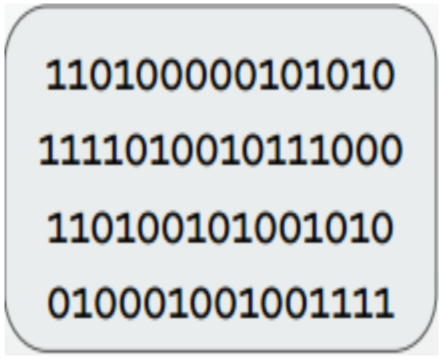
Automatic Rendering of Forensic Information from Memory Images via Application Logic Reuse

Brendan Saltaformaggio, ZhongshuGu, Xiangyu Zhang, and Dongyan Xu

Presented By
Sharani Sankaran

Memory Forensics

- Digital investigation based on analysis of non-volatile storage.
- Loss of live evidence stored in system RAM
- Information stored in RAM: executing processes open network connections volatile IPC data OS and application data structure



How It works

- It mainly capture an image of the suspect machine's volatile memory.
- The hardware and software based memory acquisition tools that are minimally invasive.
- It analyses the resulting memory image using memory analysis tools.
- The main aim is to recreate the system's previously observable state based on the memory image.

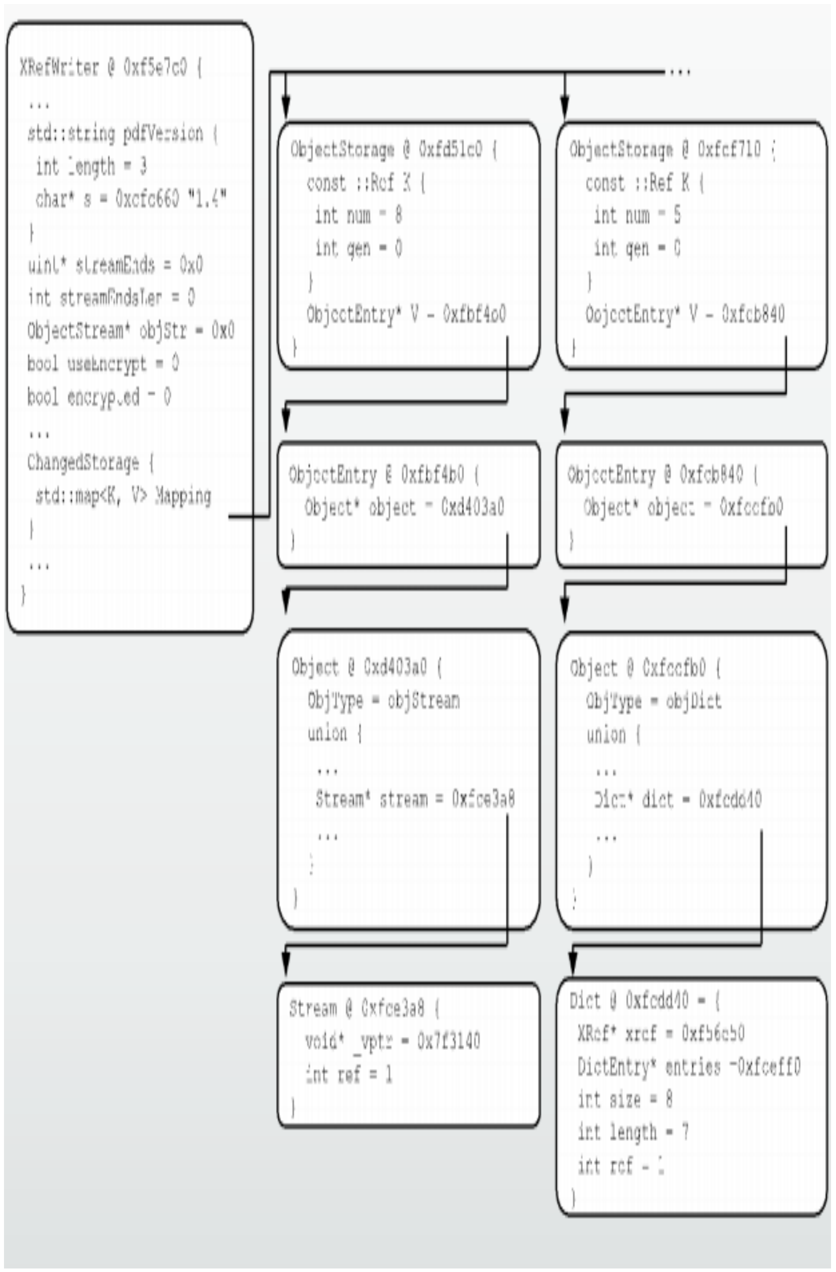
State Of Art Memory

Signature based Scanning:

- The data structure signature is mainly derived by analyzing program binaries.
- The signature is used to scan memory images and identify the instances of data structures.
- It also present contents of identified instances to forensic investigators as potential evidence.

➢It mainly finds raw data structure instances in memory image.

