returns to a point before the closure point. The parent process is then resumed, and this is repeated until all candidates are tested. A candidate is considered a valid closure point if it has accurately recreated the bytes chosen for B.

2.2.4 Memory Image Scanning

Once the data structure rendering function P has been identified, DSCRETE can build a memory scanning+rendering tool out of the subject binary. In fact, the production memory scanner is quite similar to the modified scanner used for testing candidates in the previous section. The difference is that we do not know where in a suspect memory image the data structures may be. The input to the memory image scanner tool are: 1) the chosen entry point and exit point of the printing function P, 2) the subject binary, and 3) the suspect memory image (as described in Section 2.1.3).

Again the scanner will re-execute the subject binary with the suspect memory segments mapped back into their original placements. Like before, the suspect memory segments will not be used until scanning begins, and until then the process executes using only its new allocations. With the same application input from candidate testing, the execution will reach P's entry point, where the scanner pauses the application. For each address in the memory image, the scanner will fork an identical copy-on-write child and assign P's pointer to the next address in the memory image. In essence, the scanner is executing P with a pointer to each byte of the suspect memory image. The scanning child process executes until P's end point (as defined in the previous section) and then P's output is recorded to a log or the child process crashes.

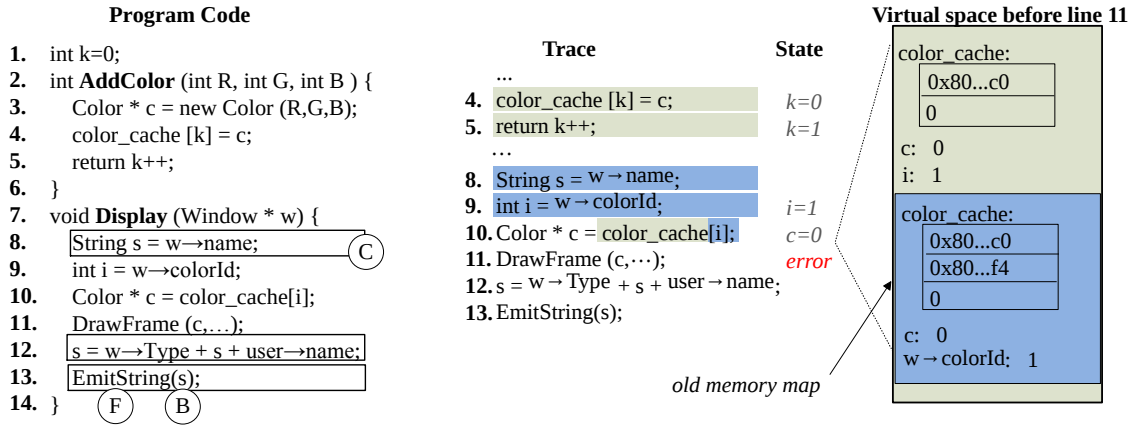
The intuition behind re-executing the application from the beginning is to automatically rebuild any dependencies required by P. DSCRETE requires that P's only input be a pointer to a possible data structure. In reality, P may depend on multiple parameters set up by the application prior to the closure point. By re-executing the application from the beginning, we ensure that any other dependencies P has are taken care of before the scanner injects a data structure pointer.

The execution of P is done in a child process to isolate side effects. Not surprisingly, the vast majority of addresses will cause invalid memory accesses or other exceptions, and by scanning each byte in a separate process the scanner ensures that side effects do not contaminate future scans or global values. To speed up scanning, multiple child processes can be spawned to run in parallel.

In some rare cases, P is too simple (performs too little input processing) to crash on invalid input. For such cases, we allow for a user-defined post-processing phase. We still assume no use of source code or reverse engineering effort, but the user may perform sanity checks based on the known format or value ranges for an application's output. For instance, in our `top` case-study we had to remove any output which had a negative process ID or blank user or process name field. In our experiments, only three cases — `CenterIM`, `top`, and `Firefox VdbeOp` — required any post-processing. Further, this only occurs for very simple textual P functions — complex cases such as those requiring content reverse engineering naturally involve more strict parsing and input sanitization.

2.2.5 Cross-State Execution

DSCRETE maps one process's address space into the address space of another. Further, when DSCRETE executes the function P, this code will evaluate data in both the old and new address spaces. Once DSCRETE replaces a data structure pointer at the closure point, the scanning process will then access fields from the data



(a) Program dependence. (b) Cross-state execution.

Figure 2.3.: Example for cross-state execution.

structure in the old address space while still using stack and other heap objects in the new address space.

Ideally, any sub-execution that depends on the closure point would exclusively access the state from the old address space. In other words, we expect the continuation after the pointer replacement would consist of two disjoint sub-executions, one corresponding to running P on the old address space and the other corresponding to the rest of the execution exclusively on the new address space. However, due to the complex semantics of real world programs, such separation may not be achievable. There are two possible problems: 1) An instruction execution may depend on state from both address spaces, resulting in some state that is infeasible in either the original or the new execution. We call such instructions *confounded instructions*. 2) Since the old memory snapshot may not be complete, an instruction may access a location in the old space that is not mapped in the new space. Note that this location may now correspond to a valid address in the new space such that the access becomes one to the new space. We call this a *trespassing instruction*. Both could cause crashes and hence false negatives.

Consider the example in Figure 2.3. Figure 2.3(a) shows two functions[4]. The first function (lines 2 - 6) creates a `Color` object and adds it to the `color_cache`. The other (lines 7 - 14) renders a window, including drawing the `Color` to a frame and emitting the window title as a string. Note that different executions may add different `Color` objects to the cache. Specifically, the number of `Color` objects and their order vary across executions. Later, the window rendering function will look up a `Color` object from the cache using its id.

Assume we (as forensics lab staff) mark the `EmitString()` function at line 13 as F and `s` (the window title) as B. Following the candidate identification algorithm, we compute the backward data-dependence of B as those boxed statements. We further identify line 8 as the closure point candidate.

However, when we test this candidate, cross-state execution leads to undesirable results if not properly handled. Let us assume that two `Color` objects were cached during the original execution, whereas only one `Color` is added in the candidate test execution. Figure 2.3(b) shows the trace of the candidate test execution on the left, and, on the right, it shows the state of the new address space right before the execution of line 11. Note that the pages of the old address space are mapped inside the new address space. Each executed statement in the trace is colored based on the address space it operates on. Particularly, lines 4 and 5 execute before the pointer `w` is replaced at line 8, and hence belong to the new space. In contrast, lines 8 and 9 belong to the old space, as their values are loaded from locations derived from the replaced `w`. Line 10 is a confounded instruction, as the array `color_cache` belongs to the new space while `i` belongs to the old space. As a result, an invalid color is loaded, leading to a crash. However, observe that lines 10 and 11 are not in the data dependence of B, as such we could potentially skip them.

Therefore, given a closure point candidate C and the corresponding termination point F, DSCRETE scans the original execution trace from C to F during the can-

[4]Our discussion is at the source code level for readability, whereas our design and implementation assume only the application binary.

didate identification phase. For each address dereference it encounters, it tests if the address is exclusively dependent on the pointer parameter at C. If not, it is a confounded dereference. DSCRETE further tests if the dereference is in the data-dependence graph of B, and if not, marks the instruction as an irrelevant dereference to be skipped during test execution and later scanning executions. In practice, we observed confounded memory dereferences in only one of the cases we studied.

Handling trespassing instructions is relatively easier. Given a closure point candidate C and its termination point F, DSCRETE scans the original execution trace from C to F and marks each address dereference that it encounters and is dependent on the pointer parameter at C. At runtime, if a marked dereference accesses a location in the new space, it is a trespassing access and can be skipped.

2.3 Evaluation

DSCRETE leverages the PIN binary analysis platform [20] to perform instrumentation. Since PIN executes before the subject binary is loaded, this allows us to map the memory image into the new process's address space before the operating system's loader can claim stack and heap regions. DSCRETE relies on minimal OS-specific knowledge (i.e., system call and application binary interface definitions), thus DSCRETE can easily be ported to any operating system that PIN supports. In the remainder of this section, we present results from evaluating DSCRETE with a number of real-world applications and focus on a subset which highlight the use of DSCRETE and a few critical observations.

2.3.1 Experimental Setup

Our evaluation used a Ubuntu 12.10 Desktop system as the "suspect" machine. Each application was installed on the machine and interacted with by the authors to generate sufficient allocations and deallocations of data structures. We used `gdb` to capture memory images from the application periodically during the system's use. To

attain ground truth, we manually instrumented the applications to log allocations and deallocations for data structures corresponding to the output of forensic interest (i.e., B in Section 2.2.1). This log was later processed to measure false positives (FP) and false negatives (FN). For analysis, we employed a Ubuntu 12.10 virtual machine. To recreate the suspect machine's running environment, we copied the applications and needed configuration files from the suspect machine's hard disk. We then performed all forensic investigation within the virtual machine.

2.3.2 Function Identification Effectiveness

This section presents results of isolating the data rendering function P in each tested application. From the `CenterIM` instant messenger, we target the component which emits the user's login and password (still in plain text) to an SSL socket. Also, given the importance of image content to investigations, we isolate image rendering functions from three common image editors: `convert`, `gnome-paint`, `gThumb`, as well as the `gThumb` GUI function which displays the current image's name to the window title. The output function of `gnome-screenshot` can allow an investigator to see what screen-shot a suspect was capturing. Additionally, we reuse `Xfig`'s figure saving P function to reconstruct a vector figure that was being worked on. The PDF saving functionality of `PDFedit` allows investigators to recover the edited PDF file. For internal application data, we identified P functions for `SQLite`'s query results and operations log (more on how these scanner+render tools are used later in this section). It is very common for attackers to tamper with server log files, so we isolated the `Nginx` webserver's connection logging function, thus an investigator can compare with the uncovered in-memory connections. Finally, for details on all running processes in a suspect system, we identified the process data printing logic in the `top` utility.

Table 2.1 shows a summary of the results from each of these applications. The application name and F function are shown in Columns 1 and 2 respectively. Column 3 details the forensically interesting data that were to be emitted by $F(B)$ and Column

4 shows the size of B in bytes. The percentage of the data dependence graphs used to generate candidates is shown in Column 5. Finally Columns 6 to 8 show the number of candidates identified by our algorithm (#C), how many of those produced any output (#O), and the final subset which accurately recreated B and could be used for valid closure points (#P), respectively.

From Table 2.1 we make the following observations: First, our algorithm/heuristics used to identify closure point candidates are accurate enough to limit the number of candidates to a reasonable search space. Although candidates are tested automatically during the candidate tester's execution, we aim to minimize the number of candidates to test. From Table 2.1, we see that 11 out of the 12 applications have less than 50 candidates. The only application with more than 50, `gThumb`, has 102, and as we see in Row 5 of the table, they are drastically narrowed down by the candidate tester. Manual investigation revealed that `gThumb`'s larger number of candidates was due to extra data dependencies caused by another parameter to its F function (`gtk_window_set_title`).

The second observation we make is that, of the total number of candidates identified, very few will be true closure points. This is intuitive since there is only one true entry to the P function in the application. Third, since the number of candidates which produced valid output is so small, it is relatively simple for a DSCRETE user to identify which candidate accurately reproduced B.

On average, each candidate testing component rendered application output for only three closure point candidates. The maximum, `convert`, rendered only seven outputs during candidate testing. Further, *more than half* of the applications produced ideal candidates — all candidates that rendered output were valid candidates. For the other five applications, about 45% of candidates which produced output accurately recreated the expected forensically interesting data (i.e., the new output matched that seen before). This shows that: 1) Visually inspecting candidate output is a reasonably quick and practical task and 2) DSCRETE can identify and validate closure point candidates with high accuracy.

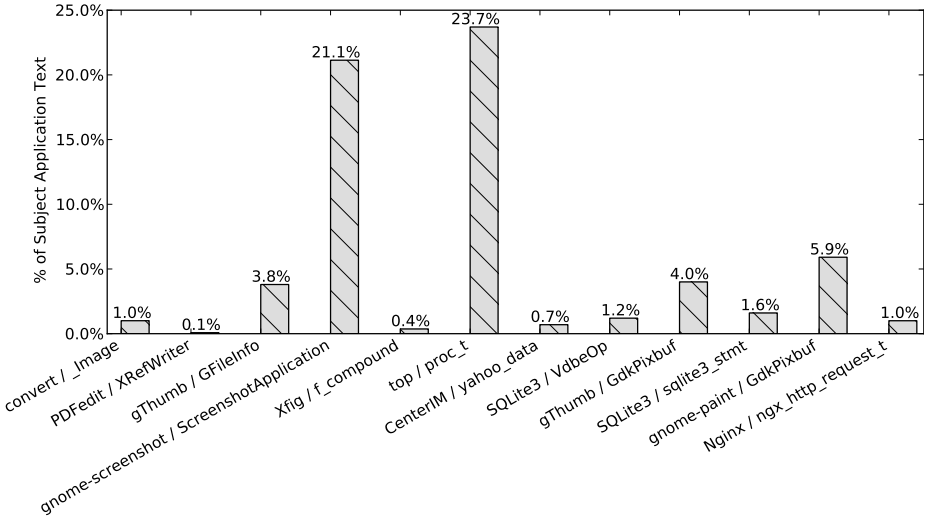


Figure 2.4.: Normalized size of P vs. entire binary code.

Table 2.1 shows that it is not uncommon for multiple correct closure points to exist for a P function. Manual investigation revealed that this is caused by two program features: nested data structure pointers and register-to-stack spilling. In the nested data structure situation, if a data structure A has a pointer to structure B and P uses the B pointer within A, then either the A pointer or its internal B pointer may be valid closure points for P. For the register-to-stack spilling situation, a pointer to an input data structure is initially stored in a register, but when contention forces that register to be spilled onto the stack, either the initial register or its later stack-saved location may be used for closure points.

Table 2.1 also shows that a valid closure point is typically located in the bottom 5% of the data dependence graph. Thus, the actual rendering function being reused is often only a small percentage of the binary's text. Figure 2.4 shows the normalized percentage of the host binary which we reuse for each scanning function. The size of the reused code is measured as the total in-memory size of all unique instructions observed during all re-executions of P. `Top`, `gnome-screenshot`, and `gnome-paint` are outliers due to the relatively small size of the applications and the resulting dependence graphs.

SQLite P Functions. An interesting application of DSCRETE can be seen in the experiments with SQLite. For these experiments DSCRETE was used with the SQLite3 command shell and a homemade database file to find P functions for a database query's result (`struct sqlite3_stmt`) and operations log (`struct VdbeOp`). These data structures are defined by the SQLite3 library and exported to client applications. The P functions DSCRETE identifies would be used to build memory scanner+renderer tools which could discover those data structures and render their content in the same format as the SQLite3 command shell.

These scanners could then be used on memory images from *any application which uses SQLite*. Since these data structures are defined by the SQLite library, any application using SQLite should transitively allocate and use these data structures. Further, we are reusing the SQLite3 command shell's P functions, so even if an application never outputs the data held in these structures, we can still discover and interpret them using the more general SQLite memory scanners. In the next section, we show results from applying these scanner+renderer tools to memory images from Mozilla Firefox and darktable image editor.

2.3.3 Memory Scanner Effectiveness

Table 2.2 reports the effectiveness of the DSCRETE-generated scanner+renderer tools when scanning a context-free memory image from each application. The application name is shown in Column 1. The subject data structure (input to P) and the structure's size are shown in Columns 2 and 3[5]. The number of true instances in the suspect memory image is shown in Column 4[6]. Column 5 shows the total number of output generated by each scanner+render tool. Columns 6 to 10 show the number of generated output which are: true positives (TP) - backed by true data structure

[5]Such information was obtained via manual instrumentation, inspection, and reverse engineering only for the purpose of evaluation. DSCRETE does not need or have access to this information during operation.

[6]This includes all the data structure instances which were allocated and not yet released and overwritten when the memory image was captured.

instances, false positives (FP) and the percentage of FPs in the total output (FP%), and false negatives (FN) and the corresponding FN percentage.

This table shows that the P function identified by DSCRETE is almost always well defined. This allows DSCRETE to uncover and render valid data structure instances with 100% accuracy for most cases. Specifically, Table 2.2 shows that DSCRETE's scanner+renderer tools are perfectly accurate (i.e., no FP and no FN) in 11 out of the 13 cases. We analyze the two FP/FN cases in detail later in this section. More importantly, DSCRETE overcomes the data structure content reverse engineering challenge by displaying the results in each application's original output format. The test cases covered in Table 2.2 span a wide range of application data: usernames and passwords, images, PDF files, vector-based graphics, as well as formatted and unformatted textual output. This portrays the generality of DSCRETE and represents several key types of evidence that would be very difficult (if at all possible) to reconstruct from raw data structure contents.

Table 2.2 shows that many of the subject data structures are smaller than the resulting application output (B from Table 2.1). Our manual analysis of these structures reveals that 10 of the 12 data structures contain several pointers to other data structures used by P. This confirms our intuition that, in order to recover usable evidence from a memory image, numerous data structures must be uncovered and interpreted. Note that an investigator never actually sees any of these structures, but rather is presented only the application output rendered from the structures' contents. Figure 2.1(a) is one such example.

Another metric to report is the time taken to scan, which varies depending on: 1) the complexity of the rendering function P and 2) the size of the memory image being scanned. Figure 2.5 shows the scanning speed in bytes-per-second for each scanner function in our evaluation. During our experiments, the size of the applications' heaps ranged from 400KB to about 5MB, and total heap scanning time ranged between 15 minutes to just over 2 hours, with most taking about 30 to 45 minutes. Admittedly the scanning and rendering of evidence is slower than typical signature-based memory

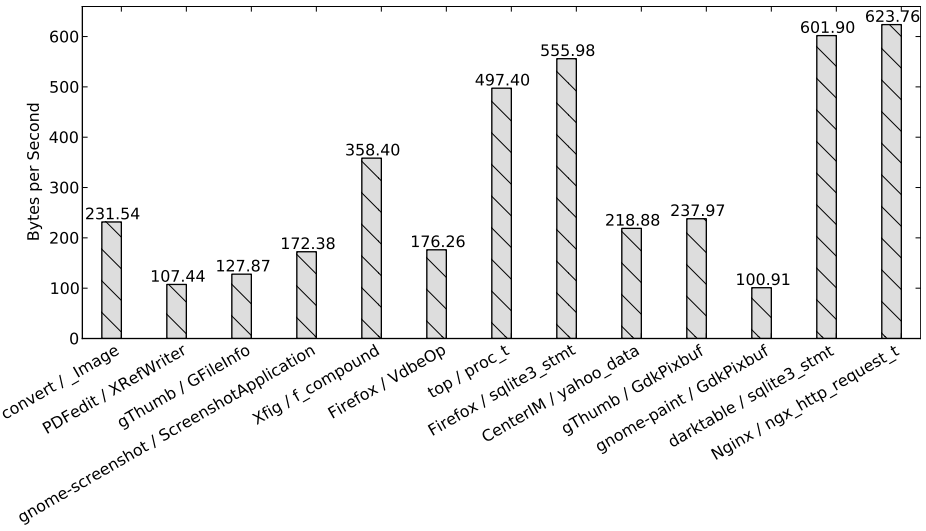


Figure 2.5.: Observed throughput of each scanner.

scanners, but still well within the typical time taken to process digital evidence, with the added benefit that evidence is presented in a human-understandable form. Ayers [21] points out that it may take "several hours or even days when processing average volumes of evidential data," which is confirmed by our collaborators in digital forensics practice.

