Convicted By Memory: Automatically Recovering Spatial-Temporal Evidence From Memory Images

Brendan Saltaformaggio

Assistant Professor - Georgia Institute of Technology
Director - Cyber Forensics Innovation (CyFI) Lab
CyFI Lab Research: Computer System Security And Cyber Forensics

Program-Analysis-Driven Memory Forensics
[Usenix Security ’14 Best Student Paper] [CCS’15] [CCS’15 Best Paper] [Usenix Security ’16] [NDSS’18]

Software Vetting and Security
[NDSS’17] [ACSAC’17] [ASPLOS’16] [WOOT’16] [CCS’15]

Cloud and Enterprise System Security
[ACSAC’17] [ACSAC’16] [DSN’14] [EuroSec’14]

Underpinning all of our research is the development of fundamental techniques for:
• Binary program analysis and instrumentation
• Digital evidence modeling
• Virtualization-based defense techniques
Memory Forensics

Memory forensics is the process of extracting and analyzing evidence from an image of a device’s volatile RAM.

- Running Processes
- Volatile IPC Data
- Executing Malware
- Network Connections
- Encryption Keys
- Application Data (Photos, Chat Logs, …)
- Much More …

Memory forensics can reveal “up to the minute” evidence about a device’s usage.

Importantly, memory forensics:
- Does not require a suspect's password to unlock the device
- Oblivious to any persistent (non-volatile) storage encryption schemes
What Is In-Memory Evidence?

Evidence in memory is stored in Data Structures
What Is In-Memory Evidence?

Investigators collect a memory image.

Reverse engineer signatures of data structures to recover.

Scan the memory image to find data structure instances.
(Prior) State Of The Art: Data Structure Recovery

Evidence is recovered from plain-text or self-evident fields

```c
struct user_account {
    [0x00] short int u_type;
    [0x04] pid_t u_pid;
    [0x08] char u_line[32];
    [0x28] char uid[4];
    [0x2C] char user[32];
    [0x4C] char password[128];
    [0xCC] char u_host[128];
    [0x14C] short e_termination;
    ...
}
```
A Cyber-Crime

Based on true events that occurred at the authors' university...
State Of The Art.. But Limited

Recovering raw data structure instances from a memory image

Still cannot *understand the content* of the data structure!

E.g., images, documents, any formatted/encoded data

Can you tell this is a PDF?
DSCRETE: Content Reverse Engineering
[Usenix Security ’14, Best Student Paper]

The application that defined the data structure contains printing/rendering logic for it too!
**PDF Editor Program Code**

```c
struct pdf* _pdf;
_pdf = load_pdf_file(...);
main_loop(_pdf); // User edits PDF
save_pdf_file(_pdf);
exit(0);
```

**Rendering Function**

```c
save_pdf_file(struct pdf* ptr) {
    char* buf = format_pdf(ptr);
    fwrite(buf, ...);
}
```
Basic Idea: Scanner+Renderer

```c
Rendering Function
save_pdf_file(struct pdf* ptr)
{
    char* buf = format_pdf(ptr);
    fwrite(buf, ...);
}
```

Present every offset of a memory image to the rendering logic

Intuition: Invalid input will crash the rendering logic
Basic Idea: Scanner+Renderer

```c
Rendering Function
save_pdf_file(struct pdf* ptr)
{
    char* buf = format_pdf(ptr);
    fwrite(buf, …);
}
```

But, a valid data structure will produce valid application output
Cross-State Execution

PDF Editor Program Code

```c
struct pdf* _pdf;
_pdf = load_pdf_file(...);
main_loop(_pdf); // User edits PDF
save_pdf_file(_pdf);
exit(0);

save_pdf_file(struct pdf* ptr)
{
    char* buf = format_pdf(ptr);
    fwrite(buf, ...);
}
```

Identified Entry Point
PDF Editor Program Code

```c
struct pdf* _pdf;
_pdf = load_pdf_file(...);
main_loop(_pdf); // User edits PDF
save_pdf_file(_pdf);
exit(0);

save_pdf_file(struct pdf* ptr)
{
    char* buf = format_pdf(ptr);
    fwrite(buf, ...);
}
```
Cross-State Execution

PDF Editor Program Code

```
struct pdf* _pdf;
_pdf = load_pdf_file(...);
main_loop(_pdf); // User edits PDF
save_pdf_file(_pdf);
exit(0);
```

Begin Cross-State Execution!