➢Thus understanding the content of these data structures is extremely difficult or impossible.



```
XRefWriter @ 0xf5e7c0 {
 ...
 std::string pdfVersion {
  int length = 3
  char* s = 0xcfc660 "1.4"
 }
 uint* streamEnds = 0x0
 int streamEndsLen = 0
 ObjectStream* objStr = 0x0
 bool useEncrypt = 0
 bool encrypted - 0
 ...
 ChangedStorage {
  std::map<K, V> Mapping
 }
 ...
}
```

```
ObjectStorage @ 0xfd51c0 {
  const ::Ref K {
   int num - 8
   int gen = 0
  }
  ObjectEntry* V - 0xfbf4o0
}
```

```
ObjectStorage @ 0xfcf710 {
  const ::Ref K {
   int num - 5
   int gen = 0
  }
  ObjectEntry* V - 0xfcb840
}
```

```
ObjectEntry @ 0xfbf4b0 {
  Object* object - 0xd403a0
}
```

```
ObjectEntry @ 0xfcb840 {
  Object* object - 0xfccfb0
}
```

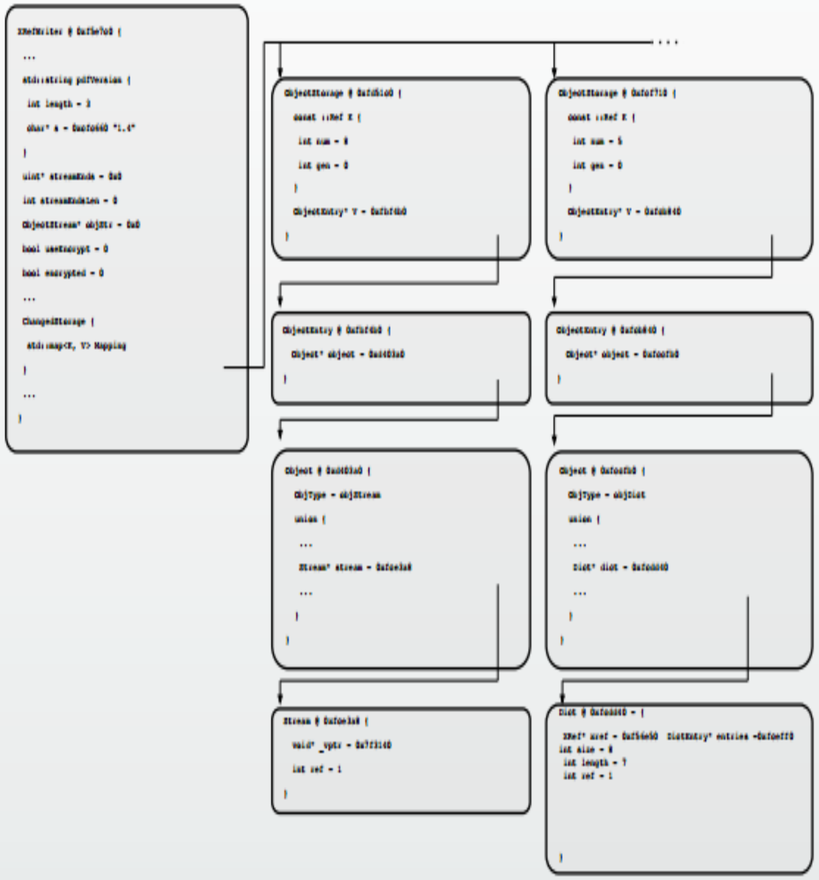
```
Object @ 0xd403a0 {
  ObjType = objStream
  union {
   ...
   Stream* stream = 0xfce3a8
   ...
  }
}
```

```
Object @ 0xfccfb0 {
  ObjType = objDict
  union {
   ...
   Dict* dict = 0xfcdd40
   ...
  }
}
```

```
Stream @ 0xfce3a8 {
  void* _vptr = 0x7f3140
  int ref = 1
}
```

```
Dict @ 0xfcdd40 = {
 XRef* xref = 0xf56cb0
 DictEntry* entries -0xfceff0
 int size = 8
 int length = 7
 int ref - 1
}
```

Content Reverse Engineering challenge



```
XRefWriter @ 0xf5e7c0 {
 ...
 std::string pdfVersion {
  int length = 3
  char* s = 0xcfc660 "1.4"
 }
 uint* streamEnds = 0x0
 int streamEndsLen = 0
 ObjectStream* objStr = 0x0
 bool useEncrypt = 0
 bool encrypted = 0
 ...
 ChangedStorage {
  std::map<K, V> Mapping
 }
 ...
}
```

```
ObjectStorage @ 0xfd51c0 {
  const ::Ref K {
   int num = 8
   int gen = 0
  }
  ObjectEntry* V = 0xfbf4b0
}
```

```
ObjectStorage @ 0xfcf710 {
  const ::Ref K {
   int num = 5
   int gen = 0
  }
  ObjectEntry* V = 0xfdb840
}
```

```
ObjectEntry @ 0xfbf4b0 {
  Object* object = 0xd4b1a0
}
```

```
ObjectEntry @ 0xfdb840 {
  Object* object = 0xfcef80
}
```