False Positive and False Negative Analysis. We notice that only the gThumb and Firefox `VdbeOp` experiments experienced any negative results. Manual investigation into these two experiments' false negative results (i.e., true data structure instances not discovered by DSCRETE) revealed that those structures were allocated, but did not contain enough data to be rendered by P. They were either in the process of initialization or deletion or being used as empty templates by the application.

Interestingly, the Firefox `VdbeOp` case study (SQLite's operations log structure) represents a counter-example to our hope that P be well defined. In this case, P performs little parsing and no sanity checks on its input. A `VdbeOp` structure is essentially a set of seven integer values, and SQLite3 uses these integers as indices in a global string table, without any sanity checks. Since this P function performs such trivial parsing, a large number of false inputs produce typical SQLite3 Shell output.

We consider this a worst-case scenario for DSCRETE, and believe it is also the case for many other memory forensic techniques when facing such a trivial data structure.

We previously introduced one example of forensic data which would be uninterpretable without data structure content reverse engineering. The complex multi-level data structure representing a PDF file requires non-trivial processing to locate the fields which contain any usable PDF content. Further, many fields are encoded, compressed, or computed only when outputting the PDF file. In the remainder of this section, we present several other application case studies with DSCRETE.

Case Study: `convert`

This case study highlights DSCRETE's content reverse engineering capability for image data structures. The `convert` utility is used to apply various transformations to an image file. The source image file is processed and converted into internal data structures, (i.e., an `_Image` and array of `_PixelPacket` structures). Various transformations (such as scaling, blurring, etc.) are applied, and the pixels are re-composed into an image and written to a file. It would be considerably difficult to reconstruct the image from its in-memory representation, even with a deep understanding of these structures' syntax and semantics. However, DSCRETE is able to overcome this challenge by identifying and reusing the image output component (function `WriteImage`) which constructs an output image file from an input `_Image` structure.

As shown in Row 2 of Table 2.1, B (the image's content) was seen as an argument to the `fwrite` function. Using this, DSCRETE identified 18 closure point candidates in the bottom 9% of the data dependence graph. Of these candidates, 16 clustered around the handling of `_PixelPacket` structures in the image reconstruction routine, and the remaining 2 candidates handled the input `_Image` structure at the entry to the `WriteImage` function.

The DSCRETE candidate tester component eliminated 16 candidates which handled `_PixelPacket` structures. For the remaining two candidates, DSCRETE pro-

```
Candidate 1 ===== Scanning from 0x6a16c0:
  fwrite@libc ( 0x6ba360 ["<89>PNG<0d0a>"...], 1, 81902, 0x6b7320 [data] )
    Arg 1 written to file "c1_0x6ba360.out"

Candidate 2 ===== Scanning from 0x6a5c90:
  fwrite@libc ( 0x6ba360 ["<89>PNG<0d0a>"...], 1, 81902, 0x6b7320 [data] )
    Arg 1 written to file "c2_0x6ba360.out"
```

(a) Candidate test result log.

(b) Output image file for Candidate 1.

Figure 2.6.: Candidate testing output. (a) Each P function is shown, similar to the Linux strace utility, with parameters seen during invocation. If the tester component is set for file output, the file name is also printed. (b) Shows the output file for Candidate 1.

duced the log and application output shown in Figure 2.6. From Figure 2.6 we see that Candidates 1 and 2 successfully executed P (ending with `fwrite`). More importantly, DSCRETE accurately rendered the `_Image` data structure's content – presenting proof that both candidates form valid P functions which can reconstruct the image seen previously. As Table 2.2 shows, this P function was well-defined and the resulting scanner located and rendered the "image of interest" in the memory image with no false positives or false negatives.

Case Study: `Xfig`

The second case study is with `Xfig`, in which data content reverse engineering is essential to uncovering usable evidence from data structure instances. `Xfig` is a Linux-based vector graphics editor which defines several types of data structures for different drawable shapes (i.e., ellipse, spline, etc.). From `Xfig`, we intended to build a scanner+renderer tool to reveal the figure a suspect was drawing. Referring back to Table 2.1, DSCRETE located 9 closure point candidates in the bottom 1% of the data dependence graph. DSCRETE tested these 9 candidates and decided that 3 of them which rendered output were valid closure points. One of those was chosen (DSCRETE prefers the closure point highest in the dependence graph) to build a scanner+renderer for `Xfig`'s `f_compound` data structure.

An `f_compound` structure is a container for several shape structures. Each shape structure stores its dimensions, coordinates, color, etc. In order to reconstruct a figure, each of these shape structures must be recovered from a memory image, interpreted, and shape-specific rendering functions must be invoked. Existing signature-based memory scanners could present an investigator with a list of shape data structures instances from a memory image, but without the interpretation logic and shape-specific rendering, the investigator cannot see what the figure looks like. By comparison, the DSCRETE-generated scanner+renderer can locate the figure's `f_compound` structure, traverse all the contained shape structures (in the P function), and output `Xfig`'s original figure content. Table 2.2 shows that this P function is well-defined and recovered the figure's content with 100% accuracy from the target memory image.

What You Get Is More Than What You See

We observe that some applications will construct more data structures than they intend to display. Without content reverse engineering, these extra data structures would all need to be manually interpreted for investigation. DSCRETE intuitively

renders such additional evidence, allowing an investigator to quickly determine if it is forensically valuable.

In our experiment with `top`, the true number of `proc_t` instances is 382, whereas while executing `top` only 31 processes were displayed at a time. Since all 382 `proc_t` structures were in `top`'s memory image, DSCRETE was able to uncover and present each as they would have been displayed by the original `top` process.

Another example is `gThumb`, which displays an image being edited and other images in the same directory. `gThumb`'s memory contained valid data structures for 63 images: 56 GUI icons and 7 suspect images, and DSCRETE recovered them all, including the 7 suspect images. More importantly, 3 of the 7 suspect images were not being displayed by the GUI. Without DSCRETE, determining which raw data structures were icons and which were evidence would require extensive manual effort. With DSCRETE, an investigator can immediately see the distinction. Note also, that those GUI icons are not false positives. Instead, they are valid and relevant image data structures, because the investigator may use such GUI artifacts to infer which application screen the suspect was focusing on.

2.4 Future Expansion of DSCRETE

As mentioned in Section 2.2.5, cross-state execution may cause conflicting memory access patterns (i.e., confounded or trespassing instructions). DSCRETE selectively skips unnecessary instructions which may cause cross-state conflicts. However, this method is limited to the instructions recorded during tracing, and cannot reason about instructions that *were not executed*. Although we did not encounter such complications in our experiments, we do believe that they exist and will explore using static dependence analysis in the future.

DSCRETE relies on each application's own rendering logic to differentiate between valid and invalid input (data structures to be rendered). As we see in Section 2.2.4, this can be problematic if the rendering function performs very little input processing

and validation. Our experiment suggests that this problem exists for highly simplified data structures, which may still be of forensic value. Handling such data structures is our ongoing work. Additionally, since DSCRETE reuses application binary logic, an interesting problem is to handle data which contains *exploits* against the rendering logic.

Another current limitation which we leave for future work is replacing multiple input data dependencies for a rendering function. Currently, DSCRETE identifies and replaces only a single data structure pointer seen as input to P. However, it is assumable that a single application output be generated from *multiple* unrelated data structures. Although we have not encountered such need, the problem is realistic and requires enhancements to the closure point identification and the scanning algorithms.

Like many binary analysis-based tools, DSCRETE is not yet ready to handle self-modifying code or binaries with highly obfuscated control flows, which may cause problems in dependence detection or state crossing. However, these problems are common in malware programs and hence worth solving. One future direction is to develop DSCRETE on an obfuscation-resistant binary analysis platform (e.g., [22]).

The methodology used in DSCRETE is designed to operate directly on a target machine binary. As such, it is not applicable to programs written in *interpreted* languages (e.g., Java). Such programming languages add layers of indirection between the machine instructions observed by DSCRETE and the application's true syntax and semantics (i.e., data structures and rendering functions). Developing new techniques to handle programs written in interpreted languages is an intriguing direction for our future research.

Table 2.1.: Results from identifying applications' P functions (#C shows the number of identified candidates, #O shows how many of those produced output, and #P shows the final subset which are valid closure points).

Application	F	Forensically Interesting Data	Size B (bytes)	p%	#C	#O	#P
CenterIM	`SSL_write`	Username & Password	336	5%	46	1	1
convert	`fwrite`	Output Image Content	81902	9%	18	7	2
gnome-paint	`gdk_pixbuf_save`	Image Content	670900	1%	18	2	2
gnome-screenshot	`gdk_pixbuf_save_to_stream`	Screenshot Content	1139791	1%	5	4	3
gThumb	`gtk_window_set_title`	File Info Window Title	85	1%	102	4	2
	`gdk_pixbuf_save_to_bufferv`	Image File Content	20360	1%	10	3	3
Nginx	`write`	HTTP Access Log	181	5%	25	1	1
PDFedit	`fwrite, fputc`	Edited PDF Content	30416	1%	46	6	3
SQLite3 Shell	`fputs`	Database Query Results	19	2%	4	1	1
	`fprintf`	Database Op. Log	38	2%	17	5	1
top	`putp`	Process Data	132	10%	1	1	1
Xfig	`fprintf`	Figure Content	1001	1%	9	3	3

Table 2.2.: Results from DSCRETE-generated scanner+renderer tools.

Application	Subject Data Structure	Size (bytes)	True Instances	Total Output	TP	FP	FP%	FN	FN%
CenterIM	`yahoo_data`	160	1	1	1	0	0.0%	0	0.0%
convert	`_Image`	13208	1	1	1	0	0.0%	0	0.0%
darktable	`sqlite3_stmt`	272	1	1	1	0	0.0%	0	0.0%
Firefox	`sqlite3_stmt`	272	1	1	1	0	0.0%	0	0.0%
	`VdbeOp`	24	788	1384	753	502	40%	35	4%
gnome-paint	`GdkPixbuf`	80	51	51	51	0	0.0%	0	0.0%
gnome-screenshot	`ScreenshotApplication`	88	1	1	1	0	0.0%	0	0.0%
gThumb	`GFileInfo`	48	382	381	381	0	0.0%	1	0.4%
	`GdkPixbuf`	80	63	63	63	0	0.0%	0	0.0%
Nginx	`ngx_http_request_t`	1312	6	6	6	0	0.0%	0	0.0%
PDFedit	`XRefWriter`	344	1	1	1	0	0.0%	0	0.0%
top	`proc_t`	720	382	382	382	0	0.0%	0	0.0%
Xfig	`f_compound`	112	1	1	1	0	0.0%	0	0.0%

3 VCR: VISUAL CONTENT RECOVERY

Due to DSCRETE, the goal of memory forensics shifted from the recovery of low-level raw data to semantic contextual evidence which reveals a suspect's *actions and motives*. This shift coincided with a surge in the importance of smartphone photographic evidence for criminal investigations and legal proceedings. Today, photographs and videos regularly served as essential evidence in criminal investigations, the most famous of which was the recent United States Supreme Court case *Riley v. California* [23]. Further, law enforcement agents rely on photographic evidence as clues during on-going investigations. Today, smartphones provide easy access to a camera at all times, and not surprisingly, photographic evidence from smartphone cameras has become commonplace in real-world cases.

During an investigation, digital forensics investigators extract such evidence from a device. Historically, investigators focused on evidence recovery from non-volatile storage such as disk-drives, removable storage, etc. Investigators make forensic copies (images) of storage devices from a crime scene and perform analysis on the images back at the forensics lab. This analysis recovers a bulk of saved files (such as pictures and videos) that the investigator examines for evidence.

More recently, investigators have realized that non-volatile storage alone only reveals a subset of the evidence held in a system. The *contextual* evidence held in a system's *volatile* storage (i.e., memory) can prove essential to an investigation [24,25]. Memory forensics research has made it possible to uncover much of an operating system kernel's data from a context free memory image [8, 11, 14, 17]. Other work has focused on recovering data structure instances from applications using known in-memory value-patterns [9–11] or with the assistance of program analysis [15]. Unfortunately, the rapid pervasion of Android devices has rendered many tools inapplicable to smartphone investigations.

Like other digital evidence, photographic evidence which persists on non-volatile storage also lacks context or simply portrays an incomplete picture of a crime — again requiring memory forensics to fill the gaps. To this end, I have developed VCR[1], a memory forensics technique which integrates recovery and rendering capabilities for all forms of in-memory evidence produced by an Android device's *cameras*. VCR is based on the observation that all accesses to a device's camera are directed through one *intermediate service*. By designing VCR's evidence recovery function to target this intermediate service, VCR can automatically recover all forms of photographic evidence regardless of the app that requests it. This trend of centralizing critical services into intermediary processes (which we term *intermediate service architecture*) is widely used in the Android framework, and this chapter examines the digital forensics and security implications of such design with regard to the camera framework.

VCR's evidence recovery faces challenges, however, because the Android framework (known as the Android Open Source Project or AOSP) is often customized by smartphone vendors. To overcome this, VCR involves novel structure definition inference techniques which apply to the Android vendor customization domain — called *Vendor-Generic Signatures*. To the best of our knowledge, VCR is among the first to handle vendor-customized data structures inline as part of targeted evidence recovery.

Additionally, VCR-recovered evidence must be reviewed, cataloged as evidence, and presented to any (not technically trained) lawyer or official. Thus, VCR must transform the unintelligible in-memory photographic data into human-understandable images. Using an instrumentation based feedback mechanism within existing image processing routines, VCR can automatically render all recovered evidence as it would have appeared on the original device.

We have performed extensive experimentation with VCR using a wide range of real-world commodity apps running on different versions of the Android framework

[1]VCR stands for "Visual Content Recovery" and is a reference to the ancient videocassette recorder device.

and two new, commercially available smartphones. Our results show that VCR can automatically and generically recover and render photographic evidence from the phones' memory images — a capability previously not available to investigators — with high accuracy and efficiency.

3.1 Motivation

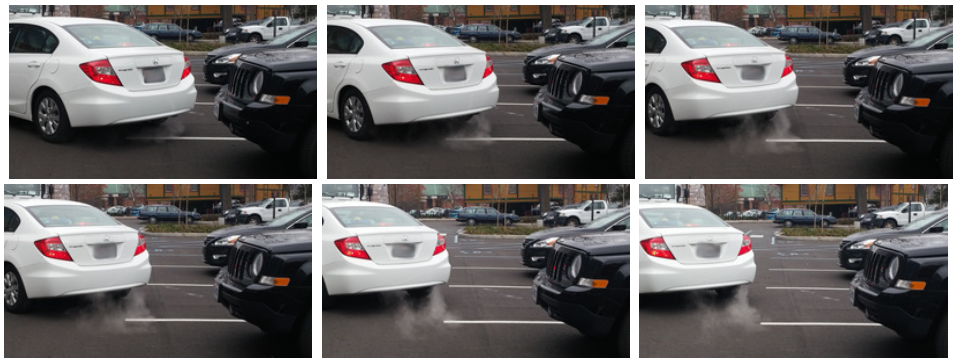


Figure 3.1.: Time-lapse effect in recovered preview frames *without explicitly taking a photo*. VCR recovers and renders these images as they would have appeared on the app's camera preview screen — the smartphone analog to a standard camera's view finder.

Smartphone cameras are employed in a variety of apps which we use everyday: taking photographs, video chatting, and even sending images of checks to our banks.

Criminals too have found many uses for smartphone cameras. To motivate the need for VCR, we quote Riley vs. California [23], a United States Supreme Court case involving smartphone photographic evidence:

> At the police station about two hours after the arrest, a detective specializing in gangs further examined the contents of the phone. The detective testified that he "went through" Riley's phone "looking for evidence, because ... gang members will often video themselves with guns or take pictures of themselves with the guns." ... Although there was "a lot of stuff" on the phone, particular files that "caught [the detective's] eye"

> included videos of young men sparring while someone yelled encouragement using the moniker "Blood." ... The police also found photographs of Riley standing in front of a car they suspected had been involved in a shooting a few weeks earlier. [23]

In the above quote, the detective explains how essential smartphone photographic evidence is to ongoing investigations. Further, our collaborators in digital forensics practice describe many other crimes in which such evidence can prove invaluable. In Section 3.3, we will consider smartphone photographic evidence in a (mock) case based on an invited talk at Usenix Security 2014 on battling against human trafficking [26].

Let us strengthen our adversary model by considering a more tech-savvy criminal than Riley — someone who deletes the image files from the device's storage or even removes the storage (e.g., external SD-card) and destroys it. Current digital forensics techniques would not recover any photographic evidence in such a case. Luckily, regardless of how tech-savvy the criminal may be, photographic evidence from the camera's most recent use remains in the system's memory. VCR gives investigators access to these last remaining pieces of photographic evidence.

A smartphone camera produces three distinct pieces of evidence: photographs, videos, and preview frames. Photographs are left in a device's memory when a user explicitly captures an image. When a smartphone records a video, individual frames are captured and sent to the requesting app — again leaving frames behind in memory.

Preview frames, however, are of particular forensic interest for a number of reasons. Preview frames are a smartphone's analog to a standard camera's view finder. When an app uses the camera, the app will, by default, display the camera's current view on the screen, allowing the user to accurately position the device for capturing the intended picture. Importantly, *whether the user captures a photo or not* the app will display the preview. This leads to the forensically important feature that: **Any app which only opens the camera, immediately leaves photographic evidence in memory**. Further, preview frames (and video frames) are captured continuously

and buffered until the app retrieves them. Thus *many frames will be present* in a memory image representing a *time-lapse* of what the camera was viewing.

Building from the scenario in Riley vs. California, imagine that Riley had carefully removed all photograph files from the smartphone's non-volatile storage or (more likely) was using an app which *does not save photograph files* such as a Skype video-call. In this case, the smartphone's non-volatile storage will not contain any evidence of the car suspected in the earlier shooting [23]. However, investigators could now use VCR to analyze the smartphone's memory image and recover the last images, videos, and preview frames left in the memory, which are likely the evidence the criminal is trying to hide.

Figure 3.1 shows some preview frames which VCR recovered from a smartphone's memory image. Notice that multiple frames are recovered and show the action of the perpetrator's car driving away (i.e., *temporal* evidence for investigators). Also note that these are *preview frames* and the smartphone user was *not actively recording video at that time*. Simply having the camera-using app open left photographic evidence in this memory image. It's easy to see how such evidence links the smartphone's owner to the car in the images (and hence to the shooting).