1. Map in memory image
2. Swap _pdf pointer
Cross-State Execution

App’s Memory

Suspect Memory Image

PDF Editor Program Code

```c
struct pdf* _pdf;
_pdf = load_pdf_file(...);
main_loop(_pdf); // User edits PDF
save_pdf_file(_pdf);
exit(0);
```

```c
save_pdf_file(struct pdf* ptr)
{
    char* buf = format_pdf(ptr);
    fwrite(buf, ...);
}
```
Let’s Catch That Criminal...

Back at the Forensics Lab...
Evidence Recovery

App.
- convert
- gnome-paint
- gThumb
- gnome-screen
- Nginx
- PDFedit
- top
- Xfig
- CenterIM
- darktable
- Firefox
- SQL query
- SQL log

<table>
<thead>
<tr>
<th>App</th>
<th>Image</th>
<th>FN</th>
<th>FN%</th>
<th>FP</th>
<th>FP%</th>
</tr>
</thead>
<tbody>
<tr>
<td>convert</td>
<td></td>
<td>0</td>
<td>0%</td>
<td>0</td>
<td>0%</td>
</tr>
<tr>
<td>gnome-paint</td>
<td></td>
<td>0</td>
<td>0%</td>
<td>0</td>
<td>0%</td>
</tr>
<tr>
<td>gThumb</td>
<td></td>
<td>0</td>
<td>0%</td>
<td>0</td>
<td>0%</td>
</tr>
<tr>
<td>gnome-screen</td>
<td></td>
<td>0</td>
<td>0%</td>
<td>1</td>
<td>0.4%</td>
</tr>
<tr>
<td>Nginx</td>
<td></td>
<td>0</td>
<td>0%</td>
<td>0</td>
<td>0%</td>
</tr>
<tr>
<td>PDFedit</td>
<td></td>
<td>0</td>
<td>0%</td>
<td>0</td>
<td>0%</td>
</tr>
<tr>
<td>top</td>
<td></td>
<td>0</td>
<td>0%</td>
<td>0</td>
<td>0%</td>
</tr>
<tr>
<td>Xfig</td>
<td></td>
<td>0</td>
<td>0%</td>
<td>0</td>
<td>0%</td>
</tr>
<tr>
<td>CenterIM</td>
<td></td>
<td>0</td>
<td>0%</td>
<td>0</td>
<td>0%</td>
</tr>
<tr>
<td>darktable</td>
<td></td>
<td>0</td>
<td>0%</td>
<td>0</td>
<td>0%</td>
</tr>
<tr>
<td>Firefox</td>
<td></td>
<td>0</td>
<td>0%</td>
<td>0</td>
<td>0%</td>
</tr>
</tbody>
</table>
DSCRETE Shifted The Goal Of Memory Forensics

This coincided with a surge in the importance of smartphones in cyber investigations and legal proceedings.
The Most Obvious Evidence Is The GUI Complex

A GUI displays a variety of contextual evidence (e.g., text, images, graphics)

An Android app’s GUI is a virtual billboard of highly diverse, application-specific data structures

New Challenge: Cannot reverse engineer the vast number of data structures for every app

Our Goal: App-Agnostic Recovery

We must find an automated and general solution for GUI recovery
The Android GUI library sets up a GUI Tree in the app’s memory.

This GUI Tree **generically encodes** the GUI’s geometric structure and graphical content.
The Android GUI library sets up a GUI Tree in the app’s memory.

This GUI Tree **generically encodes** the GUI’s geometric structure and graphical content.

**Problem Solved?**
When an app is sent to the background, Android explicitly frees and nullifies some of the GUI Tree’s connections.
When an app is sent to the background, Android explicitly frees and nullifies some of the GUI Tree’s connections.
GUITAR: GUI Tree ARchaeology [CCS ’15, Best Paper]

TreeNodes

DrawOpLists

Reassembled GUI Tree

Remapped DrawOpLists (graphical content)

App-Agnostic GUI Recovery

Instrumented Android GUI Binary
GUITAR: GUI Tree ARchaeology [CCS ’15, Best Paper]

Step 1: Recover the hundreds or thousands of disconnected GUI components
Step 2: Reconstruct the GUI Tree’s structure

Depth-first **topology recovery algorithm** rebuilds the lost parent-to-child structure
Step 3: Reconnect TreeNodes to their DrawOpLists

Unfortunately, all connections between TreeNodes and DrawOpLists have been nullified!
Step 3: Remapping Drawing Operations

Key Observation: TreeNode describes a geometrical portion of the screen

Recovered Leaf TreeNodes

Recovered DrawOpLists
- DrawBitmapOp
- ClipRectOp
- TranslateOp
- DrawBitmapOp
- TranslateOp
- DrawTextOp
Step 3: Remapping Drawing Operations

Key Observation: TreeNode describe a geometrical portion of the screen

And...