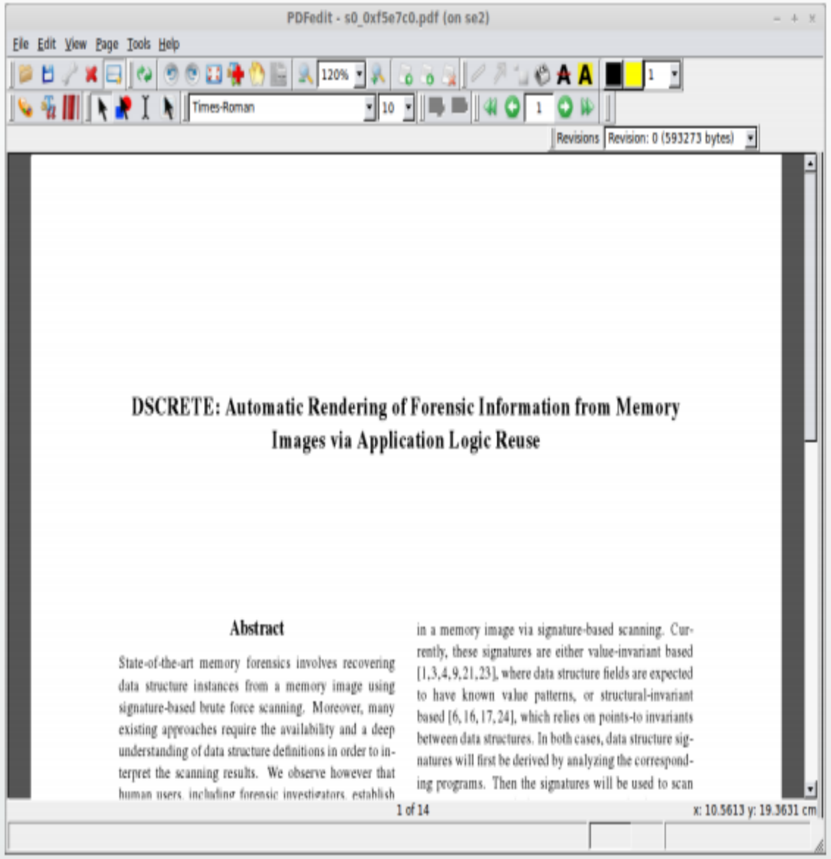
```
Object @ 0xd4b1a0 {
  ObjType = objStream
  union {
   ...
   Stream* stream = 0xfce1a8
   ...
  }
}
```

```
Object @ 0xfcef80 {
  ObjType = objDict
  union {
   ...
   Dict* dict = 0xfcd600
   ...
  }
}
```

```
Stream @ 0xfce1a8 {
  void* _vptr = 0x7f3140
  int ref = 1
}
```

```
Dict @ 0xfcd600 = {
 XRef* xref = 0xf5e6d0  DictEntry* entries =0xfcef70
int size = 8
 int length = 7
 int ref = 1
}
```

(a) Signature-based scanner output.


PDFedit - s0_0xf5e7c0.pdf (on se2)
File Edit View Page Tools Help
120%
Times-Roman
10
1
Revisions
Revision: 0 (593273 bytes)
1 of 14
x: 10.5613 y: 19.3631 cm

DSCRETE: Automatic Rendering of Forensic Information from Memory Images via Application Logic Reuse