Our study reveals that this photographic evidence always persists in the smartphone's memory — without being erased or overwritten — until a new app uses the camera (filling the previous image buffers with new evidence). Thus, VCR will always have some evidence to recover. Note that these buffers are not app-specific, only containing frames from the most recent app which used the camera. More importantly, the buffers storing other media data (e.g., audio) are allocated from separate memory pools than the camera's buffers and thus cannot interfere with photographic evidence. Further, VCR is not specific to suspects' smartphones, investigators can apply VCR to memory images from a witness or victim's Android device as well, for instance to collect proof of the user's whereabouts.

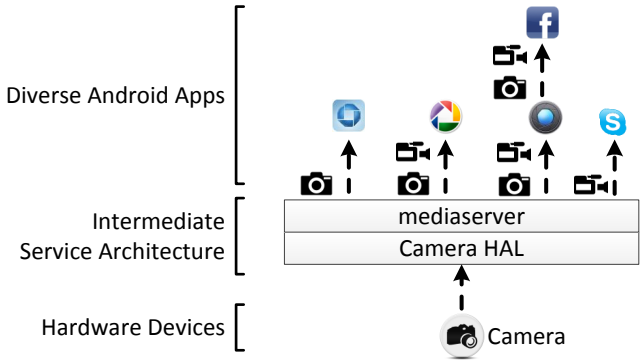


Figure 3.2.: Intermediate service architecture. The mediaserver acts as a mediator between the apps and the camera device. Also, some apps utilize the camera by requesting the default camera app to perform actual image captures (such as the Facebook app shown here).

3.1.1 Centralized Photographic Evidence

For many core services, Android has adopted an intermediate service architecture. Specifically, accesses to peripheral devices and system services are mediated by an intermediate process. For the camera(s) this process is called the *mediaserver*. Figure 3.2 presents a high level view of the intermediate service architecture, specifically for the mediaserver's components: Apps, the mediaserver process, and camera hardware abstraction layer (HAL). This high-level intermediate service design makes app development easier and abstract regarding the hardware back-end.

Intermediate services present a standard interface to the apps. Each service is designed to generically handle any vendor/hardware specific implementation beneath it. Most importantly, the AOSP defines generic data structures for the vendor's code to use in order to conform with the standard interface presented to the apps.

The key observation behind VCR's design is that any app which uses the camera *must transitively use the generic data structures to retrieve photographic data from the mediaserver.* This creates a unique opportunity for VCR[2]. By locating and recovering

[2]However, as we point out later, this also centralizes privacy-critical components and may benefit attackers as well.

these generic "middleware" data structures, VCR is able to reconstruct and render evidence *without any app-specific knowledge*. More importantly, VCR can remain mostly generic to any hardware-specific implementations because the camera HAL must also use the generic data structures to return photographic data to the apps. This is beneficial to VCR, which can now be designed in a more robust, generic way than tools that must recover data from individual (highly diverse) Android apps.

In fact, the mediaserver also delegates *audio requests* (accesses to speakers and microphones) and most media streaming. We note that photographic evidence is only part of the mediaserver's potential forensic value. VCR can be extended to extract other evidence formats from the mediaserver's memory.

3.1.2 Assumptions and Setup

VCR assumes that an investigator has already captured a memory image from an Android device. Previous research has designed both hardware [6] and software [7] acquisition tools to obtain a forensic image of a device's memory. VCR operates on memory images captured by any standard memory acquisition tool.

Similar to previous memory forensics projects [8, 10, 17] including DSCRETE, VCR assumes the kernel's paging structures are intact in the memory image. This is required because VCR operates only on the mediaserver process' memory session. Tools (e.g., [8]) exist to rebuild a process' memory space from a whole-system memory image.

3.2 Design

VCR consists of two phases: 1) identify and recover photographic data from an input memory image, and 2) transform the unintelligible recovered data into photographic evidence which investigators can review and present.

3.2.1 Recovering Evidentiary Data

Since photographic image buffers are encoded and indistinguishable from random data, brute-force scanning for the buffers would return countless false results. VCR adopts a more robust algorithm: for each type of evidence (preview frames, photographs, and video frames), VCR locates and recovers a distinct group of interconnected data structures, one of which contains the image data. For simplicity, we refer to such groups of interconnected data structures as "data structure networks."

Ideally, VCR would only need to verify the points-to invariants between the targeted data structures (i.e., each pointer field within each structure points to another structure in the network). In this way, each recovered data structure attests to the validity of the network, thus the located network is not a false positive. However, for key reasons described below, points-to invariants alone are insufficient in this scenario.

The structures which VCR recovers form a closed network which is unfortunately too small to derive a viable points-to invariant signature. Instead, VCR must also employ value-invariant signatures for each data structure. However, due to vendor customizations, the structures' field positions and value-invariants cannot be fully known *a priori*.

Nearly every Android device uses a customized (possibly close-source) version of the AOSP. Device vendors make a proprietary copy of the AOSP repository and customize the low level framework (kernel, drivers) and high level utilities (GUI, standard apps). For data structures, vendors may add fields to store custom data, move existing fields to different offsets within the structure, or change the values that existing fields can be assigned (such as adding a new enumeration value). Specific to VCR, vendors modify the camera's allocation pools and internal operation (specific drivers, image processing, etc.). These modifications lead to different definitions of the data structures that VCR must recover.

Luckily, although vendors may customize the data structures, they must still conform to a "gold standard" in order to interact with unmodified portions of the

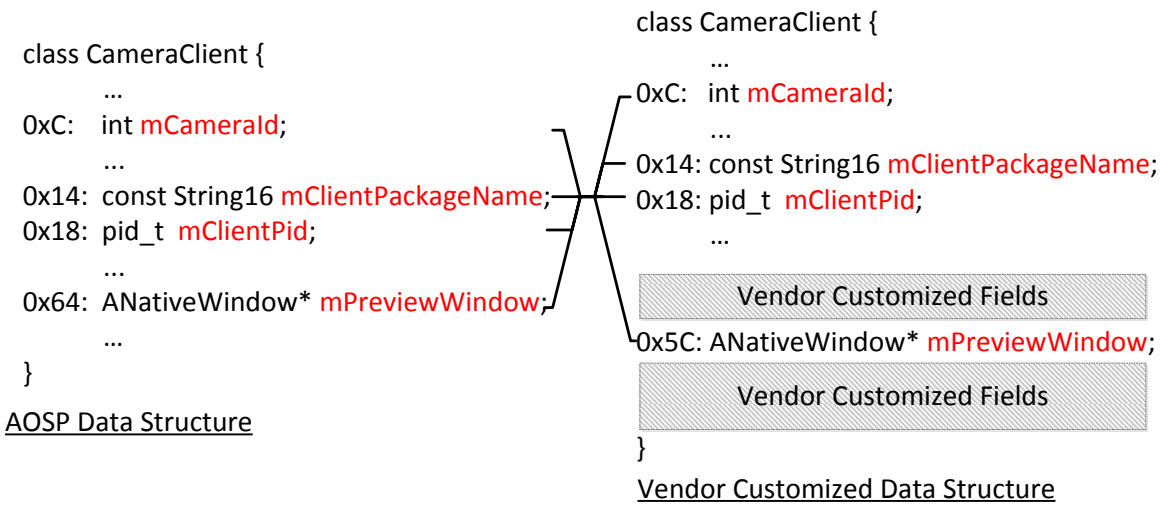


Figure 3.3.: AOSP vs. vendor customized structure.

AOSP. We use the term "gold standard" to refer to the many components of the AOSP that are not customizable (e.g., middleware libraries, core functionality, etc.), and thus vendor customizations must not remove structures and data fields **at the source code level** which other components rely on.

As an example, Figure 3.3 shows the CameraClient class from the AOSP versus the LG vendor customized version. The vendor customizations change the offset of the mPreviewWindow field, but *in order to interact with unmodified AOSP components* all the "gold standard" fields (which VCR relies on) must remain in the structure. In our evaluation we observed vastly different implementations of several data structures which VCR must recover.

After the vendor-customized source code is compiled, VCR loses access to the mapping between source code definitions and binary data structure layouts. Essentially we know that the fields exist, but cannot know *where they are* when locating data structures in a new memory image. To overcome this, VCR is prepackaged with *Vendor-Generic Signatures* (Section 3.2.2) for the customizable data structures. VCR then dynamically derives *Vendor-Specific Signatures* (Section 3.2.3) during data structure location and recovery.

Beyond vendor customization, VCR's generic signatures are also robust to changes *between AOSP versions*. When Google updates features in the AOSP, this also leads to changes in data structure layouts. In fact, several fields were added to the CameraClient class between AOSP versions 4.4 and 5.0. VCR's signatures however do not need to be updated because they can adapt to the input memory image. Further, it is easy to add additional signatures in the event that Google fully redesigns some data structure network.

3.2.2 Vendor-Generic Signature Derivation

VCR operates on only an input memory snapshot and assumes no source code availability. Thus VCR must adapt signatures for any necessary data structures dynamically. To this end, VCR comes packaged with a set of *Vendor-Generic Signatures*. Vendor-Generic Signatures are data structure signatures which contain invariants on the structure's fields but *do not have set locations (offsets in the structure) for those fields*. Specifically, we preprocessed the AOSP "gold standard" version of each data structure D_i which VCR must recover. For each field f_j within the "gold standard" D_i, a field constraint (described below) is built.

Field Constraints We define 4 *primitive constraints* to describe each field: 1) A Type/Size constraint defines the field's type definition (e.g., floating point, pointer, etc.) and in-memory byte size. Since VCR operates on binary data these constraints are essentially sanity-checks on the discovered memory locations. 2) Value Range constraints are value invariants specific to field f_j. 3) Field Offset constraints define where f_j is *likely* to be in D_i. For some fields (e.g., inherited from a superclass) we know the byte offset in D_i for certain, but for most fields we cannot know where the vendor's modifications moved them. 4) For pointer fields, Pointer Target constraints define a set of *other primitive constraints* on the pointer's target. Specifically, our confidence in f_j being a pointer to a data structure depends on the validity of the target data structure.

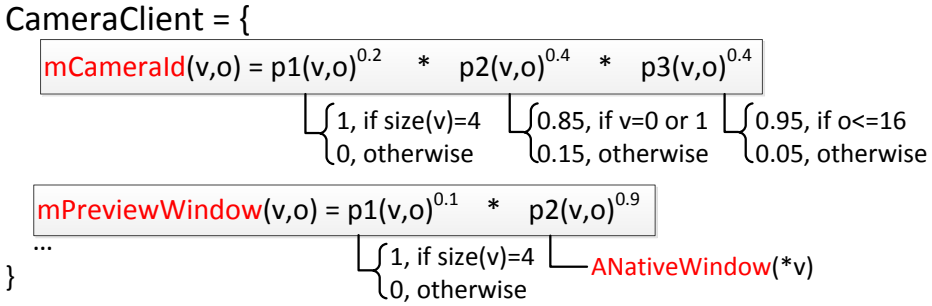


Figure 3.4.: Illustration of a partial CameraClient signature (an unordered set of field constraints).

Therefore, based on the AOSP definition of f_j in D_i, we automatically build a probability match function $p_i(v, o)$ for each primitive constraint. $p_i(v, o)$ defines the probability that a value v at byte offset o in a discovered data structure matches that constraint. We can then define a *field constraint* for f_j as:

$$f_j(v, o) = p_1(v, o)^{w_1} \times p_2(v, o)^{w_2} \times ... \times p_n(v, o)^{w_n} \tag{3.1}$$

where p_i is the i$^{\text{th}}$ primitive constraint for field f_j, and w_i is a corresponding weight to adjust for stronger constraints (the sum of all weights must be 1).

Therefore, the signature of the structure D_i is an *unordered set of field constraints*. The set is unordered because VCR cannot know the offsets of those fields in a vendor customized data structure *a priori*. During memory image scanning, VCR will order the field constraints to adapt to the vendor customizations (described in the next section).

Figure 3.4 shows part of a CameraClient signature. Notice that each field constraint includes a number of primitive constraints — for instance, the mCameraId field has constraints on its type and size (a 32-bit integer), value range (between 0 and 1 with high probability), and offset (with high probability in the top 16 bytes of the data structure).

Not all data structures which VCR recovers are vendor customizable. For these we rely on existing points-to and value invariant signature generation techniques to build a "hard signature." When a signature contains a pointer to one of these structures,

we set that pointer field's Pointer Target constraint value to 1.0 (i.e., pointing to a valid hard signature provides full confidence in that pointer field).

Field Dependence We notice that not all fields in a data structure are independent. For simplicity, we only consider dependence based on two (or more) fields' *location in a data structure*: (1) Non-pointer fields *of the same type* tend to be clustered (e.g., floating point width and height fields) and (2) Fields accessed consecutively in a C++ class's member function are likely to be defined next to one another.

For these two cases, a scaling factor (α) is applied to increase the match probability of the dependent fields when a signature matching maps them consecutively. Simply put, if VCR locates these fields next to each other in a potential signature match then we can be more confident in that match — compared to matching those fields in separate locations. Fields dependent by (1) above are given $\alpha = 0.2$ scaling factor (i.e., matching such fields consecutively increases the probability of the entire signature matching by a factor of 0.2). Conversely, we assume stronger correlation for fields dependent by (2) and thus set $\alpha = 0.8$.

Based on the signatures generated before, we update each field constraint of any dependent fields to account for the scaling factor. Here we use the function $dep(c_a, c_b)$ to denote that the field constraints c_a and c_b are dependent. Consider two matches for those field constraints a_i and a_j where each a_n is the n^{th} field in a potentially matching data structure instance. We define $P_{match}(c_a, a_i)$ as follows (where $c \rightarrow a$ denotes "c matches to a"):

$$P(c_a \rightarrow a_i | \forall c_b : dep(c_a, c_b) \wedge c_b \rightarrow a_j) = P_{match}(c_a, a_i)$$

$$\text{where } P_{match}(c_a, a_i) = \begin{cases} (c_a(a_i))^{\alpha} & \text{if } i = j-1 \text{ or } j+1 \\ c_a(a_i) & \text{otherwise} \end{cases} \tag{3.2}$$

Essentially, Equation 3.2 applies the scaling factor to c_a if c_a and c_b are dependent and c_b has previously mapped to a neighbor of a_i. Otherwise, the formula simplifies to the field constraint probability defined in Equation 3.1.

3.2.3 Memory Image Scanning

To use VCR to recover photographic evidence, investigators need only input a context-free memory image. VCR then employs a two-pass scanning algorithm. In the first pass, VCR marks all memory locations which match hard signatures (i.e., not vendor customizable and do not require probabilistic inference) — we refer to these as "hard matched" data structures. During the second pass, VCR uses probabilistic inference with our previously generated Vendor-Generic Signatures to construct *Vendor-Specific Signatures* to identify and recover true data structure instances.

Starting from the hard-matched data structures, VCR backward propagates confidence to all potential matches to Vendor-Generic Signatures. To calculate a match for a signature S, VCR first converts a candidate memory region (the region we want to map to S) into a set A of tuples:

$$A = \{(v_0, o_0), (v_1, o_1), \ldots (v_n, o_n)\} \tag{3.3}$$

where v_i is the i$^{\text{th}}$ value at offset o_i in the candidate memory region A. To match Vendor-Generic Signature fields, VCR may combine adjacent tuples to satisfy the field's type/size constraint. If no such match can be made, then the type/size constraint will yield a 0 probability. Later, we will use a_i to denote (v_i, o_i).

VCR then creates a permutation of the vendor-generic signature by computing the best fit mapping $S \rightarrow A$, using the following greedy algorithm: For each randomly chosen field constraint c_i, VCR matches a binary tuple a_j (from the remaining unmatched tuples in A) which maximizes c_i's match probability (i.e., $P_{match}(c_i, a_j)$). This repeats for each c_i until all field constraints have been matched. Yielding the final match probability equation for a signature S to a candidate structure A:

$$S(A) = P_{match}(c_0, a_0) \times P_{match}(c_1, a_1) \times \ldots P_{match}(c_n, a_n) \tag{3.4}$$

where the subscripts indicate order of matching (not order in the signature). Computing Equation 3.4 for a single $S \rightarrow A$ mapping yields VCR's confidence in that particular permutation of S for that candidate A.

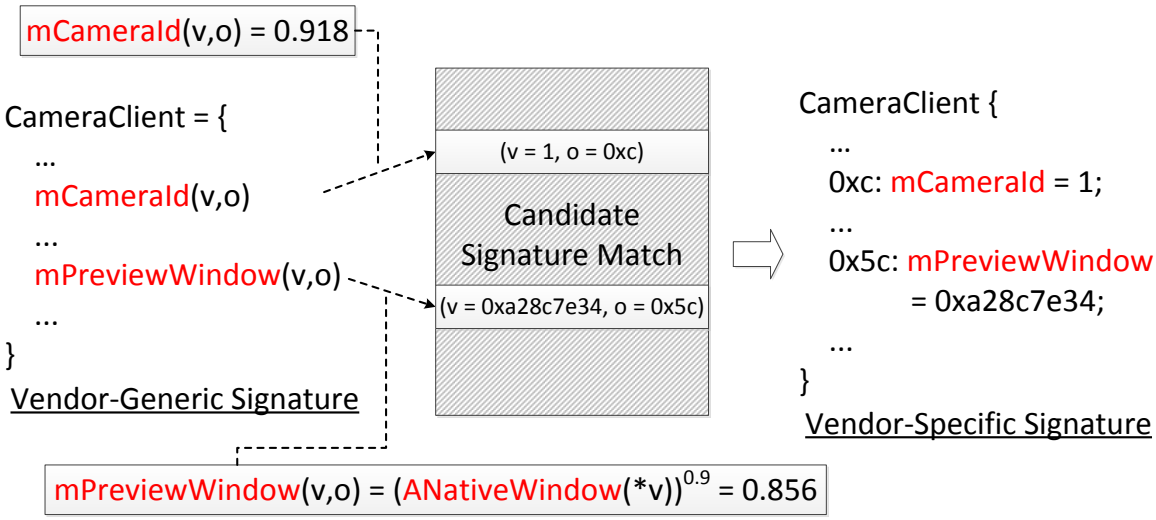


Figure 3.5.: Matching a candidate CameraClient instance via field constraint computation.

Figure 3.5 shows an example of matching the CameraClient signature's field constraints to a candidate memory location. Computing each field constraint for the best matching a_i yields one permutation of the CameraClient signature's fields for that candidate memory location.

Computing the match probability of Pointer Target constraints requires knowing the probability of the pointer target's match. Because matching is done via backward propagation, VCR only computes a signature's match probability once all of its Pointer Target constraints have been computed. Recall that if the target is a hard-match then this confidence is 1.0. Thus, VCR requires recoverable data structure networks to have hard-signatures at the leaves (which can be ensured automatically during signature generation).