Each DrawOpList contains renderable graphic content
Piecing Together The GUI

More Formally: Global best fit of GUI graphical components

Recovered Leaf TreeNodes

Recovered DrawOpLists

DrawBitmapOp
ClipRectOp
TranslateOp
DrawBitmapOp
TranslateOp
DrawTextOp
Piecing Together The GUI

More Formally: Global best fit of GUI graphical components

Model the problem as a Weighted Bipartite Graph Matching

Node “fitness” is defined by the drawing-content
GUITAR In Practice

At the Forensics Lab...
Bypassing The Password

Many highly-secure apps require users to log in when they bring the app to the foreground.

But GUITAR recovers the last screen the user was viewing before backgrounding the app.
Before the investigation began, the suspect was interacting with their apps...

Without access to the suspect’s password or breaking Telegram’s fully encrypted storage!
## The Limitation Of GUITAR

<table>
<thead>
<tr>
<th>Screen 0</th>
<th>Screen 1</th>
<th>Screen 2</th>
<th>Screen 3</th>
<th>Screen 4</th>
<th>Screen 5</th>
<th>Screen 6</th>
</tr>
</thead>
</table>

### In Memory GUI Data:

- Screen 0
- Screen 1
- Screen 2
- Screen 3
- Screen 4
- Screen 5
- Screen 6

### Time:
Are The Old Screens Really Gone? ... Yes And No

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

App Internal Data
Not for GUI drawing:
- Raw Chat Strings,
- Account Balance,
Android Asks The App To Draw A Screen

Android sends a Redraw Command

1) A Canvas is sent for the app to fill
   - Apps register *draw* routines with Android
2) The app builds GUI structures which “package” the internal data
   - Destroying the previous screen!
3) The filled Canvas is rendered on the device’s screen
Idea: Ask The Memory Image To Draw A Screen

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

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

3) Memory = Static Data
   - Execution context is gone

Our Goal: “Plug And Play” App-Agnostic Recovery
RetroScope: Spatial-Temporal Memory Forensics

[Usenix Security ’16]

From a single memory snapshot (one time instance) RetroScope recovers entire temporally-ordered sequences of past GUI screens for any app (an entire time interval)

This introduced a new paradigm of Spatial-Temporal Evidence Recovery for memory forensics
RetroScope: Spatial-Temporal Memory Forensics
[Usenix Security ’16]

RetroScope invented a novel binary instrumentation and steering technique (“the puppeteer”)

Generically interleave the execution of a live Android environment with code and data from a memory image.
Execution puppeteering is handled by RetroScope’s Binary Steering Engine.
Execution Puppeteering

Execution puppeteering is handled by RetroScope’s Binary Steering Engine.

We formally modeled the interleaving of execution states as a finite automata with state transition rules guided by the executing instruction semantics.

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

Steer the execution to the memory image app’s drawing routines

RetroScope monitors the state transitions and corrects the execution
Selective Reanimation

RetroScope guides the execution of the memory image routines to naturally access its internal data.
Selective Reanimation

Function calls for the new Canvas are directed to the live GUI environment (manages the state mixing).
Selective Reanimation

Finally: the newly filled Canvas is rendered by the live GUI environment and saved.
Selective Reanimation

This process repeats for each registered screen drawing function in the memory image.
Breaking The Case Wide Open!
Deleted Messages: WeChat (And Others)

From LG G3 Device
RetroScope’s Impact

RetroScope immediately garnered media attention
The Register, NSF News, ACM TechNews, IEEE Electronics360
Homeland Preparedness News (full list on my webpage)

The precursor to RetroScope (GUITAR) was awarded the Best Paper Award at CCS’15 for recovering only a single screen (spatial recovery only)

Invited to give an “Android Memory Forensics” TechTalk by Google’s Anti-Abuse Research Team

Opened a new dialog about Citizen Privacy versus Forensic Capabilities
IOT/Cyber Physical Forensics And Security

Cross-domain techniques for evidence acquisition and analysis for cyber-physical attacks (e.g., APTs) against autonomous vehicles

Combine investigation of the vehicle’s control software execution, physical system operation, human operation, and their interplay

Tight integration of physical and cyber evidence analysis e.g., FDR records, program execution traces, memory/disk images, wireless communication traces...