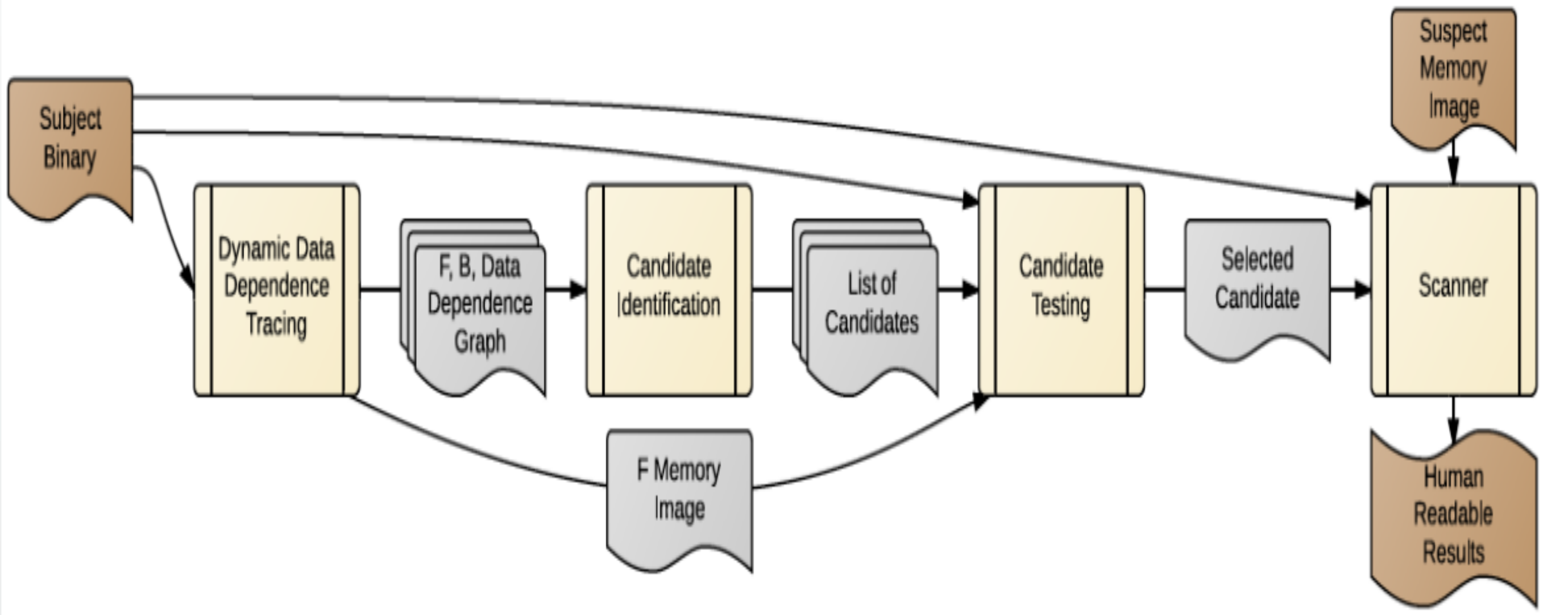
Abstract

State-of-the-art memory forensics involves recovering data structure instances from a memory image using signature-based brute force scanning. Moreover, many existing approaches require the availability and a deep understanding of data structure definitions in order to interpret the scanning results. We observe however that human users, including forensic investigators, establish

in a memory image via signature-based scanning. Currently, these signatures are either value-invariant based [1,3,4,9,21,23], where data structure fields are expected to have known value patterns, or structural-invariant based [6,16,17,24], which relies 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

(b) DSCRETE-based scanner output.

DISCRETE WORKFLOW

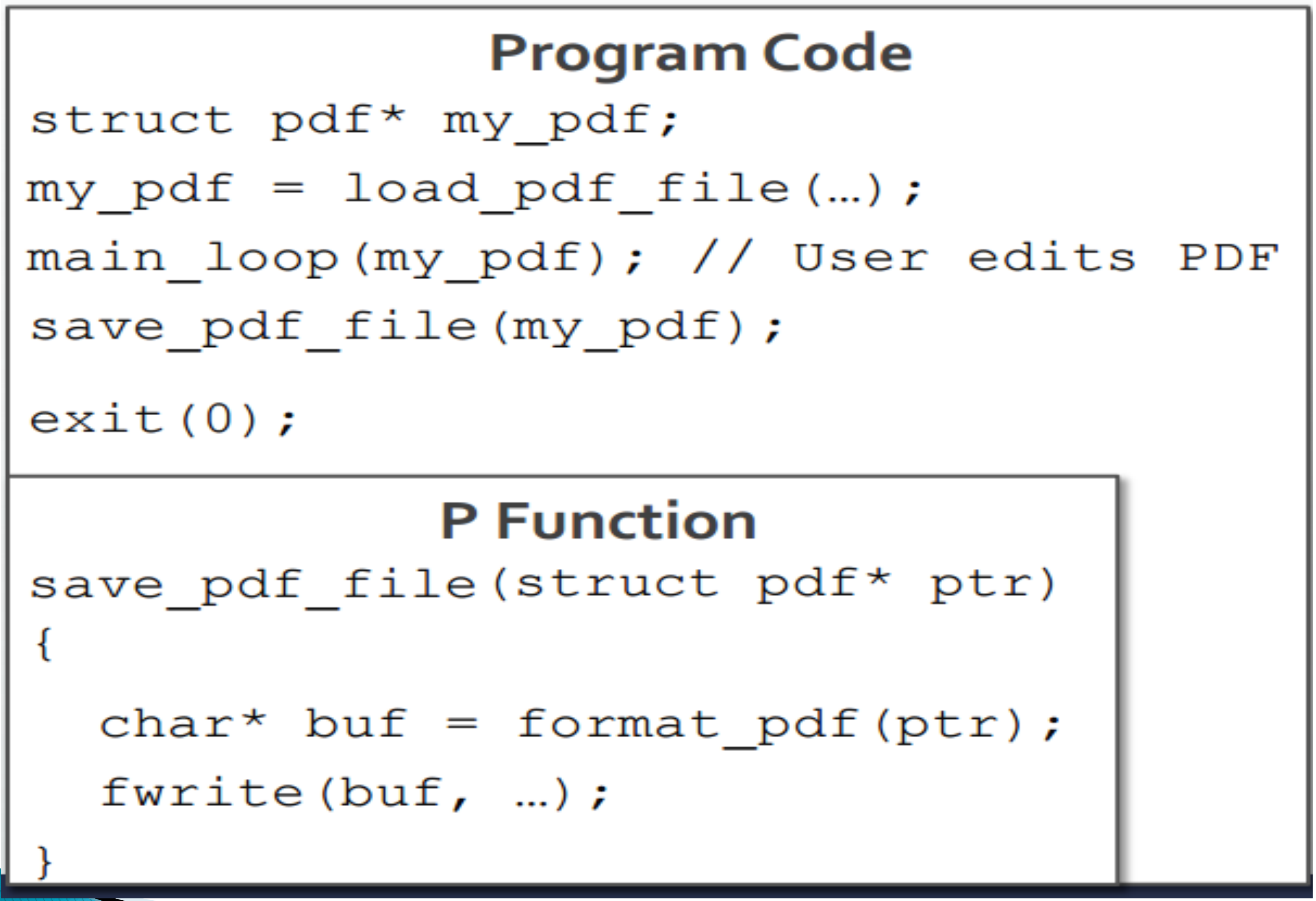


Content Reverse Engineering

•Application that defined the data structure contains printing/ rendering logic for it too.