VCR repeats the above greedy algorithm and Equation 3.4 computation independently for all candidate memory regions for a single signature. Note that the first greedy matching may not result in the correct mapping of c_i to a_i (resulting in different $S \rightarrow A$ mappings for different candidate memory locations). We observe that: for a final mapping of c_i constraints to be correct, it must be *constant across all*

discovered instances of that data structure. Thus only a single mapping of $S \rightarrow A$ can be correct for all candidate memory locations (i.e., A sets) for that signature.

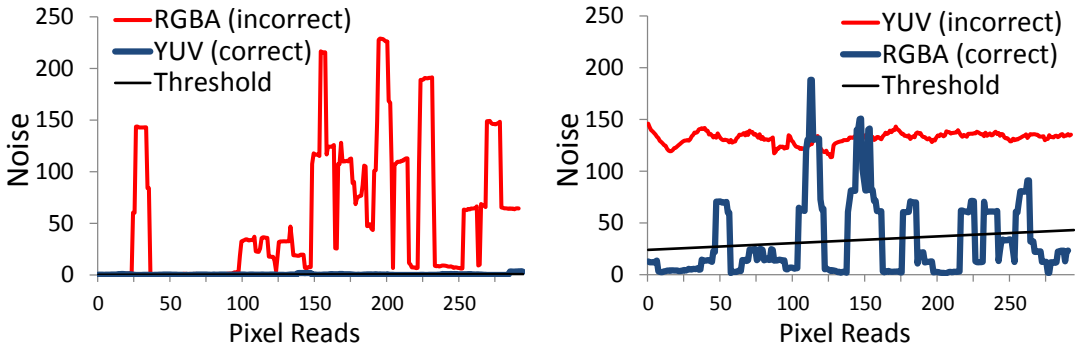
VCR iteratively repeats the above process for all candidate memory locations (choosing the $S \rightarrow A$ mapping which maximizes the probability of a match), until an optimal mapping is found across all candidates for a single signature. This final signature which VCR selects is referred to as a Vendor-Specific Signature (i.e., the Vendor-Generic Signature with set field locations). VCR will recover all candidate data structures which match the Vendor-Specific Signatures to a certain threshold. In our evaluation, we use a threshold of 0.75 since most candidates polarize with invalid candidates near 0.3 and valid matches near 0.8.

3.2.4 Rendering Evidence

Once VCR has recovered the structures containing photographic evidence, the data must be reconstructed into human-understandable images. This is essential as the raw contents of these photographic buffers would be unintelligible to forensic investigators.

Photographic data may be in any format that the app requests (e.g., NV21, ARGB, etc.). We observe, however, that apps (i.e., image buffer consumers) must have access to decoding logic for any format supported by the AOSP. Based on this, VCR automatically reuses the existing AOSP image decoding logic, but which decoding algorithm to use cannot be known *a priori.* VCR must determine (specifically avoiding burdening human users) which decoding is appropriate for each recovered image.

Image buffers, specifically photographs, are highly *periodic.* That is, the data values follow regular periods across the image's strides, height, and width. Based on this, image processing techniques often compute the *spacial locality* of the image's pixel values when performing image analysis [27]. VCR builds on this idea to *validate image decoding* by enforcing a periodic constraint on the image data as it is read by the decoding algorithm.



(a) YUV Buffer (b) RGBA Buffer

Figure 3.6.: Reading (a) YUV and (b) RGBA buffers with different decoding algorithms. Correct decoding algorithms will minimize the area under the noise curve. In fact, we can hardly see the YUV curve in (a) because its noise is always close to 0.

VCR instruments each decoding algorithm to verify that the values read from the input buffer follow a *periodic data constraint* (i.e., the spacial locality between each pixel value should be small and form a smooth curve). VCR attempts to decode each recovered image buffer with each available algorithm. During image decoding, VCR computes the Euclidean distance between the pixel values read from the input buffer [27] which we refer to as the decoding "noise."

Ideally, decoding an image buffer with the correct decoding algorithm should produce very little noise (i.e., the pixel values closely follow the periodic data constraint). In practice, images may contain local, sharp changes of color or brightness (which is particularly obvious when encoded in YUV format), so VCR computes a moving average of the noise values as a *noise threshold*. The algorithm which produces the smallest noise threshold is marked and the output of that decoding is presented to investigators as evidence.

For empirical comparison, Figure 3.6 shows graphs of two image buffers being decoded by the correct and incorrect algorithms. For simplicity, we only compare

two algorithms, but VCR considers all 19 decoding algorithms used by the AOSP camera framework.

Figure 3.6(a) plots the noise values for a buffer encoded in YUV format being decoded using the RGBA algorithm (red curve) and YUV algorithm (blue curve). We can see that decoding the YUV buffer with an RGBA decoder produces a large amount of noise, whereas decoding with the YUV (i.e., correct) algorithm produces so little noise that the blue curve is hardly visible. We also plot the noise threshold from the YUV decoding, but this too is always near 0.

In Figure 3.6(b), we plot an RGBA buffer being decoded as YUV (red curve) and RGBA (blue curve). In this case, we see that the correct decoding (RGBA) contains some noise but the incorrect decoding (YUV) induces 3 to 4 times more noise. Further, the RGBA noise threshold (plotted in black) is again always near 0.

Finally, all recovered buffers of a single type (i.e., preview frames, photographs, or video frames) must use the same encoding — because an app only specifies this encoding once. Thus as a final sanity check, VCR ensures that the chosen decoding algorithm minimizes the decoding noise across all image buffers. Because of the large noise disparity between correct and incorrect algorithms, in our evaluation VCR was able to identify the correct decoding algorithm in all of our test cases.

3.3 Evaluation

Our evaluation tested a variety of different Android devices as "devices under investigation." We performed evaluations with two new commercially available Android smartphones: an LG G3 and a Samsung Galaxy S4. Both smartphones run two different, highly vendor-customized versions of the AOSP. Further, we set up unmodified Android emulators running AOSP versions 4.3, 4.4.2, and 5.0. These are the most recent major versions of Android and represent nearly half of all the Android market-share [28]. In total, this allows us to stage "crimes" involving 5 vastly different Android devices to evaluate VCR's effectiveness and generality.

We first installed each app on our test devices and interacted with its camera features (i.e., taking photos, videos, and simply watching the preview screen). We then closed the app and used `gdb` to capture a memory snapshot from the mediaserver process. To attain ground truth, we manually instrumented the mediaserver to log allocations and deallocations of data structures containing photographic evidence. This log was later processed to measure false positives (FP) and false negatives (FN).

We used VCR to analyze the previously captured memory images and recorded the output photographic evidence. Despite the variety of different and customized AOSP versions tested, all evaluation was conducted using VCR with *the same set of Vendor-Generic Signatures* (generated from Google's AOSP 4.4.2 repository) which VCR automatically adapted to each input memory image. In a real-world law enforcement scenario, in-the-field investigators obtain images of a device's volatile RAM and non-volatile storage, and the collected memory images are later analyzed using VCR by forensic lab staff. Also note that VCR is a lightweight, efficient tool and could even be operated at the scene of a crime from an investigator's laptop. In all of our tests, VCR produced fully-rendered results from an input memory image in under 5 minutes (except for two specially noted cases at the end of this section).

3.3.1 App-Agnostic Evidence Recovery

This section presents the results of applying VCR to memory images containing photographic evidence generated by the following seven apps on our two smartphone devices. Five of the apps have features for taking individual photographs, videos, and displaying preview frames: the two smartphones' pre-installed camera apps, Google's Google Camera app, the Facebook app, and Instagram app. Each of these apps accesses and uses the camera device in different and forensically interesting ways. We also investigated evidence from the Skype app, which employs only video capture and preview functionalities. We also analyzed the Chase Bank app's check image

Table 3.1.: VCR recovery from apps on commodity Android smartphones.

Device	App	Evidence	Live Instances	w/ Image Data	Recovered	FP	FN
LG G3	Instagram	Preview	32	11	11	0	0
		Photo	1	1	1	0	0
		Video	20	20	20	0	0
	Facebook	Preview	32	11	11	0	0
		Photo	1	1	1	0	0
		Video	20	20	20	0	0
	Chase Banking	Preview	32	2	2	0	0
		Photo	1	1	1	0	0
	Skype	Preview	32	9	9	0	0
		Video	9	9	9	0	0
	LG Default Camera	Preview	32	10	10	0	0
		Photo	1	1	1	0	0
		Video	20	20	20	0	0
	Google Camera	Preview	32	11	11	0	0
		Photo	1	1	1	0	0
		Video	20	20	20	0	0
Samsung Galaxy S4	Instagram	Preview	32	7	7	0	0
		Photo	1	1	1	0	0
		Video	16	8	8	0	0
	Facebook	Preview	32	7	7	0	0
		Photo	1	1	1	0	0
		Video	16	8	8	0	0
	Chase Banking	Preview	32	8	8	0	0
		Photo	1	1	1	0	0
	Skype	Preview	32	7	7	0	0
		Video	9	8	8	0	0
	S4 Default Camera	Preview	32	7	7	0	0
		Photo	1	1	1	0	0
		Video	16	8	8	0	0
	Google Camera	Preview	32	7	7	0	0
		Photo	1	1	1	0	0
		Video	16	8	8	0	0

and upload feature. In the next sections we will highlight some of these apps as case studies.

Table 3.1 shows a summary of our evaluation results. Column 1 shows the device on which the evaluation was performed. Columns 2 and 3 show the app's name and which types of photographic evidence it can generate, respectively. The number of "live" frames (i.e., frames which were allocated and not yet freed) in the memory image is shown in Column 4. Column 5 shows the subset of those image frames which the camera HAL had filled when the memory image was captured[3]. Column 6 shows the number of images (i.e., photograph, video frames, or preview frames) which VCR recovered and rendered. Columns 7 and 8 show false positives (image frames which VCR wrongly reported) and false negatives (image frames which VCR missed).

From Table 3.1, we can make a number of key observations. First, VCR is highly effective at recovering and rendering photographic evidence left behind by a variety of Android apps. This confirms that VCR's Vendor-Generic Signatures ensure that the recovery mechanism is highly accurate. Table 3.1 shows that these constraints are indeed strong enough to effectively prune all invalid data and attest to the accuracy of any recovered evidence — resulting in VCR producing no false positive or false negative results. In total, VCR recovered 245 pieces of photographic evidence in these test cases.

Table 3.1 shows that of the 32 total test cases, all 12 cases left behind several preview frames. These results range from a high of 11 preview frames in the LG G3's Google Camera, Facebook, and Instagram cases to only 2 frames in the LG G3's Chase Bank test case. Interestingly, the average "preview frames recovered per app" appears to be phone dependent: 7.17 for the Samsung Galaxy S4 and 9 for the LG G3 (or even 10.4 if we ignore the outlier: the Chase Bank app). This implies some connection between phone hardware or vendor customizations versus the amount of

[3]The information in Columns 4, 5, and 6 was obtained via manual instrumentation only for the purpose of evaluation. VCR does not have access to such runtime information and operates on only the input static memory image.

Figure 3.7.: Sample video frames recovered from the Skype case study. This is an example of how multiple recovered frames can capture evidence of time and direction for the suspect shown here.

potential evidence. Since both phones have relatively equally powerful hardware, we reason that the latter is more influential. Again, these preview frames are generated by the apps *automatically* when the user *only opens the app's photographic features.*

Also shown in Table 3.1 is that video frames are far more prevalent than any other form of photographic evidence. This is intuitive given that video frames are often sampled at higher rates than preview frames. Our evaluation shows that on average each app left 12.9 video frames. Again, the LG G3 provides more evidence with an average of 17.8 video frames per app versus the Samsung at 8 video frames on average. Intuitively, Skype leaves fewer frames than the other apps in our tests (9 frames on the LG G3 and 8 on the Samsung Galaxy S4) likely because of the high throughput design of Skype's video-call feature. Also note that the recovered video frames are the result of explicitly recording video with the tested apps, unlike the preview frames which are generated without any explicit user command to record.

Finally, Table 3.1 shows that only one photograph per application is available in the memory images. Manual investigation revealed that the Android framework prefers to reuse buffers as quickly as possible, so despite taking several photos during our testing only a single photograph is left buffered — always accompanied by a number of preview frames.

Case Study 1: Camera Apps

Camera apps are standard Android apps which only provide a front-end user interface to the camera back-end (the mediaserver and camera HAL). A newly purchased Android device will come with a pre-installed camera app, but the user may install a new camera app and select one to use as the default. To illustrate the generality of VCR, we evaluate both pre-installed camera apps from our test phones as well as the third-party Google Camera app. The results of the LG G3 Default Camera and Google Camera tests are shown in Rows 5 and 6 of Table 3.1, and the Samsung Galaxy S4 Default Camera and Google Camera tests are shown in Rows 11 and 12.

Table 3.1 shows that in each of the camera app tests VCR is able to accurately recover and render all photographic evidence. For the LG G3 Default Camera case, we see that VCR recovered 10 preview frames, 1 photo, and 20 video frames, and similarly for the Google Camera test VCR recovered 11 preview frames, 1 photo, and 20 video frames. Again we observe fewer recoverable frames in the Samsung cases: 7 preview frames, 1 photo, and 8 video frames for both the Default Camera and Google Camera evaluations.

The default camera app is important because other apps may rely on it for photographic operations. When choosing test cases, we intentionally included the Facebook app as an example of this (shown in Rows 2 and 8 of Table 3.1). The Facebook app allows users to capture and post videos and photos on-the-fly (i.e., without leaving the Facebook app). To implement this, the Facebook app requests the default camera app to take a photo or video and then return the resulting image. Thus when the Facebook app user requests to capture a photo or video, the default camera app opens, manages the image capture, and makes the resulting image available to the Facebook app.

The fact that the Facebook app (and others like it) employ the default camera to handle photography, leads to a forensically interesting observation: photographic evidence from such apps will likely use similar formatting and sizing parameters to

conform with the default camera pass-through interface. In the Facebook app case studies from Table 3.1, we see that VCR is able to render 11 preview and 20 video frames plus 1 photograph for the LG G3 test and 7 preview and 8 video frames plus 1 photograph for the Samsung S4 case.

It is important to note that among all of our test cases only the Facebook app is an example of requesting photography through the default camera. Although default camera pass-through is common, we intentionally focused the majority of our evaluation on test cases which implement their own photography features. This directly shows VCR's generality with regards to the evidentiary apps' implementation.

Case Study 2: Skype

In this case study, we highlight the Skype app because image frames collected by Skype are *never present on non-volatile stores* — Skype immediately encodes, packages, and transmits the image frames over the internet. Thus the only visual artifacts of a Skype video-call will be the frames left in the device's memory. Such frames provide vital evidence in a digital investigation — as we will show with a scenario based on the Usenix Security 2014 invited talk "Battling Human Trafficking with Big Data." [26]

Imagine, for the sake of example, that a human-trafficking suspect is using Skype video calls from a smartphone to show victims to potential clients. While the criminal may be careful not to show his or her identity, the video frames of the Skype call clearly link the smartphone user to the victims of the crime. Further, this criminal may try deleting (or obfuscating) Skype's call history, but even after the criminal has ended the Skype calls and finished trying to hide the evidence, the last snippets of video are still recoverable in the device's memory. Later, when law enforcement agents arrest the suspect, investigators will not find any evidence on the smartphone's non-volatile storage. Applying VCR to the smartphone's memory will reveal the last video frames

of the Skype call showing one or more victims of this crime, and providing vital evidence to investigators which would otherwise be inaccessible.

For this case study, we set up a simplified crime reenactment by having one of the authors walk slowly through a Skype video call's field of view. We then used VCR to recover the remaining video frames frozen in the device's memory image. The LG G3 device was used in this trial, and the results of analyzing the device's memory image are shown in Row 4 of Table 3.1.

Figure 3.7 shows some of the recovered video frames and gives a clear example of the importance of VCR-recovered photographic evidence to an investigation. The 6 frames shown in Figure 3.7 are a subset of the 9 video frames in total which VCR recovered. These frames reveal a person walking through the Skype call's field of view, and we can easily see how this provides substantial evidence to investigators about the human-trafficking victims in our crime scenario above.

In addition to recovering the video frames shown in Figure 3.7, VCR also recovered 9 preview frames still buffered in the memory image (as shown in Table 3.1). Visual inspection of all 18 images recovered for this test case revealed that 4 of the 9 preview frames were identical (to the investigator's eye) to 4 of the recovered video frames. Thus yielding 14 total unique images to be used as evidence. This case study shows the importance of VCR recovered photographic evidence to aiding a digital investigation.

3.3.2 Analysis Across Android Frameworks

Given that many versions of the AOSP are being widely used today [28], VCR must be effective for a majority of devices that investigators may face. In this section, we evaluate VCR's effectiveness against memory images taken from the three most recent, widely used versions of the AOSP.

To perform this evaluation, we set up unmodified Android emulators running AOSP versions 4.3, 4.4.2, and 5.0. As before, we used VCR to analyze memory images after interacting with each of the tested applications. For this evaluation, we

Table 3.2.: VCR recovery from current and future Android versions.

Device	App	Evidence	Live Instances	w/ Image Data	Recovered	FP	FN
Android 4.3	Facebook	Preview	32	3	3	0	0
		Photo	1	1	1	0	0
		Video	31	31	31	0	0
	Skype	Preview	32	3	3	0	0
		Video	1	1	1	0	0
	Default Camera	Preview	32	5	5	0	0
		Photo	1	1	1	0	0
		Video	202	202	202	0	0
Android 4.4.2	Facebook	Preview	32	3	3	0	0
		Photo	1	1	1	0	0
		Video	16	16	16	0	0
	Skype	Preview	32	3	3	0	0
		Video	1	1	1	0	0
	Default Camera	Preview	32	3	3	0	0
		Photo	1	1	1	0	0
		Video	24	24	24	0	0
Android 5.0	Facebook	Preview	32	3	3	0	0
		Photo	1	1	1	0	0
		Video	19	19	19	0	0
	Skype	Preview	32	3	3	0	0
		Video	1	1	1	0	0
	Default Camera	Preview	32	3	3	0	0
		Photo	1	1	1	0	0
		Video	297	297	297	0	0

selected three of the apps to use in each of the three emulators: Facebook, Skype, and each emulator's default camera app.

Table 3.2 presents the results which VCR rendered from the different emulators' memory images. Column 1 shows the version of Android that the emulator is running. Columns 2 and 3 show the app and types of photographic evidence evaluated respectively. Like in Section 3.3.1, the number of "live" image frames in each memory image is show in Column 4, and Column 5 shows the subset of these which contained image data. Column 6 shows the number of images which VCR recovered and rendered. Finally, Columns 7 and 8 report the false positives and false negatives.

Table 3.2 shows that VCR is highly effective at recovering and rendering photographic evidence produced on the most widely used Android versions. We observe that the emulated camera device used in the Android emulator does not produce frames at a high rate similar to our test smartphone devices. This leads to (as shown in Table 3.2) fewer frames being available in the memory images. On average, the tests in Table 3.2 produce only 5.8 frames (with the exception of the outliers: the 4.3 and 5.0 emulators' default cameras).