Simultaneously address recoverability and integrity of evidence e.g., forensics-aware chips, persistent RAM, data lifetime

“According to accident-investigation reports and other records, since 2001 there have been 237 military drone crashes [...] that destroyed the aircraft.”

- Washington Post
Conclusion

My research leverages program analysis to automatically reconstruct spatial-temporal evidence from memory images.

Future Work: Cyber attack investigation and defense across cyber/cyber-physical environments.
Thank you!
Questions?

Brendan Saltaformaggio
brendan@ece.gatech.edu
https://saltaformaggio.ece.gatech.edu
https://cyfi.ece.gatech.edu
(We are hiring!)
“Graceful” loss of backgounded GUI

LG G3 Contacts App

1 hour

5 hours

24 hours
Step 1: Find The Rendering Logic

Execute the binary under dynamic analysis

DSCRETE collects an execution trace

Mark the output functions that emits the evidence

E.g., fwrite( ... ) that saved PDF file

Execution Trace

```
main( )
    push ebx
    mov   ebx, [eax+0Ch]
push ebx
xor   esi,esi
lea   ebx, [edx+12h]
jle   40105C
add   esp,8
mov   eax, [esp+14h]
mov   [ebx], eax
push esi
call  401530
fwrite(buf, ...)
```
Step 1: Find The Rendering Logic

Execute the binary under dynamic analysis

DSCRETE collects an execution trace

Mark the output functions that emits the evidence

e.g., fwrite( ... ) that saved PDF file

DSCRETE searches backward in the trace for the output evidence buffer
Step 1: Find The Rendering Logic

Execute the binary under dynamic analysis
  DSCRETE collects an execution trace

Mark the output functions that emits the evidence
  e.g., fwrite(... ) that saved PDF file

DSCRETE searches backward in the trace for the output evidence buffer

Then identifies all instructions in the trace that contributed to the buffer’s final value
  The rendering logic must be within that slice
Step 2: Isolate Entry Point

Need to isolate the entry point of the rendering logic

This is where DSCRETE can input the suspect’s data structure, e.g., `save_pdf_file(struct pdf*)`
Step 2: Isolate Entry Point

Need to isolate the entry point of the rendering logic

This is where DSCRETE can input the suspect’s data structure, e.g., `save_pdf_file(struct pdf*)`

DSCRETE tests “candidate” entry point instructions

Candidate instructions must:

1. Take a heap pointer as input
2. All selected output data must depend on it

The isolated rendering logic can now be reused for a Scanner+Renderer, but DSCRETE must first set up the tool’s execution environment
Why not interact with the device?

Law Enforcement follow Strict Legal Requirements for Digital Evidence Preservation: Evidence must be collected and examined in **minimally intrusive** and **reproducible** ways


Manual interaction with the device may raise **Reasonable Doubt** in court:
One side may not be able to reproduce (app with locking screen) “Uncontrolled Environment” while interacting with the device (e.g. The Trojan Defense – “malware did it!”)

Instead: “Image and Analyze” without fear of jeopardizing the investigation
Privacy Implications of RetroScope?

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

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

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

The RetroScope technique is not specific to any Android version
- The same evidence is present in Android v. 4.1 (maybe earlier) – 6.0 (newest)

Like any forensics technique, may need porting if future Android versions change resources RetroScope uses
- Easier than porting Linux tools to Windows
- Our paper discusses porting to future Android versions

Our testing devices & current implementation are on Android 4.4.2
- These devices are ~2 years & still the most widely used

Future work requires porting our framework to Android 6.0 ... stay tuned!
# DSCRETE or RetroScope?

<table>
<thead>
<tr>
<th></th>
<th>DSCRETE</th>
<th>RetroScope</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data Recovered</td>
<td>Any Single <strong>Recoverable</strong> Data Structure</td>
<td>Thousands of GUI Structures <strong>Rebuilt</strong> from Internal Data</td>
</tr>
<tr>
<td>Methodology</td>
<td>App-Specific (Structure-Specific) Training &amp; Logic Extraction</td>
<td>App-Agnostic In-Place “Retargeting” of Draw Routines</td>
</tr>
<tr>
<td>Enabling Techniques</td>
<td>Binary Logic Identification (e.g., slicing/extraction)</td>
<td>Interleaved Re-Execution Engine</td>
</tr>
<tr>
<td></td>
<td>1-to-1 Data Structure Scanning + Rendering</td>
<td>Executes code on a <strong>syntactic basis</strong> without understanding the semantics</td>
</tr>
</tbody>
</table>