•Let’s call this function as P

•The P function should take asinput the raw in memory data structure format it or process it to a human readable understandable PDF file



Program Code

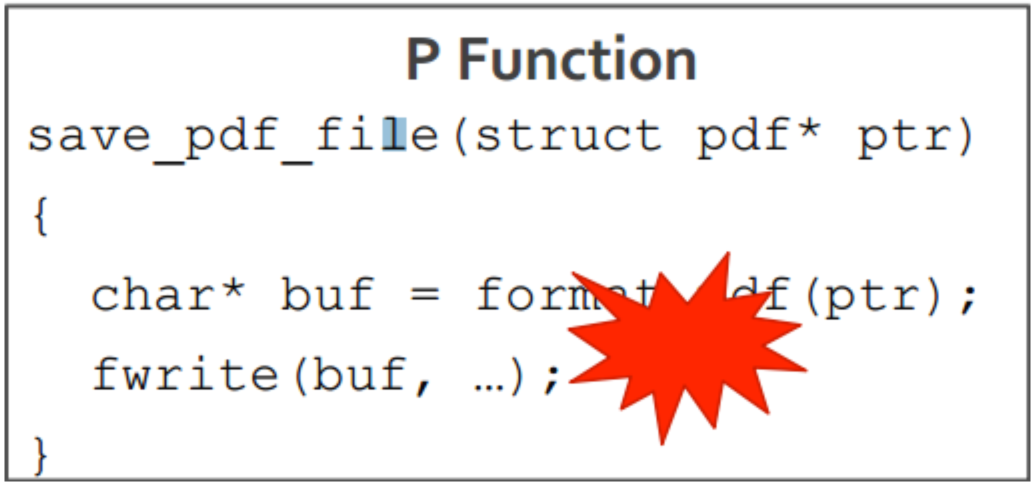
```
struct pdf* my_pdf;
my_pdf = load_pdf_file(...);
main_loop(my_pdf); // User edits PDF
save_pdf_file(my_pdf);

exit(0);
```

P Function

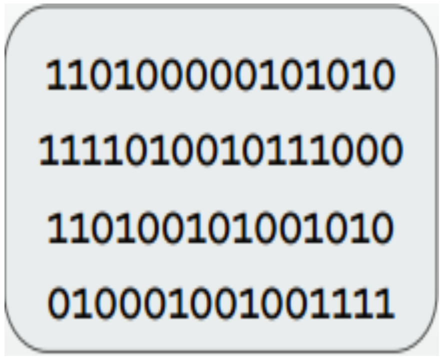
```
save_pdf_file(struct pdf* ptr)
{

  char* buf = format_pdf(ptr);
  fwrite(buf, ...);
}
```


P Function

```
save_pdf_file(struct pdf* ptr)
{
   char* buf = form     df(ptr);
   fwrite(buf, …);
}
```

- ➢DSCRETE reuse P to build reusing the existing data structure interpretation and binary a scanner+renderer tool.



- ➢Invalid input will mainly crash the function P.

DSCRETE Binary to Scanner Renderer

- The investigators recover the binary from the suspects computer .

- DSCRETE then builds a scanner+renderer tool in 2 steps.