Additionally, the preview frame buffer is rarely filled above 3 frames. This results in VCR recovering only those 3 preview frames for all three apps on all three emulators, except for the Android 4.3 Default Camera test in which VCR recovered all 5 preview frames. Again, VCR is able to recover and render all instances of photographic data in the evaluated memory images without any false positive or false negative results — as Table 3.2 shows, 627 pieces of photographic evidence in total for these test cases.

Notably, Table 3.2 contains two exceptional cases. The default cameras for Android 3.4 and Android 5.0 report very large numbers of video frames. We performed manual inspection of the results and found that all output images were valid (i.e., from distinct buffers filled individually by the camera HAL). Further investigation revealed that there existed a bottleneck when saving those video frames to the emulator's storage. Admittedly, this is likely an emulator configuration error, but the resulting backup of frames further demonstrates the effectiveness of VCR's recovery and rendering — though run-times for these two cases were nearly 30 minutes.

3.3.3 Recovering Temporal Evidence

As shown in Table 3.1, numerous preview frames and/or video frames can be recovered for a single app — representing a *time-lapse* of what the camera was viewing. Here, we analyze how a set of preview or video frames can give investigators temporal evidence of the incident under investigation.

To measure the time captured by a set of recovered frames, we reran the two camera app test cases on the two smartphones. Time lapses were measured using the camera apps to record video of a stopwatch for a period of 1 minute, and the phones were rebooted between each test. Note that the "stopwatch" used here was actually a stopwatch app on the first author's smartphone. While this measurement may seem "low-tech," our results in Table 3.3 show that the time-lapse captured by the recovered sets of frames is long enough to make an empirical measurement very accurate.

After recording for 30 seconds, we captured a memory image from the device, and VCR was used to recover all available preview and video frames from the memory images. The output image frames were grouped into three sets: Preview frames, Video frames, and a Union set containing all visually unique frames from both the preview and video sets (which would be recoverable for any app which captures video). We then manually measured the difference between the earliest frame and the latest frame in each set. Figure 3.8 shows an example of some recovered stopwatch preview frames.

Table 3.3 presents the time measurements captured within the sets of recovered preview and video frames. Columns 1 and 2 show the tested device and app. Column 3 names the type of set being measured: Preview, Video, or the Union set. Column 4 shows the number of frames in the set, and the measured time difference is shown in Column 5.

From the times in Table 3.3 we can make several observations: First, the windows of time captured by the recovered frames are large enough to provide substantial evi-

Table 3.3.: Time-lapse evaluation.

Device	App	Evidence	Frames	Time-Lapse
LG G3	LG Default Camera	Preview	11	1.3s
		Video	20	0.6s
		Union	22	1.4s
	Google Camera	Preview	11	0.9s
		Video	20	0.4s
		Union	25	0.9s
Samsung Galaxy S4	S4 Default Camera	Preview	7	0.5s
		Video	8	0.3s
		Union	10	0.5s
	Google Camera	Preview	7	0.4s
		Video	8	0.3s
		Union	11	0.5s

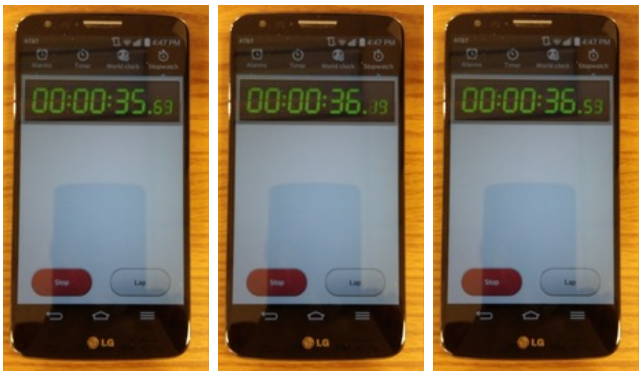


Figure 3.8.: Recovered preview frames used to measure temporal evidence. For this experiment, we recorded another smartphone's stopwatch app and used VCR to recover the preview and video frames — yielding empirical measurements of the temporal evidence captured in VCR recovered evidence.

dence to an investigation — we already empirically saw this in the evidence recovered for the "crimes" in Figures 3.1 and 3.7. To best analyze the results show in Table

3.3, consider the first row as meaning: The set of preview frames from the LG G3's Default Camera captures 1.3 seconds of time divided over 11 images. From this, we see that a majority of the results yield over a half second of time-lapse.

This may seem like a small amount of time, but considering how quickly many crimes can occur and how powerful this evidence can be (such as an image of a car involved in a shooting or a human-trafficking victim) this provides a significant amount of evidence to investigators. Specifically, the example sequences of images shown in Figures 3.1 and 3.7 both represent a time-lapse of less than 1 second. Table 3.3 shows that of the 12 measurements, the LG G3 provides much longer time windows with the average being 0.92 seconds per test. The Samsung provides an average of 0.42 seconds per test.

A second observation we make from Table 3.3 is that preview frames capture longer time windows in fewer frames but video frames provide many more images. Preview frame sets on the LG G3 average more than double the time window of video frame sets (i.e., 0.4 seconds versus 0.9 seconds and 0.6 seconds versus 1.3 seconds). However, the video frame sets in the LG G3 test contain 20 frames compared to only 11 frames in the preview sets. The Samsung results show a similar pattern but the differences between sets are much closer (e.g., 8 frames over 0.3 seconds versus 7 frames over 0.5 seconds). As a consequence, we can observe that the time delta between images is much shorter between video frames than between preview frames.

Finally, Table 3.3 shows that when video and preview frames are available then (not surprisingly) considering the union of those sets yields the best results. In practice, nearly any app which generates video frames will also generate preview frames. Table 3.3 shows that the video frames will mostly be enclosed by the larger time delta captured by the preview frames. For example, consider the LG G3's Google Camera Union test: the investigator can now see 0.9 seconds of time captured in 25 images — leading to roughly a 0.036 second time delta between each image. Using such analysis, investigators can gain a wealth of evidence from only the frames being recovered by VCR.

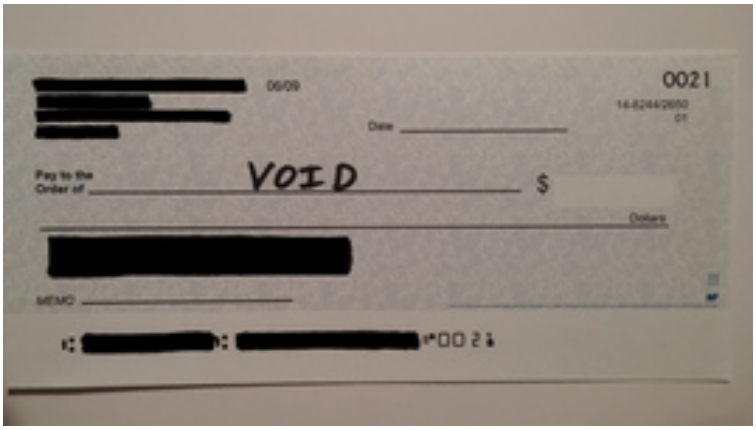


Figure 3.9.: Recovered check image left behind in a memory image. This case study gives an example of the potentially sensitive user information which VCR (or worse, malware) can *generically* recover from the mediaserver's memory.

3.3.4 Privacy Concerns

Finally, this section highlights a potential privacy concern which VCR reveals. VCR exploits the centralized design of the Android framework to access app-agnostic photographic evidence. It should be noted however that the same properties which make the mediaserver beneficial for digital forensics also make it a target for attack.

There has been extensive prior work on exploiting vulnerabilities in the Android framework to glean information about a smartphone's owner [29, 30]. Following that line of work, we can envision a malware which aims to steal confidential information and remain as stealthy as possible. Unfortunately, the mediaserver is a great target for such malware for a few reasons: 1) As we will show, the mediaserver handles very sensitive data regarding the device's owner, 2) As we have shown, it is beneficial to utilize the mediaserver's centralized design to capture photographic evidence from all apps generically, and 3) The mediaserver runs as a background service in a dedicated process (which makes for a great hiding spot for malware).

To underscore the potential danger of malware gaining access to a device's intermediate service processes (like the mediaserver), we have included the Chase Bank

app in our previous evaluations. The Chase Bank app, like many other financial institutions' apps, includes a check image and upload feature. When the device's owner has a check to deposit, they simply take a picture of the check and upload the image to Chase from within the app. The image is never saved to non-volatile storage and handled securely once the Chase app has received the image. Unfortunately, the image is buffered in the mediaserver long before it is returned to the Chase app and may remain buffered for long after.

To highlight this point, Figure 3.9 shows one of the check images that VCR recovered during our previous evaluations. Further, Table 3.1 shows that VCR was effective at recovering and rendering all forms of photographic evidence from the Chase Bank app test cases (12 images in total). The real danger here is that by employing the same techniques as VCR, malware can also have access to *any image taken by any app on the smartphone* — in the same way that VCR operates independent of which app generated the photograph. Moreover, if malware has access to the image buffers in the mediaserver at the right time, it may even *alter the check image before the Chase app receives it*. In light of this, we hope to emphasize the importance of Android's intermediate service processes as a security critical component and the need for security mechanisms to prevent malware from tampering with these services.

4 GUITAR: GUI TREE ARCHAEOLOGY

After VCR, I identified another paradigm-shifting opportunity in memory forensics: Moving away from individual pieces of evidence (e.g., a PDF recovered by DSCRETE or video recovered by VCR) toward evidence which holistically reveals how a suspect *used their device in the commission of a crime.* In this regard, possibly the most probative form of in-memory evidence is an application's graphical user interface (GUI). The 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 [31–39], 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 [24, 25, 40–42]. Moreover, if the app requires a password login every time it is brought to the foreground (see the case study in Section 4.3.2), 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 [6, 7], 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 [8–11, 14, 17, 18] focus on the recovery of *individual data structures*. Given a memory image and a data structure of interest, existing techniques (e.g., [8–11,17,18, 43]) 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 perceivable format, DSCRETE 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 4.1, 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) [44]. To address this challenge, I will 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."

4.1 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 4.2. 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 4.1 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 4.1), 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.

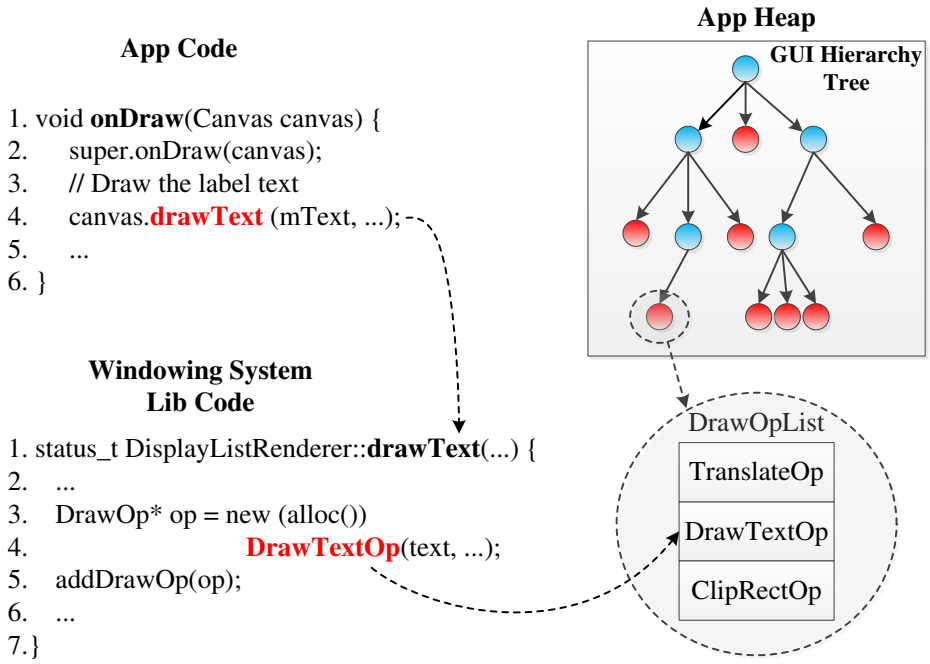
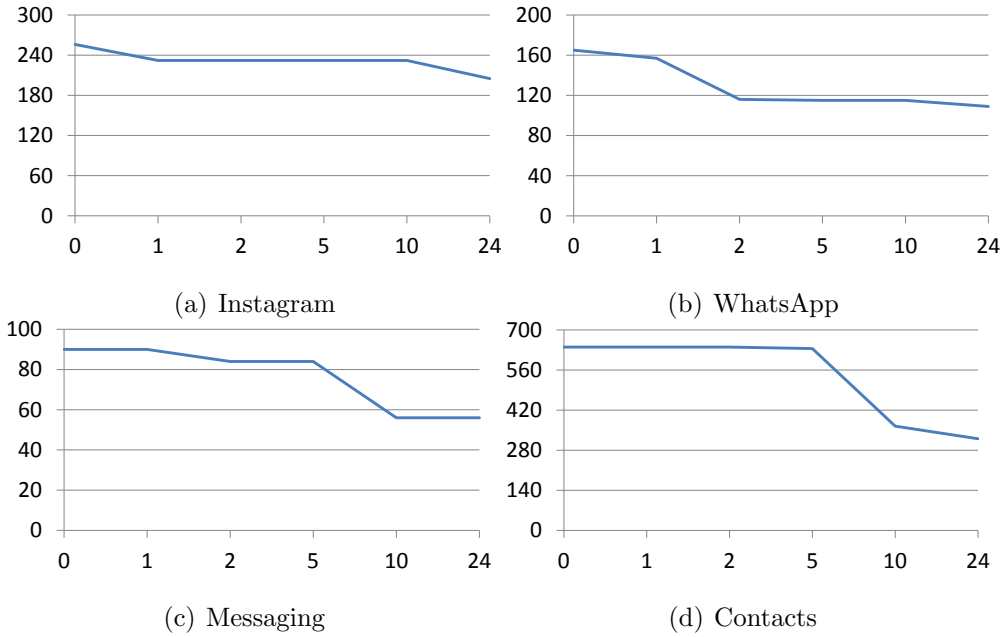


Figure 4.1.: Overview of a windowing system library. Each app maintains a GUI tree, with each leaf node containing Drawing Operations. Changes to the app's GUI are reflected by changes to the GUI tree.

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 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 4.2 shows measurements of 4 apps' GUI tree data structures after those apps



(a) Instagram (b) WhatsApp

(c) Messaging (d) Contacts

Figure 4.2.: Number of recoverable GUI data structures of backgrounded apps over 24 hours.

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 4.2 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 4.3 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 4.3 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.

4.1.1 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 4.2.1) 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 remapping 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 4.2.2) 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 4.2.3) 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.

4.2 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. 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 4.2.4 for more information about GUITAR's data structure signatures.

4.2.1 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 4.3 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 4.3).

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

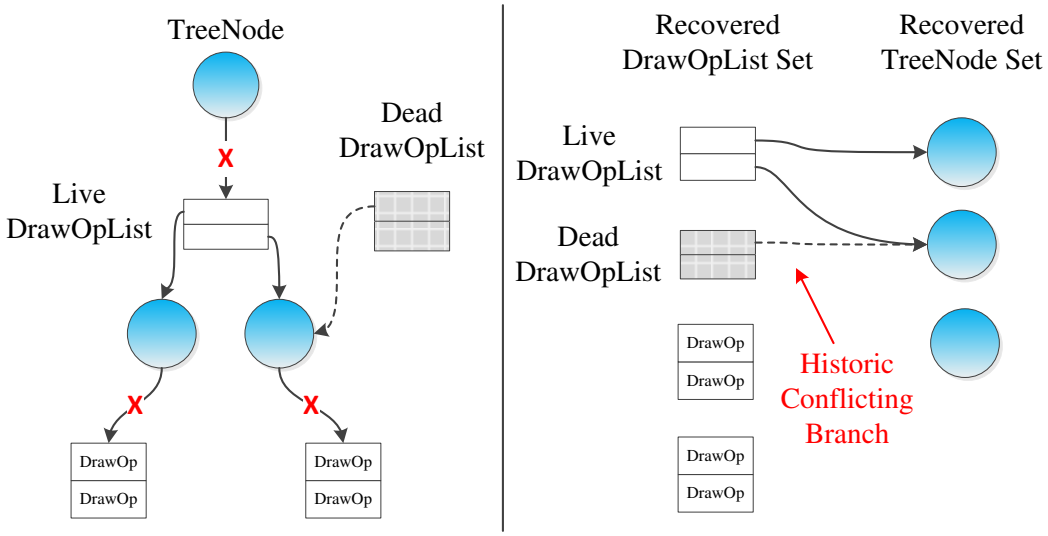


Figure 4.3.: Broken tree structure due to nullified pointers (marked with a red X). GUITAR must rebuild the tree from the recovered TreeNodes and DrawOpLists. Note the conflicting branch introduced by historic data structures.

the "DrawOpList to child" pointers (shown in Figure 4.3) to recover the "parent to DrawOpList" pointers.