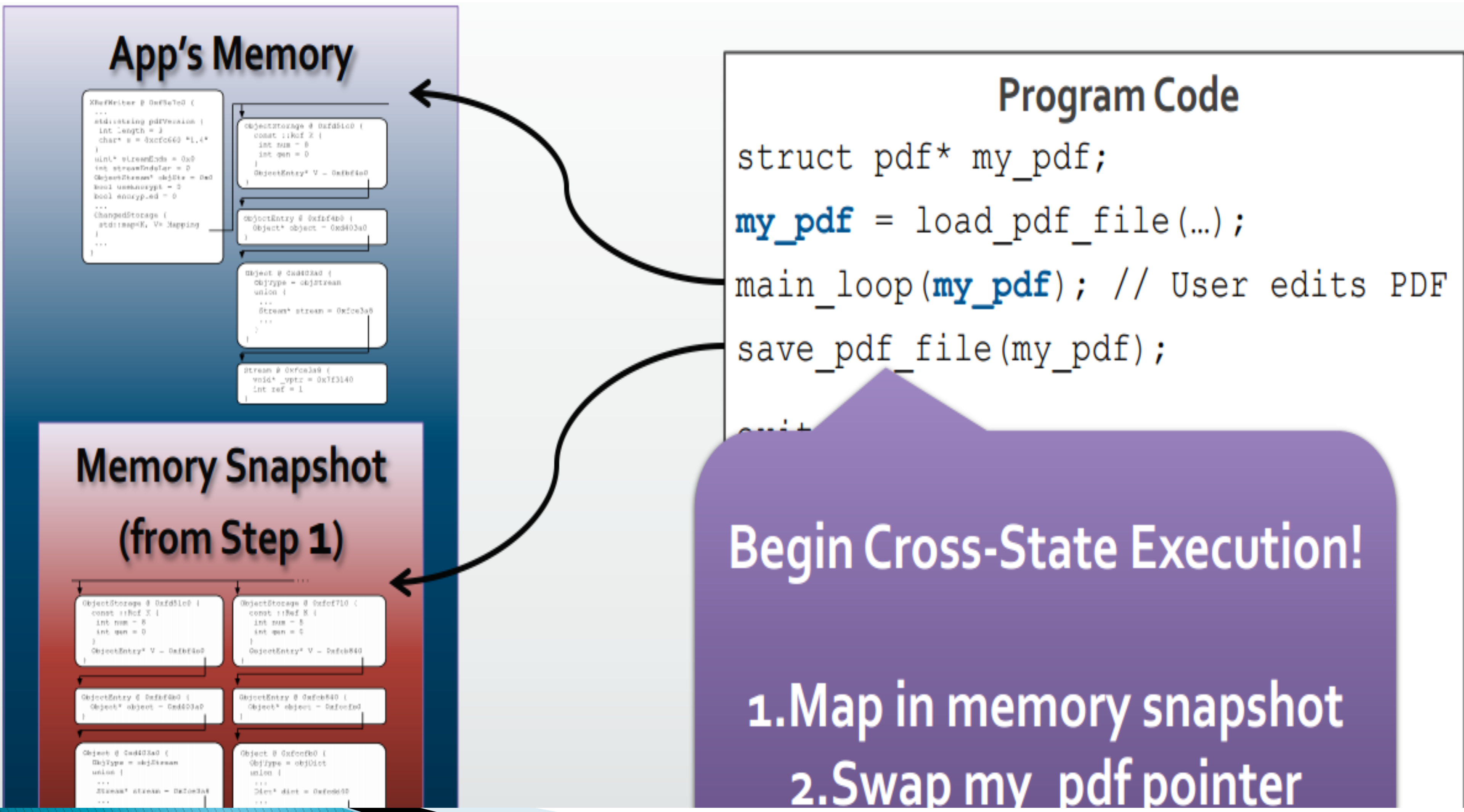
- Thus the tool can be reused in all future investigations of that application

Identifying Functional Closure

- It mainly execute the binary from the suspect's computer .
- The slicing techniques find printing/rendering component.
- Select all the output functions that emit evidence.
- DSCRETE saves a memory snapshot during output function

Finding the Scanner’s Entry Point

- DSCRETE finds candidates for the entry point.
- Candidates must take a heap pointer as input.
- All these selected output/rendering functions must depend on it.
- It mainly uses the technique of Cross state execution to find the correct candidates.



App's Memory

```
XRefWriter @ 0xf5e7c0 {
 ...
 std::string pdfVersion {
  int length = 3
  char* s = 0xcfc660 "1.4"
 }
 uint* streamEnds = 0x0
 int streamEndsLen = 0
 ObjectStream* objStr = 0x0
 bool useEncrypt = 0
 bool encrypted = 0
 ...
 ChangedStorage {
  std::map<K, V> Mapping
 }
 ...
}
```

```
ObjectStorage @ 0xfd51c0 {
  const ::Ref X {
   int num = 8
   int gen = 0
  }
  ObjectEntry* V = 0xfbf4a0
}
```

```
ObjectEntry @ 0xfbf4b0 {
  Object* object = 0xd403a0
}
```

```
Object @ 0xd403a0 {
  ObjType = objStream
  union {
   ...
   Stream* stream = 0xfce3a8
   ...
  }
}
```

```
Stream @ 0xfce3a8 {
  void* _vptr = 0x7f3140
  int ref = 1
}
```

Memory Snapshot (from Step 1)

```
ObjectStorage @ 0xfd51c0 {
  const ::Ref X {
   int num = 8
   int gen = 0
  }
  ObjectEntry* V = 0xfbf4a0
}
```

```
ObjectStorage @ 0xfcf710 {
  const ::Ref X {
   int num = 5
   int gen = 0
  }
  ObjectEntry* V = 0xfcb840
}
```

```
ObjectEntry @ 0xfbf4b0 {
  Object* object = 0xd403a0
}
```

```
ObjectEntry @ 0xfcb840 {
  Object* object = 0xfccfb0
}
```

```
Object @ 0xd403a0 {
  ObjType = objStream
  union {
   ...
   Stream* stream = 0xfce3a8
   ...
```

```
Object @ 0xfccfb0 {
  ObjType = objDict
  union {
   ...
   Dict* dict = 0xfcdd40
   ...
```

Program Code

```
struct pdf* my_pdf;
my_pdf = load_pdf_file(…);
main_loop(my_pdf); // User edits PDF
save_pdf_file(my_pdf);

exit
```

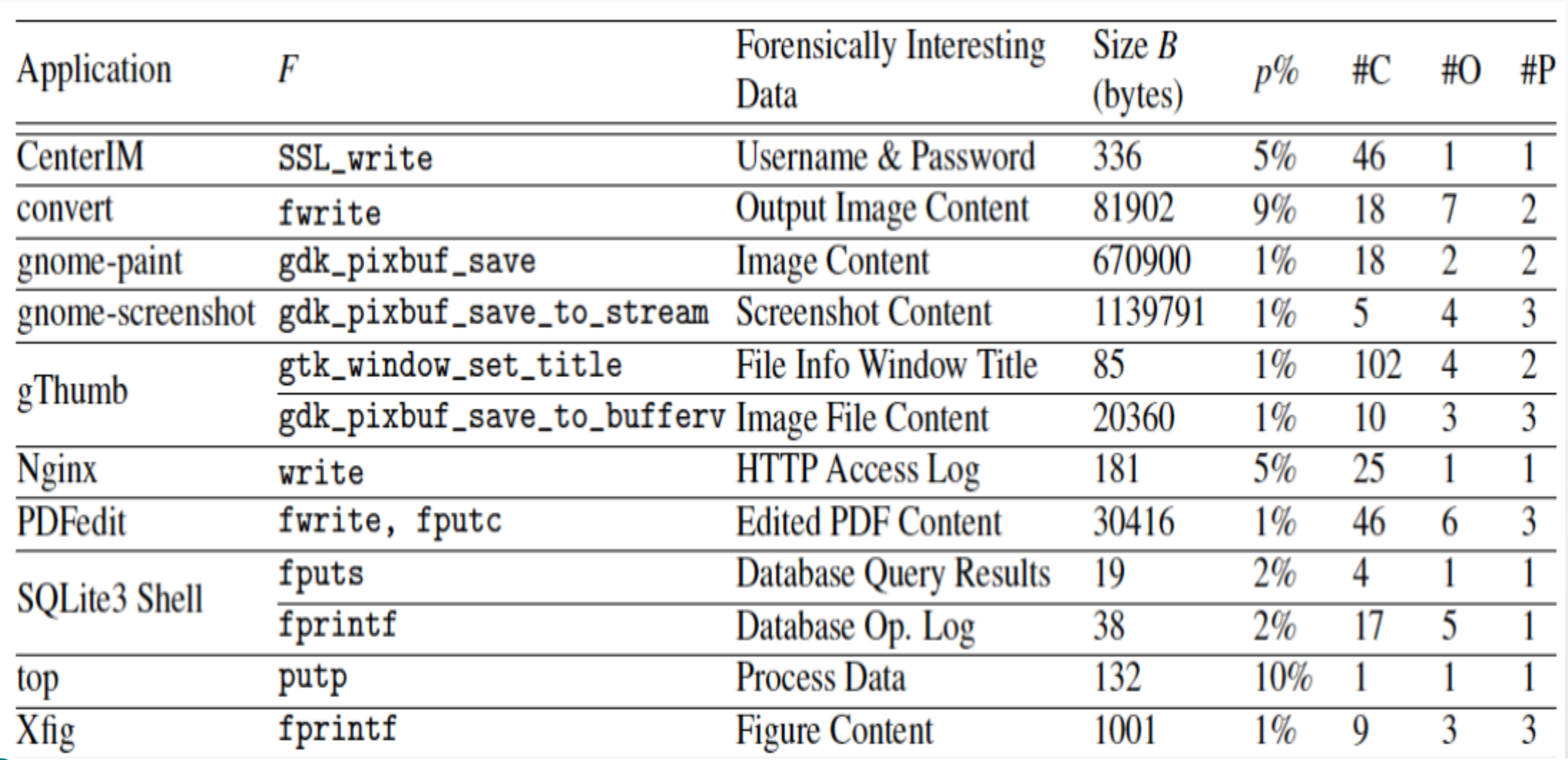
Begin Cross-State Execution!