However, an issue arises during topology recovery: *conflicting branches* may be introduced to the reconstructed 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 4.3 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 4.3. 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 4.2.2). Our evaluation in Section 4.3 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 2) 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 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

Algorithm 2 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)
```

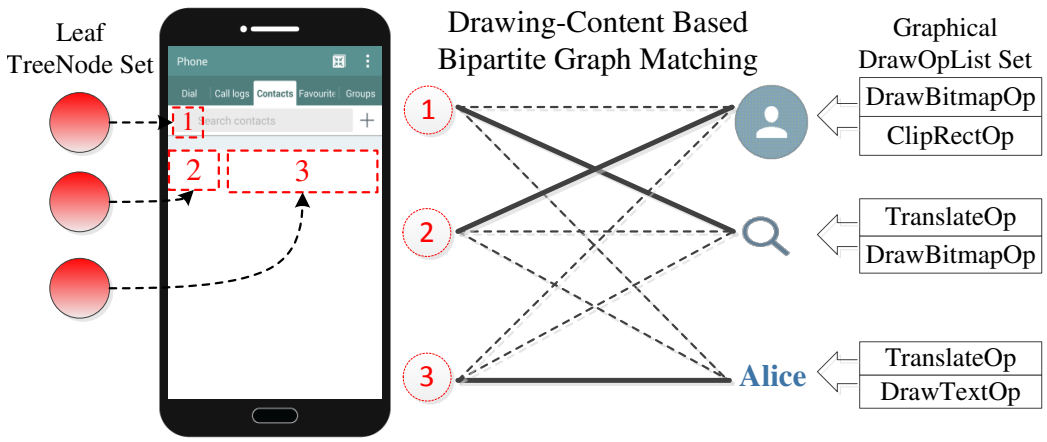


Figure 4.4.: Example of drawing-content based bipartite graph matching.

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.

4.2.2 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 4.1), unlike those pointed to by non-leaf nodes. Figure 4.4 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 3 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 3. 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 3.

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 3), 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 3, 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

Algorithm 3 Building Draw-Content-Weighted Bi-graph

Input: Leaf Set L, DrawOpList Set D, ScalingFactor f

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

$V_{leaves} \leftarrow \varnothing$
$V_{opLists} \leftarrow \varnothing$
$E \leftarrow \varnothing$
$maxWeight \leftarrow 0$
for $ops \in D$ **do** ▷ Compute DrawOpList dimensions
  $(ops_{width}, ops_{height}) \leftarrow computeDrawSize(ops)$
  $ops.width \leftarrow ops_{width}$
  $ops.height \leftarrow ops_{height}$
  $V_{opLists} \leftarrow V_{opLists} \cup ops$ ▷ Insert DrawOpList vertex

for $leaf \in L$ **do** ▷ Compute leaf dimensions
  $right, below \leftarrow getNeighbors(leaf)$
  $(x_{right}, y_{right}) \leftarrow getFullCoord(right)$
  $(x_{below}, y_{below}) \leftarrow getFullCoord(below)$
  $(x_{leaf}, y_{leaf}) \leftarrow getFullCoord(leaf)$
  $(leaf_{width}, leaf_{height}) = (x_{right} - x_{leaf}, y_{below} - y_{leaf})$
  $leaf.width \leftarrow leaf_{width}$
  $leaf.height \leftarrow leaf_{height}$
  $V_{leaves} \leftarrow V_{leaves} \cup leaf$ ▷ Insert leaf vertex

for $ops \in V_{opLists}$ **do**
  for $leaf \in V_{leaves}$ **do** ▷ Compute edge weights
    $d_{width} \leftarrow leaf.width - ops.width$
    $d_{height} \leftarrow leaf.height - ops.height$
    if $d_{width} < 0$ **or** $d_{height} < 0$ **then**
      $scale \leftarrow f$ ▷ Scale factor for over-drawing
    else
      $scale \leftarrow 1.0$
    $weight \leftarrow scale * (\sqrt{(d_{width})^2 + (d_{height})^2})$
    $E(ops, leaf) \leftarrow weight$ ▷ Insert edge weight
    if $weight > maxWeight$ **then**
      $maxWeight \leftarrow weight$ ▷ Update max weight

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.

Algorithm 4 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)
```

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 4 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 3). 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 bipartite graph assignment solving algorithm is represented in Algorithm 4 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 $O(n^3)$ [45]. 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.

4.2.3 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

[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.

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.

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, be-

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

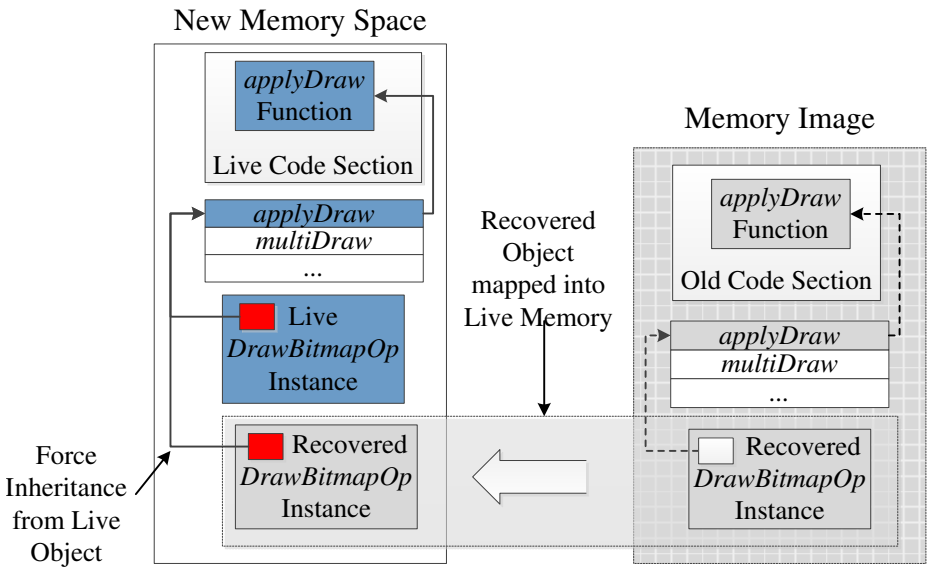


Figure 4.5.: Illustration of forced polymorphism.

cause the function's old location is abandoned and corrected to the live location. Figure 4.5 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.

4.2.4 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 4.6.: DrawTextOp class definition and resulting data structure signature.

GUITAR uses a combination of structural and value invariant signatures for each structure it recovers. Figure 4.6 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`).

Table 4.1.: Recovery of backgrounded GUI data structures.

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

4.3 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.

4.3.1 GUI Data Elements (Puzzle Pieces)

As stated in Section 4.1, 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,

[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.

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 4.1 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 4.1, 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 4.1 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).

Table 4.1 shows that a large percentage of these data structures persist after the application is backgrounded. As presented in Section 4.1, this percentage will drop over time if the app remains in the background — Section 4.3.3 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.

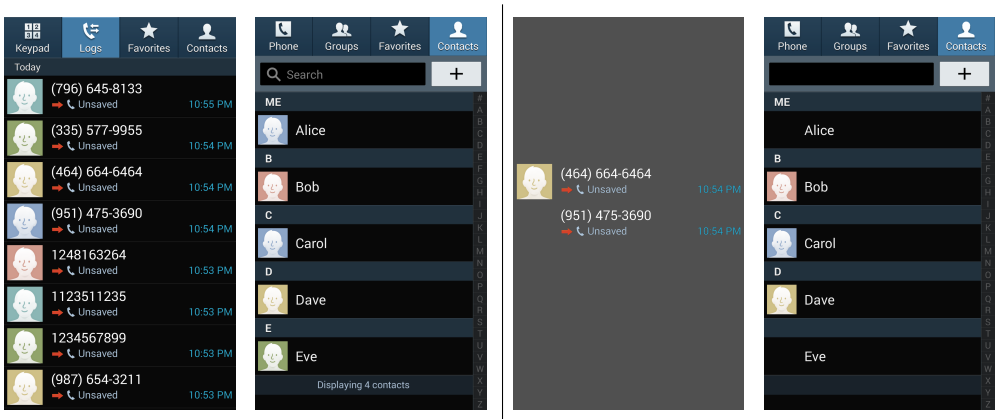
Lastly, these results show that GUITAR's memory image scanner is robust enough to recover 100% of the data structure instances in the backgrounded apps' memory images. Although we point out that individual data structure recovery is *not* GUITAR's primary capability and may be performed by other existing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

4.3.2 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.

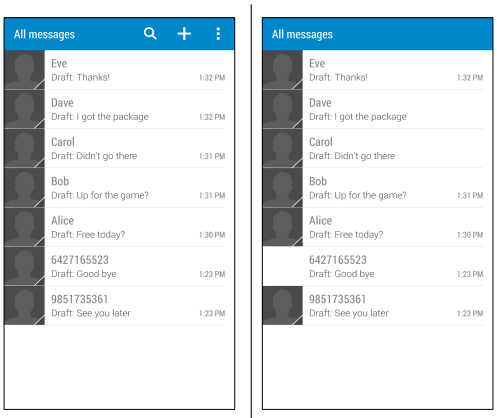
We first need to compare a GUITAR-reconstructed GUI tree with a *Ground Truth Tree* (i.e., the app's true GUI tree as it was in the memory image). To obtain the Ground Truth Tree, we instrumented each app to log the structure and content of its GUI tree in the foreground. From that log, we subtracted 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 [46, 47] and the default CEDD indexing feature [48]. Notice that this comparison is actually *unfavorable*



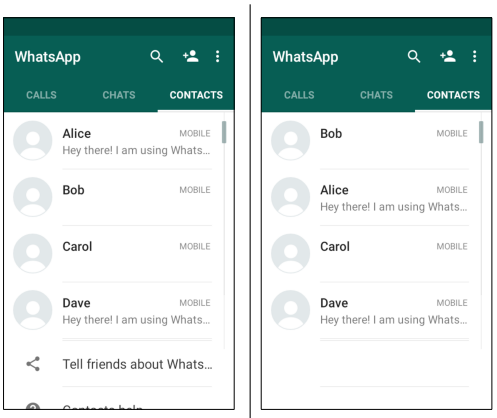
(a) Earlier Screen. (b) Latest Screen. (c) Conflict Branch. (d) Recovered GUI.

Figure 4.7.: Samsung Contacts app with redrawn full conflict branch.



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

Figure 4.8.: HTC Messaging.



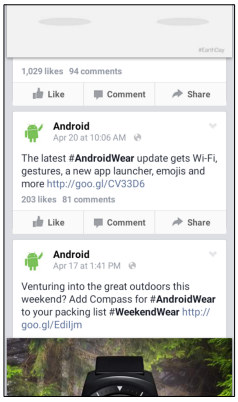
(a) App Screen. (b) Recovered GUI.

Figure 4.9.: LG WhatsApp Contacts.

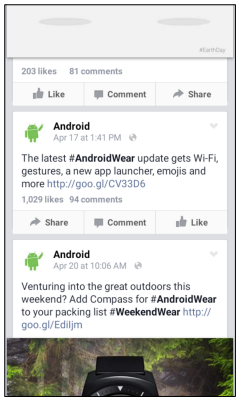
to GUITAR because the screenshot is taken when the app is in the foreground, and GUITAR has no control over what data is overwritten when the app is backgrounded.

Table 4.2 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

[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.

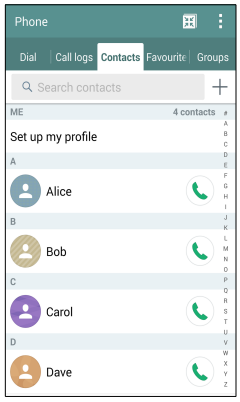


(a) App Screen.

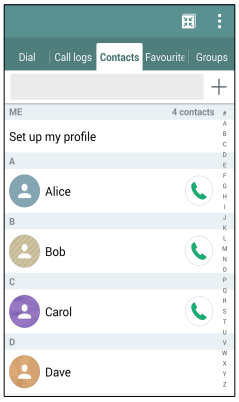


(b) Recovered GUI.

Figure 4.10.: Samsung Facebook.



(a) App Screen.



(b) Recovered GUI.

Figure 4.11.: LG Contacts app.

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 4.2 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 4.2, 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 4.2 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 that were present on an even earlier screen. Figure 4.7 shows the two previous GUI screens

which contributed to the resulting GUITAR-reconstructed GUIs: one from the most recently viewed screen (Figure 4.7(b)), and a portion of the earlier screen (a full conflicting branch, Figure 4.7(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 4.2.2, 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 4.8 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 4.2 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 4.10 shows the *best case*: Samsung S4's Facebook, and Figure 4.11 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% identi-

cal 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 4.10). 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 4.2. 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.

Case Study: WhatsApp on LG G3

In several test cases presented in Table 4.2 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 4.1, 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 4.2 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 4.9 presents the foreground screenshot of the app and the GUI reconstructed by GUITAR. 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.

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.

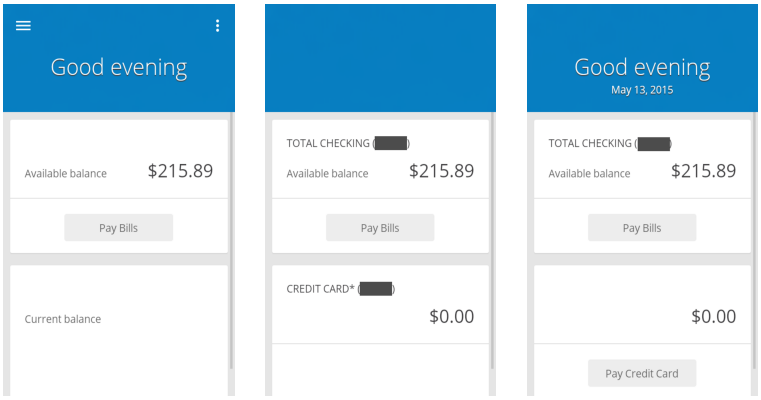
Importantly however, when such a secure app is being used, the last screen the user views before backgrounding the app is always some *internal* screen of the app after the user has already logged in. Further, this most recent internal screen will be the one present in the app's heap *even after the 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 4.2 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 4.2 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 4.12 shows the reconstructed app GUIs for all 3 devices (and one of the authors' graduate-student-size account balance).

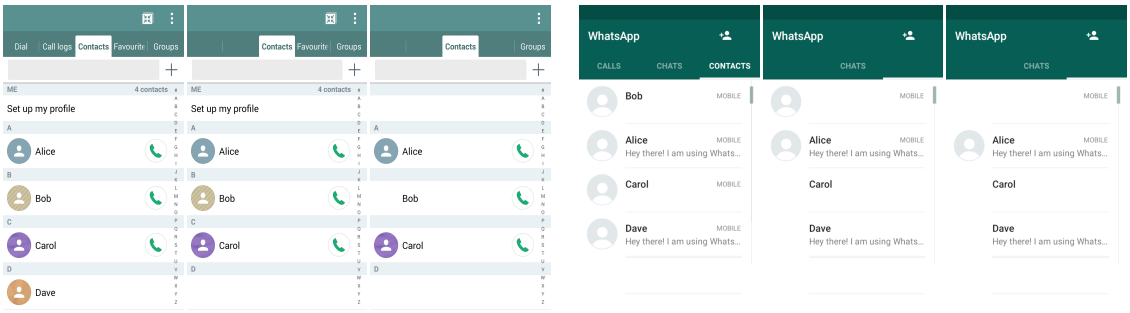
We point out that a broader impact of this case study is the user privacy concerns it raises for running highly sensitive apps on smartphones. Interestingly, even apps focusing on privacy (such as TextSecure [49]) cannot disrupt GUITAR's recovery. This is because GUITAR operates on the lowest-level GUI 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



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

Figure 4.12.: Reconstructed Chase Banking GUIs. Although the user was logged out, recovered GUIs still reveal sensitive information. Note: we manually blocked out the account number.

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

Figure 4.13.: Contacts and WhatsApp GUIs with varying degrees of loss over time.

4.3.3 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 4.1, 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 4.2, a non-trivial amount of the GUI data persist in the app's free heap space over a period of 24 hours.

Using the memory images of 4 apps taken in Section 4.3.1 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 4.3 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 4.3 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 4.3 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 4.3.1: 89.23%. Further, GUITAR reconstructs GUI trees that are all more than 71% identical to their Ground Truth Trees.

Notably, Table 4.3 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 4.3 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 4.13 shows the gradual (but graceful) degradation of GUI reconstructed by GUITAR 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 4.13) are still of forensic value.

Table 4.2.: Reconstruction of GUI trees of various apps from different phones.

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

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

5 RETROSCOPE: SCREEN AFTER PREVIOUS SCREEN

Shortly after GUITAR, I realized that any expansion beyond GUITAR's capability would be limited by how much GUI data remained in a smartphone's memory, which the previous chapter has shown to be roughly 80% of *only* a single screen per app. While this one screen may reveal some evidence to investigators, it can hardly portray a suspect's actions and motives (e.g., after a suspect logs out, GUITAR will only recover the "log (back) in" prompt). In this chapter, 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.

Recall that GUITAR 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, I will 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 by GUITAR, 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 approach 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 [8, 11, 17, 18] and rendering instances of those app data (as we have seen in DSCRETE, VCR, and GUITAR), 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 these observations, 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 5.2.2). 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.

5.1 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, GUITAR shows that every newly rendered app screen destroys and overwrites the GUI data structures of the previous screen.

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

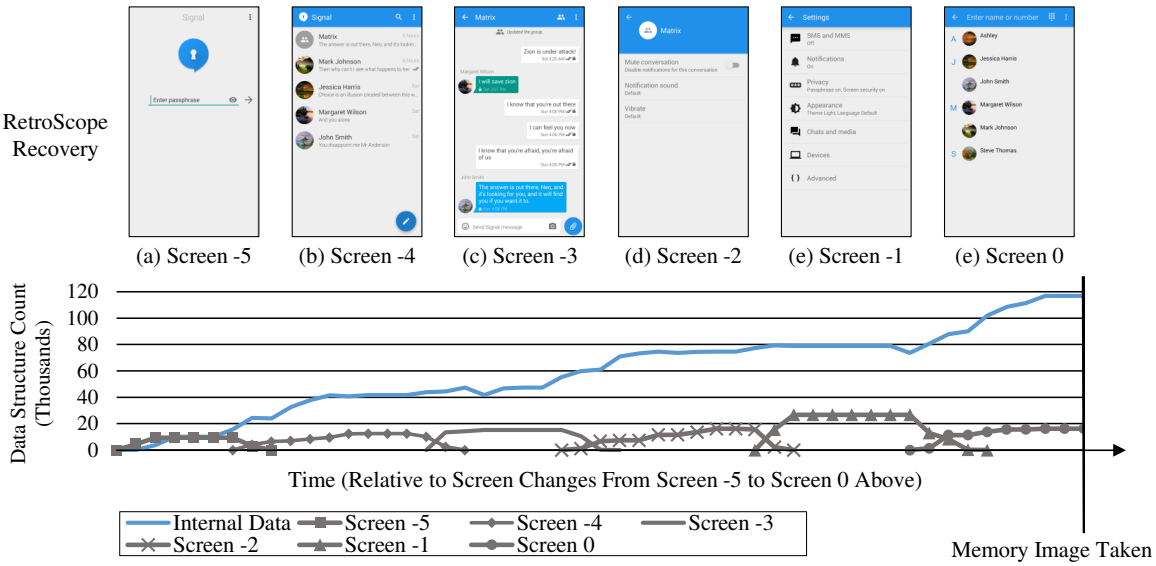


Figure 5.1.: Life cycles of GUI data structures versus app-internal data across multiple screen changes.

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 5.3). 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 5.1 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

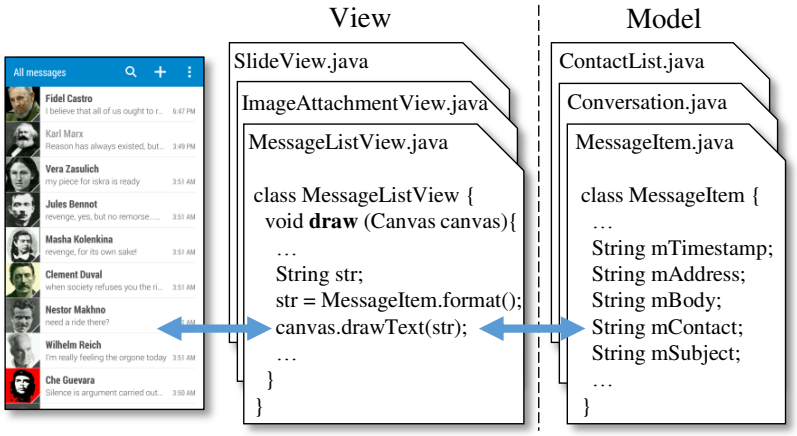


Figure 5.2.: The typical model/view implementation split of Android apps.

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 5.1 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 5.1), our GUITAR technique will only have access to the GUI data for Screen 0. Meanwhile, the app's internal 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 [8,50] to meaningfully recover and redisplay the app's internal data on Screens -1 to -5 in Figure 5.1.

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 5.2, 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 5.2 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 5.2'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.

5.2 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 5.1 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, e.g., DSCRETE and VCR.

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.

5.2.1 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 5) — 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 5.2.2) 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 5). 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 5.4). 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 op-

[2]Please see Section 5.4 regarding Dalvik JVM versus ART runtimes.

eration 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 5). 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 5.1, 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.

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 5). 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 5). 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 sub-Views. Intuitively, the most recently modified portion of the screen (sub-View) will yield an increasingly large ID.

5.2.2 Interleaved Re-Execution Engine

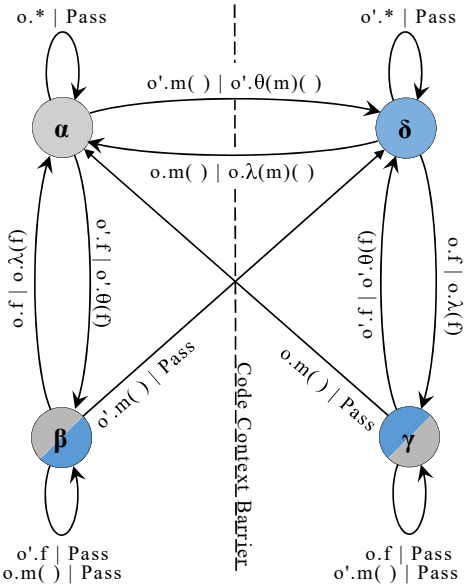


Figure 5.3.: State interleaving finite automata.

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 5.3), 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 5.4 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:

$$\begin{aligned} (\text{new code}, \text{new runtime}) &= \textcircled{\alpha} \\ (\text{new code}, \text{old runtime}) &= \textcircled{\beta} \\ (\text{old code}, \text{new runtime}) &= \textcircled{\gamma} \\ (\text{old code}, \text{old runtime}) &= \textcircled{\delta} \end{aligned} \tag{5.1}$$

In Figure 5.4, we have denoted which state the IRE is in before and after executing each line of code. For ease of explanation, Figure 5.4 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 γ).

Modeling State-Transitions. In Figure 5.3, 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 5.3. Note that our discussion will follow Java's object-oriented design, but the transitions in Figure 5.3 are equally applicable to static (i.e., o == NULL) execution.

The state transitions in Figure 5.3 are modeled as a Mealy machine [51] 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 α to δ is represented as:

$$\alpha \rightarrow \delta \; : \; o'.m() \mid o'.\theta(m)() \tag{5.2}$$

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 5.4 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 γ to α is represented as:

$$\gamma \rightarrow \alpha \; : \; o.m() \mid \text{Pass} \tag{5.3}$$

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 5.4. At the beginning of Line 11, the IRE is in state ⓨ 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 5.3 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. Interestingly, we find a number of test cases in Section 5.3 *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 5.3 does not illustrate state transitions for return instructions, the IRE does perform state correction for them. Unlike the transitions in Figure 5.3 (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 5.4. Before the invocation of `getClipBounds` (Line 11), the IRE is in state $\textcircled{\gamma}$. Function `getClipBounds` executes in state $\textcircled{\alpha}$, and upon its return the IRE pops state $\textcircled{\gamma}$ 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 $\textcircled{\alpha}$ or $\textcircled{\delta}$ because no data from the other environment enter those functions.

Native Execution. The IRE operates on the Java byte-code 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 5.3 (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.

5.2.3 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 (ⓐ or ⓓ) 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 ⓐ, 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.

5.3 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

[3]These devices all run vendor-customized versions of Android Kitkat (the most widely used Android version [28]).

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 [52]), Telegram (whose encrypted broadcast channels are popular with terrorist organizations [53]), 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 Tables 5.1, 5.2, and 5.3. These tables 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. Tables 5.1, 5.2, and 5.3 provide interesting insights into the complexity and scale of screen redrawing via selective reanimation. From these, we learn that an average of 231,867 byte-code instructions and 5,047 JNI

Table 5.1.: Samsung S4 results of RetroScope evaluation.

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

Table 5.2.: LG G3 results of RetroScope evaluation.

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)
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

Table 5.3.: HTC One results of RetroScope evaluation.

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)
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

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 that each screen is made of "thousands of GUI data structures." Most importantly, as also shown previously, 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.

5.3.1 Spatial-Temporal Evidence Recovery

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

[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.

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 5.6). Overall, Tables 5.1, 5.2, and 5.3 show 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.

Tables 5.1, 5.2, and 5.3 highlight 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 5.5 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 those tables 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, Tables 5.1, 5.2, and 5.3 show 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.

5.3.2 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 5.3 Row 2 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 5.6. 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.

5.3.3 Case Study II: Background Updates

Another interesting case is WhatsApp Messenger on the Samsung S4. Table 5.1 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 5.7 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 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 5.7(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.

5.3.4 Case Study III: Deleted Messages

In addition to the WhatsApp case above, RetroScope recovered *extra screens* for four other cases: Telegram (Table 5.1 Row 12), WeChat (Table 5.2 Row 14), WhatsApp (Table 5.2 Row 15), and TextSecure (Table 5.3 Row 13). 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 5.8 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 [52, 54]. Despite the few hardening measures discussed in Section 5.4, RetroScope can provide such alternative evidence which would otherwise be unavailable to investigators.

5.4 RetroScope and Privacy Implications

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 [40, 41], 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., [8, 17, 18]) 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 5.3 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 [28]) 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.

Algorithm 5 RetroScope Selective Reanimation.

Input: Target App Memory Image M

Output: GUI Screen Ordered Set S

▷ Rebuild the Target App runtime environment.

for Segment $S \in M$ **do** ▷ Remap memory segments.
    if isNeededForReanimation(S) **then**
        Map($S.startAddress$, $S.length$, $S.content$)
SymbiontApp.initialize() ▷ Set up Symbiont App.
JavaGlobalStructs $G \leftarrow \varnothing$ ▷ Isolate the Target App runtime state.
for Segment $S \in M$ **do** ▷ Find Java control data.
    if containsJavaGlobals(S) **then**
        $G \leftarrow$ getJavaGlobals(S)
        break

▷ Register reanimation points with the IRE.

InterleavedReexecutionEngine IRE
View Set $V \leftarrow \varnothing$ ▷ Top-level Views.
for Class $C \in G \rightsquigarrow Classes$ **do** ▷ Find top-level Views.
    if $C <: ViewParent$ **then** ▷ '$<:$' denotes subtype.
        if not isSubView(C) **then**
            IRE.beginOn($C.draw$) ▷ Register drawing function.
            View Set $views \leftarrow C.instances$
            $V \leftarrow V \cup views$
View $T \leftarrow$ *SymbiontApp*.getTopLevelView()
T.invalidate() ▷ Cause screen redraw command to be issued.
procedure CatchRedrawCommand

▷ Invoked when redraw command is issued.

    for View $view \in V$ **do**
        $T \leftarrow view$ ▷ Override the Symbiont App's top-level View.

▷ Record largest subView ID.

        $largestID \leftarrow \max_{v \in view.subViews} v$.getField($ID$)
        deliverRedrawCommand()

▷ IRE handles re-execution of redrawing code.

        Screen $s \leftarrow T$.copyGUIBuffer()
        S.insert($largestID$, s)
end procedure

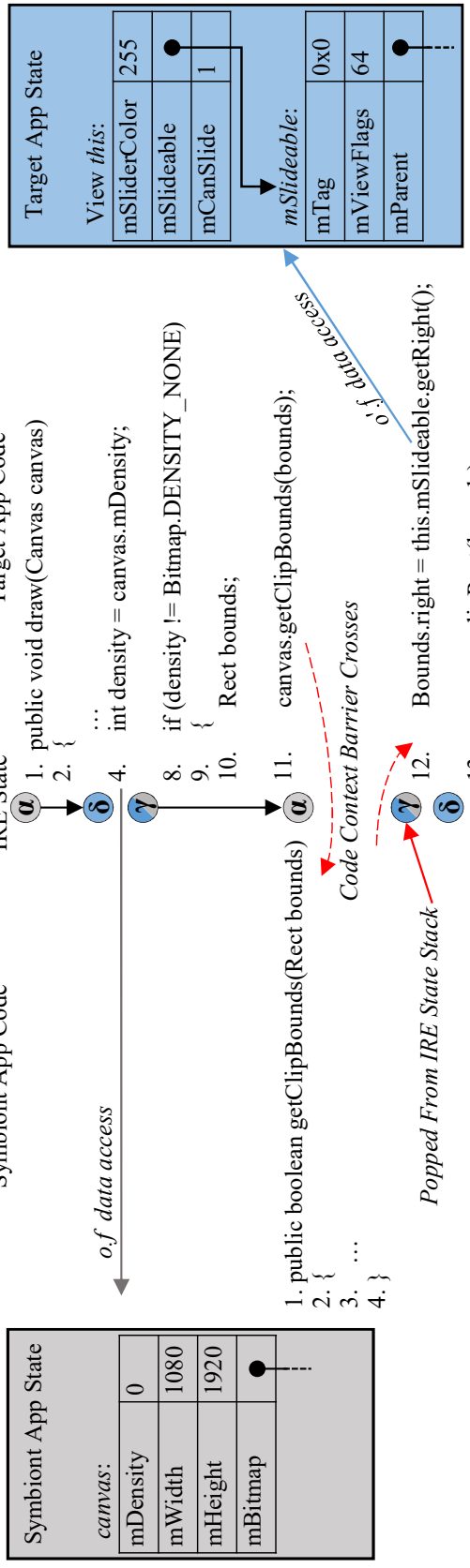
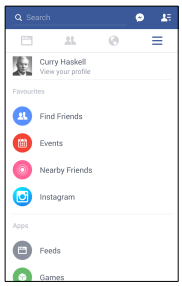
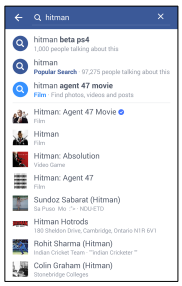


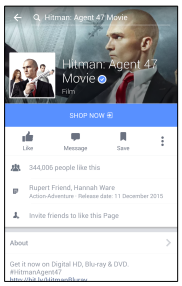
Figure 5.4.: Example of interleaved re-execution.



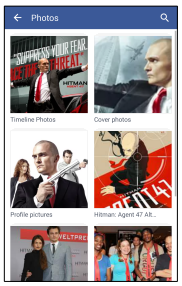
(a) Screen -6.



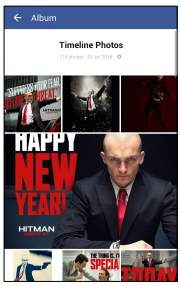
(b) Screen -5.



(c) Screen -4.



(d) Screen -3.

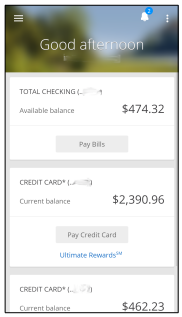


(e) Screen -2.

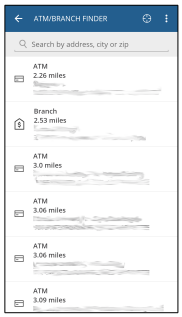
(f) Screen -1.

(g) Screen 0.

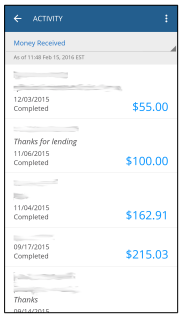
Figure 5.5.: LG G3 Facebook recovery.



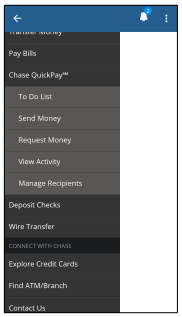
(a) Screen -10.



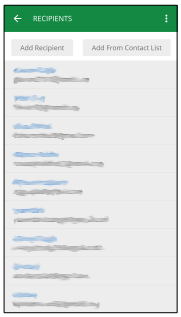
(b) Screen -9.



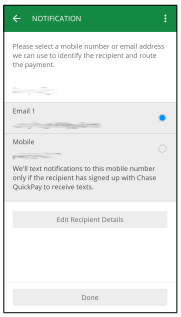
(c) Screen -8.



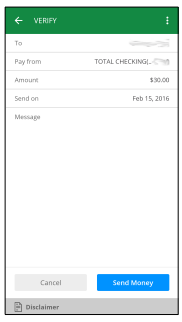
(d) Screen -7.



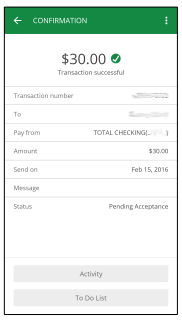
(e) Screen -6.



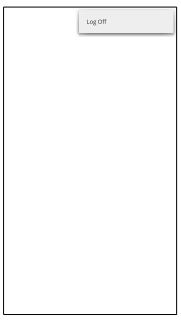
(f) Screen -5.



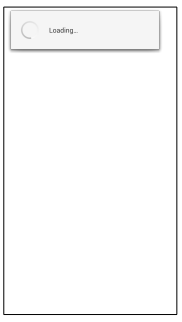
(g) Screen -4.



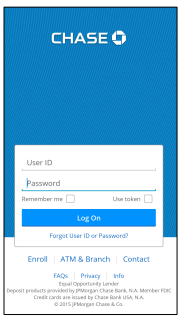
(h) Screen -3.



(i) Screen -2.

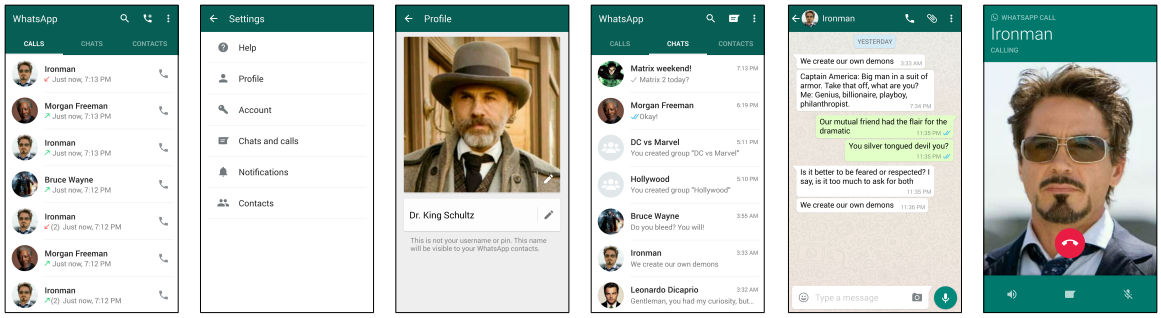


(j) Screen -1.

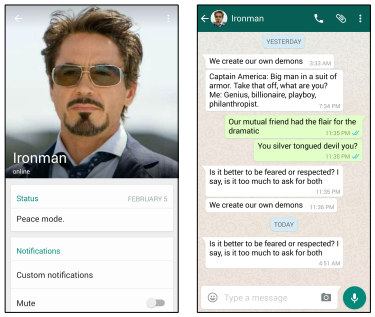


(k) Screen 0.

Figure 5.6.: HTC One Chase Banking recovery.

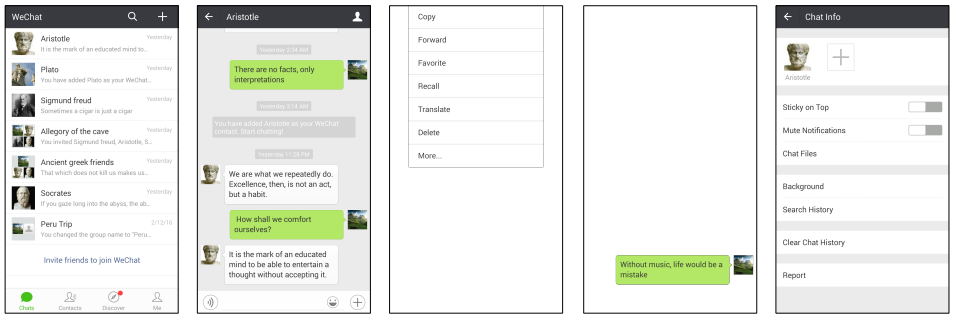


(a) Screen -6. (b) Screen -5. (c) Screen -4. (d) Screen -3. (e) Screen -2. (f) Screen -1.



(g) Screen 0. (h) Screen +1.

Figure 5.7.: Samsung S4 WhatsApp recovery.



(a) Screen -4. (b) Screen -3. (c) Screen -2. (d) Screen -1. (e) Screen 0.

Figure 5.8.: LG G3 WeChat recovery.

6 RELATED WORKS

Acquisition of Memory Images. 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 [55], LiME [56]) or hardware (e.g., Tibble [6], CoPilot [57]) tools. Other notable techniques have used the DMA-capable Firewire port [58] to acquire memory images, existing hibernation or swap files [10, 59–61], or cold/warm booted devices [62–64], but such approaches are only employed for highly specialized investigations. A more comprehensive list of memory image acquisition tools can be found in [65].

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 [56] and TrustDump [66] techniques. In an alternative approach, Hilgers et al. [62] proposed cold-booting Android phones to perform memory forensics. Our evaluation 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 [67, 68].

Memory Image Analysis. Prior to my work, researchers and investigators alike considered data-structure recovery the ultimate goal of memory image forensics. Earlier techniques for the analysis of memory images can be roughly divided into the following two categories based on the data structure signatures they employ:

1. Value-invariant signatures leverage known in-memory value patterns or invariants to locate data structure instances via brute-force scanning [8–13].

2. Structural-invariant (or "points-to") signatures rely on the interconnection of data structure networks. SigGraph [17] most embodies this line of work as it uses such signatures for brute-force memory image scanning. To date, most forensic tools and reverse engineering systems rely on traversing data structures (making use of structural-invariant assumptions) [15, 22, 69–71].

Binary reverse engineering techniques [15,16,70] or unsupervised learning [72] can be used to reverse engineer data structure definitions (e.g., field types) from binaries. Such tools are essential when the subject data structures are entirely unknown. Building upon these, DIMSUM [18] used probabilistic inference to locate known data structures in un-mapped memory. However, DIMSUM requires input data structure definitions to be correct. These works are most related to VCR, but the challenge VCR faces is unique (and not observed in any of these prior works): VCR relies on the availability of the AOSP structure definitions but assumes that they are not correct and therefore employs probabilistic inference to derive signatures for vendor customization.

Compared to the work presented in this dissertation, these techniques represent the traditional methodology of memory forensics: recovering individual pieces of raw data. Their recovery capabilities lack any semantic contextual understanding of this evidence. This limitation is what motivated my initial content reverse engineering efforts pioneered by DSCRETE. My later work moved away from relying on structure definitions, most notably, as a fundamentally new memory forensics technique, RetroScope requires neither structure signature generation nor memory scanning.

Smartphone Memory Forensics. Due to the relatively recent interest in Android memory forensics, few works have focused specifically on the topic. DEC0DE [73] 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 [56,66], several efforts began recovering app-specific data from memory images. Originally, Thing et al. [74] investigated recovering Android in-memory

message-based communications. Sylve et al. [7], followed by my earlier work [43], ported Linux memory analysis tools to recover Android kernel data.

Later, Macht [75] recovered raw Dalvik-JVM control structures. Dalvik Inspector [50] built on that to recover Java objects from app memory dumps. Apostolopoulos et al. [76] recovered login credentials from memory images of certain apps. Lastly, Hilgers et al. [62] proposed using memory analysis on cold-booted Android phones.

The work presented in this dissertation shares the same analysis subjects with these efforts: Android memory images. However, these techniques focus on the recovery of low-level raw data (e.g., Dalvik JVM structures or app-specific login credential). My work has specifically sought to develop *application generic and application agnostic* solutions for both recovering and semantic contextual evidence, which is a step beyond only locating data structure instances. Further, the work presented in this dissertation uniquely enables the fundamentally more powerful capability of spatial-temporal evidence recovery from the same smartphone memory images.

Binary Component Identification and Reuse. At the heart of many of my techniques is application logic reuse. For example, DSCRETE uses dynamic binary program tracing to identify which functional component of a binary application is responsible for generating forensically interesting output. They hence shares some common underlying techniques with existing binary identification and reuse techniques [77–79] and program feature identification [80, 81].

Similar to how DSCRETE employed a data dependence graph, Wong et. al. [80] used program slicing to identify the code region for a program feature. To further understand which application components contribute to an observed runtime behavior, Greevy et al. [81] used feature-driven dynamic analysis to isolate computational units of an application. In contrast, DSCRETE uses only an application's data dependence to identify candidates for later construction of a memory scanner+renderer.

Binary Code Reutilization (BCR) [77] involved using a combination of dynamic and static binary analysis to identify and extract malware encryption and decryption

functions. The goal of BCR was to reuse such extracted logic as a functional component in a different program developed by the user. Inspector Gadget [78] uses dynamic slicing to identify specific malware behavior for extraction and later reuse/analysis. Lin et al. [79] suggested using dynamic slicing to identify applications' functional components to compose reuse-based trojan attacks. In contrast, my work does not aim to extract application logic from a target binary, but rather re-execute it in-place to analyze a memory image and the semantic contextual evidence it contains.

Virtuoso [82] involves using dynamic slicing to identify logic from in-guest applications which could be reused for virtual machine introspection. However, Virtuoso is not able to handle input that is not encountered during off-line training. Later, VMST [83] and Hybrid-Bridge [84] use system-wide instruction monitoring to allow introspection of one VM's kernel data from another. DSCRETE is most similar to works in this area. Compared to VMST, which redirects memory accesses for every instruction of the reused logic, DSCRETE only needs to replace the data structure pointer at the closure point. Further, VMST relies on system call definitions to start logic reuse, while DSCRETE must automatically identify such a starting point (i.e., the closure point) in the subject binary.

7 CONCLUSION

In this dissertation, I have presented a line of research which has proposed a paradigm shift in memory image analysis. My work has purposely broken away from traditional data-recovery-oriented forensics, and instead I have developed a memory forensics framework which leverages program analysis to automatically understand the artifacts that applications leave in a memory image. In doing so, this framework has enabled the recovery of spatial-temporal evidence from only such in-memory artifacts. These four techniques, and the new program analysis techniques which enable them, have introduced new encryption-oblivious forensics capabilities far exceeding traditional data-structure recovery.

DSCRETE reuses an application's own logic from a subject binary program to uncover and render forensically interesting data in a memory image. DSCRETE is able to recreate intuitive, human-observable application output from the memory image, without the burden of reverse engineering data structure definitions.

VCR contributed novel memory forensics techniques to recover key data structures in the face of vendor customizations in order to recover and render photographic evidence from Android device memory images.

To address the real-world smartphone forensics challenge of GUI reconstruction, I presented GUITAR. Instead of focusing on recovering individual data structures, GUITAR pieces back together GUI data structures — already deallocated by Android — to recreate an original GUI.

Finally, RetroScope invented a spatial-temporal memory forensics technique (and new paradigm) that recovers multiple previous screens from an app's memory image. Based on a novel interleaved re-execution engine, RetroScope selectively reanimates an app's screen redrawing functionality without requiring any app-specific knowledge.

Our experiments show that DSCRETE is able to effectively identify interpretation/rendering functions in a variety of real-world applications — overcoming the long standing content reverse engineering challenge. Our tests with a variety of different versions of the Android framework led to several key observations about the importance of VCR rendered photographic evidence and the temporal evidence which they provide to investigations. We found 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. Lastly, RetroScope is shown to recover visually accurate, temporally ordered screens (ranging from 3 to 11 screens) for a variety of apps on three different Android phones.

In conclusion, the robust, encryption-oblivious forensics capabilities realized by this new memory image analysis framework highlight the impactful benefit and possibilities of program-analysis-driven forensics techniques.

REFERENCES

REFERENCES

[1] American Civil Liberties Union. This Map Shows How the Apple-FBI Fight Was About Much More Than One Phone. `https://www.aclu.org/blog/speak-freely/map-shows-how-apple-fbi-fight-was-about-much-more-one-phone`, 2016.

[2] Brendan Saltaformaggio, Zhongshu Gu, Xiangyu Zhang, and Dongyan Xu. DSCRETE: Automatic rendering of forensic information from memory images via application logic reuse. In *Proc. USENIX Security Symposium*, 2014. Best Student Paper Award.

[3] Brendan Saltaformaggio, Rohit Bhatia, Zhongshu Gu, Xiangyu Zhang, and Dongyan Xu. VCR: App-agnostic recovery of photographic evidence from android device memory images. In *Proc. ACM Conference on Computer and Communications Security*, 2015.

[4] Brendan Saltaformaggio, Rohit Bhatia, Zhongshu Gu, Xiangyu Zhang, and Dongyan Xu. GUITAR: Piecing together android app GUIs from memory images. In *Proc. ACM Conference on Computer and Communications Security*, 2015. Best Paper Award.

[5] Brendan Saltaformaggio, Rohit Bhatia, Xiangyu Zhang, Dongyan Xu, and Golden G Richard III. Screen after previous screens: Spatial-temporal recreation of android app displays from memory images. In *Proc. USENIX Security Symposium*, 2016.

[6] Brian D Carrier and Joe Grand. A hardware-based memory acquisition procedure for digital investigations. *Digital Investigation*, 1, 2004.

[7] Joe Sylve, Andrew Case, Lodovico Marziale, and Golden G Richard. Acquisition and analysis of volatile memory from android devices. *Digital Investigation*, 8, 2012.

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

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

[10] Nick L Petroni Jr, Aaron Walters, Timothy Fraser, and William A Arbaugh. FATKit: A framework for the extraction and analysis of digital forensic data from volatile system memory. *Digital Investigation*, 3, 2006.

[11] Brendan Dolan-Gavitt, Abhinav Srivastava, Patrick Traynor, and Jonathon Giffin. Robust signatures for kernel data structures. In *Proc. ACM Conference on Computer and Communications Security*, 2009.

[12] Chris Betz. Memparser forensics tool. `http://www.dfrws.org/2005/challenge/memparser.shtml`, 2005.

[13] C Bugcheck. Grepexec: Grepping executive objects from pool memory. In *Proc. Digital Forensic Research Workshop*, 2006.

[14] Martim Carbone, Weidong Cui, Long Lu, Wenke Lee, Marcus Peinado, and Xuxian Jiang. Mapping kernel objects to enable systematic integrity checking. In *Proc. ACM Conference on Computer and Communications Security*, 2009.

[15] Zhiqiang Lin, Xiangyu Zhang, and Dongyan Xu. Automatic reverse engineering of data structures from binary execution. In *Proc. Network and Distributed System Security Symposium*, 2010.

[16] Asia Slowinska, Traian Stancescu, and Herbert Bos. Howard: A dynamic excavator for reverse engineering data structures. In *Proc. Network and Distributed System Security Symposium*, 2011.

[17] Zhiqiang Lin, Junghwan Rhee, Xiangyu Zhang, Dongyan Xu, and Xuxian Jiang. SigGraph: Brute force scanning of kernel data structure instances using graph-based signatures. In *Proc. Network and Distributed System Security Symposium*, 2011.

[18] Zhiqiang Lin, Junghwan Rhee, Chao Wu, Xiangyu Zhang, and Dongyan Xu. DIMSUM: Discovering semantic data of interest from un-mappable memory with confidence. In *Proc. Network and Distributed System Security Symposium*, 2012.

[19] Bogdan Korel and Janusz Laski. Dynamic program slicing. *Information Processing Letters*, 29(3), 1988.

[20] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *ACM SIGPLAN Notices*, volume 40, 2005.

[21] Daniel Ayers. A second generation computer forensic analysis system. *Digital Investigation*, 6, 2009.

[22] Junyuan Zeng, Yangchun Fu, Kenneth A. Miller, Zhiqiang Lin, Xiangyu Zhang, and Dongyan Xu. Obfuscation resilient binary code reuse through trace-oriented programming. In *Proc. ACM Conference on Computer and Communications Security*, 2013.

[23] Riley v. California. 134 S. Ct. 2473, (2014).

[24] Brian D Carrier. Risks of live digital forensic analysis. *Communications of the ACM*, 49(2), 2006.

[25] Frank Adelstein. Live forensics: Diagnosing your system without killing it first. *Communications of the ACM*, 49(2), 2006.

[26] Rolando R. Lopez. Battling Human Trafficking with Big Data. Invited talk, USENIX Security Symposium, 2014.

[27] Per-Erik Danielsson. Euclidean distance mapping. *Computer Graphics and image processing*, 14(3), 1980.

[28] Google, Inc. Android dashboards - platform versions. `https://developer.android.com/about/dashboards/index.html`, 2015.

[29] Qi Alfred Chen, Zhiyun Qian, and Z Morley Mao. Peeking into your app without actually seeing it: UI state inference and novel android attacks. In *Proc. USENIX Security Symposium*, 2014.

[30] Chia-Chi Lin, Hongyang Li, Xiaoyong Zhou, and XiaoFeng Wang. Screenmilker: How to milk your android screen for secrets. In *Proc. Network and Distributed System Security Symposium*, 2014.

[31] Gates Rubber Co. v. Bando Chemical Industries, Ltd. 9 F. 3d 823, (1993).

[32] Schaghticoke Tribal Nation v. Kempthorne. 587 F. Supp. 2d 389, (2008).

[33] US v. Scholle. 553 F. 2d 1109, (1977).

[34] US v. Vela. 673 F. 2d 86, (1982).

[35] US v. Bonallo. 858 F. 2d 1427, (1988).

[36] John Paul Mitchell Systems v. Quality King Distributors, Inc. 106 F. Supp. 2d 462, (2000).

[37] Pearl Brewing Co. v. Jos. Schlitz Brewing Co. 415 F. Supp. 1122, (1976).

[38] Illinois Tool Works v. Metro Mark Products, Ltd. 43 F. Supp. 2d 951, (1999).

[39] Nat. Union Elec. Corp. v. Matsushita Elec. Indus. Co. 494 F. Supp. 1257, (1980).

[40] John Ashcroft, Deborah J Daniels, and Sara 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.

[41] H Marshall Jarrett, Michael 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.

[42] 7 American Law Reports. 4th, 8, 2b.

[43] Brendan Saltaformaggio. Forensic carving of wireless network information from the android linux kernel. *University of New Orleans Theses and Dissertations, Paper 20*, 2012.

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

[45] Hungarian algorithm method source. `https://github.com/maandree/hungarian-algorithm-n3/blob/master/hungarian.c`, 2014.

[46] Mathias Lux and Savvas A Chatzichristofis. Lire: Lucene image retrieval: An extensible java CBIR library. In *Proc. ACM International Conference on Multimedia*, 2008.

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

[48] Savvas A Chatzichristofis and Yiannis S Boutalis. CEDD: Color and edge directivity descriptor: a compact descriptor for image indexing and retrieval. In *Computer Vision Systems*. 2008.

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

[50] 504ENSICS Labs. Dalvik Inspector. `http://www.504ensics.com/automated-volatility-plugin-generation-with-dalvik-inspector/`, 2013.

[51] George H Mealy. A Method for Synthesizing Sequential Circuits. *Bell System Technical Journal*, 34(5):1045–1079, 1955.

[52] 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.

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

[54] 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.

[55] Ivor Kollár. Forensic ram dump image analyser. *Master's Thesis, Charles University in Prague*, 2010.

[56] 504ENSICS Labs. LiME Linux Memory Extractor. `https://github.com/504ensicsLabs/LiME`, 2013.

[57] Nick Petroni, Timothy Fraser, Jesus Molina, and William Arbaugh. Copilot - a coprocessor-based kernel runtime integrity monitor. In *Proc. USENIX Security Symposium*, 2004.

[58] Michael Becher, Maximillian Dornseif, and Christian Klein. Firewire: All your memory are belong to us. *CanSecWest*, 2005.

[59] Jesse D Kornblum. Using every part of the buffalo in windows memory analysis. *Digital Investigation*, 4, 2007.

[60] Michael Gruhn. Windows NT pagefile.sys virtual memory analysis. In *Proc. IT Security Incident Management & IT Forensics (IMF)*, 2015.

[61] Golden G Richard and Andrew Case. In lieu of swap: Analyzing compressed ram in mac os x and linux. *Digital Investigation*, 11, 2014.

[62] Christian Hilgers, Holger Macht, Tilo Muller, and Michael Spreitzenbarth. Post-mortem memory analysis of cold-booted android devices. In *Proc. IT Security Incident Management & IT Forensics (IMF)*, 2014.

[63] J Alex Halderman, Seth D Schoen, Nadia Heninger, William Clarkson, William Paul, Joseph A Calandrino, Ariel J Feldman, Jacob Appelbaum, and Edward W Felten. Lest we remember: Cold-boot attacks on encryption keys. In *Proc. USENIX Security Symposium*, 2008.

[64] Timothy Vidas. Volatile memory acquisition via warm boot memory survivability. In *Proc. Hawaii International Conference on System Sciences*, 2010.

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

[66] He Sun, Kun Sun, Yuewu Wang, Jiwu Jing, and Sushil Jajodia. Trustdump: Reliable memory acquisition on smartphones. In *Proc. European Symposium on Research in Computer Security*. 2014.

[67] Seung Jei Yang, Jung Ho Choi, Ki Bom Kim, and Taejoo Chang. New acquisition method based on firmware update protocols for android smartphones. *Digital Investigation*, 14, 2015.

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

[69] Andrew Case, Andrew Cristina, Lodovico Marziale, Golden G Richard, and Vassil Roussev. FACE: Automated digital evidence discovery and correlation. *Digital Investigation*, 5, 2008.

[70] JongHyup Lee, Thanassis Avgerinos, and David Brumley. TIE: Principled reverse engineering of types in binary programs. In *Proc. Network and Distributed System Security Symposium*, 2011.

[71] Paul Movall, Ward Nelson, and Shaun Wetzstein. Linux physical memory analysis. In *Proc. USENIX Annual Technical Conference, FREENIX Track*, 2005.

[72] Anthony Cozzie, Frank Stratton, Hui Xue, and Samuel T King. Digging for data structures. In *Proc. Symposium on Operating Systems Design and Implementation*, 2008.

[73] Robert Walls, Brian N Levine, and Erik G Learned-Miller. Forensic triage for mobile phones with DEC0DE. In *Proc. USENIX Security Symposium*, 2011.

[74] Vrizlynn LL Thing, Kian-Yong Ng, and Ee-Chien Chang. Live memory forensics of mobile phones. *Digital Investigation*, 7, 2010.

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

[76] Dimitris Apostolopoulos, Giannis Marinakis, Christoforos Ntantogian, and Christos Xenakis. Discovering authentication credentials in volatile memory of android mobile devices. In *Collaborative, Trusted and Privacy-Aware e/m-Services*. 2013.

[77] Juan Caballero, Noah M Johnson, Stephen McCamant, and Dawn Song. Binary code extraction and interface identification for security applications. In *Proc. Network and Distributed System Security Symposium*, 2010.

[78] Clemens Kolbitsch, Thorsten Holz, Christopher Kruegel, and Engin Kirda. Inspector Gadget: Automated extraction of proprietary gadgets from malware binaries. In *Proc. IEEE Symposium on Security and Privacy*, 2010.

[79] Zhiqiang Lin, Xiangyu Zhang, and Dongyan Xu. Reuse-oriented camouflaging trojan: Vulnerability detection and attack construction. In *Proc. IEEE/IFIP International Conference on Dependable Systems and Networks*, 2010.

[80] W Eric Wong, Swapna S Gokhale, and Joseph R Horgan. Quantifying the closeness between program components and features. *Journal of Systems and Software*, 54(2), 2000.

[81] Orla Greevy and Stéphane Ducasse. Correlating features and code using a compact two-sided trace analysis approach. In *Proc. European Conference on Software Maintenance and Reengineering*, 2005.

[82] Brendan Dolan-Gavitt, Tim Leek, Michael Zhivich, Jonathon Giffin, and Wenke Lee. Virtuoso: Narrowing the semantic gap in virtual machine introspection. In *Proc. IEEE Symposium on Security and Privacy*, 2011.

[83] Yangchun Fu and Zhiqiang Lin. Space traveling across VM: Automatically bridging the semantic gap in virtual machine introspection via online kernel data redirection. In *Proc. IEEE Symposium on Security and Privacy*, 2012.

[84] Yangchun Saberi, Alireza Fu and Zhiqiang Lin. Hybrid-Bridge: Efficiently bridging the semantic gap in virtual machine introspection via decoupled execution and training memoization. In *Proc. Network and Distributed System Security Symposium*, 2013.

VITA

VITA

Brendan Dominic Saltaformaggio earned a Master of Science in Computer Science from Purdue University and a Bachelor of Science with Honors in Computer Science from the University of New Orleans in New Orleans, LA (where he was born and raised). His research interests lie in computer systems security and cyber forensics with focuses on memory forensics, binary analysis and instrumentation, vetting of untrusted software, and cloud computing security. His work has been awarded the Best Student Paper Award at Usenix Security 2014 and the Best Paper Award at ACM CCS 2015. His Ph.D. research has been partially funded via the 2016 Symantec Research Labs Graduate Fellowship, and he was recently honored as the inaugural recipient of the Emil Stefanov Memorial Fellowship. In the Spring of 2017, he will begin a brief appointment as a postdoctoral researcher with his advisors at Purdue. At the time of writing this dissertation, he is beginning his job search for a position as an Assistant Professor to begin in the Fall of 2017.