1. Map in memory snapshot
2. Swap my_pdf pointer

Memory Scanner Effectiveness

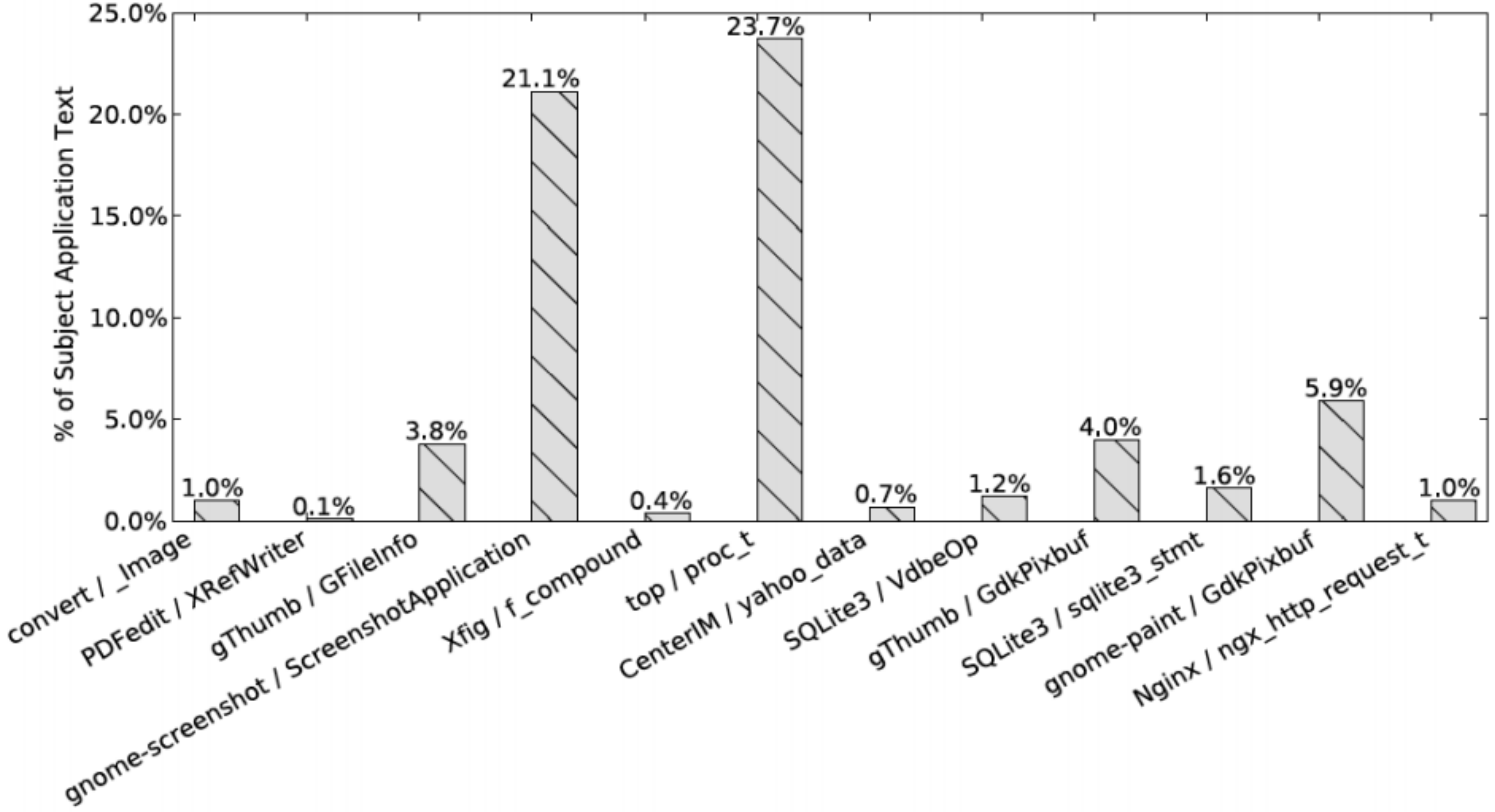
- A correct candidate will output the PDF.
- It mainly presents each offset in suspect’s memory image to P and reports natural application output as evidence.
- This tool can be used in all future investigations.

P function identification effectiveness



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

Normalized size of P vs. entire binary code



Conclusion

- This has identified the main problem content Reverse Engineering problem in forensics.
- DSCRETE leverages binary logic reuse toautomatically locate data structures in memory images and reverse engineer content
- They are highly effective in recovering many forms of digital evidence

DSCRETE: Automatic Rendering of Forensic Information from Memory Images via Application Logic Reuse.

Brendan Saltaformaggio, Zhongshu Gu, Xiangyu Zhang, and Dongyan Xu. In UsenixSecurity'14

Paper Discussion

- Zhenyu Ning
- CSC 6991 – Advanced Computer System Security

- In contrast with the state-of-the-art memory forensics, this paper presents a new approach to achieve memory forensics without reverse engineering. The most amazing part of the new system, DSCRETE, is that it output the display of the target data structure instead of just raw bytes of it.

- To achieve this, DSCRETE try to run the target binary application in the same environment with the target machine at the very beginning and generate a memory image, together with an instruction record, after creating enough target data structure and outputting the data structure. Then through some static analysis mechanism, it found some candidates of closure points, which may be the beginning of editing a target data structure. After that, the binary application is re-executed. When the execution reaches a candidate, a sub process is forked and pointer to the target data structure is then modified to point to some old data which is mapped from the memory image generated in the first execution. With the result of execution after modify the pointer, DSCRETE then briefly judge whether a candidate is a real closure points. After it gets some real closure points, the binary application is executed for the third time in which closure points and sub processes are used to find all potential target data structures in the memory dump and also show the display of the data structure directly to investigator.

- The evaluation shows that DSCRETE can show images, pdfs, files and some other complex data structures effectively, but has a bad performance when facing some trivial data structure. It is a pity that DSCRETE is not applicable to applications written in interpreted language like Java. But notice that we can reverse Java application much easily than application written in other language. If mechanism of DSCRETE can be used to Java by leverage reverse engineering, I guess it is also a good way to analysis memory in Android application.

Paper Discussion

- Lucas Copi
- CSC 6991
- 14 October 2015
- Memory Forensics
- The paper *DSCRETE: Automatic Rendering of Forensic Information from Memory Images via Application Logic Reuse* discusses a new method for forensically retrieving files from a from a systems' memory image using DSCRETE. Traditional forensics utilizes signature based scanning to uncover data structures in memory. However, many data objects in memory include application specific encoding, making it difficult for investigators to render the data in a meaningful way. The DSCRETE system both interprets and renders data structures found in memory to present the data in a human readable format.
- DSCRETE is based on the assumption that data structures are stored with rendering logic in the original application binary. This assumption allows DSCRETE to isolate data structure printing functionality in the application binary. This process requires tracing the subject applications dynamic data dependences and locating the closure point for the rendering function. Once the data structure rendering function has been fully identified, DSCRETE can build a scanning+rendering tool from the subject binary.
- DSCRETE was implemented and tested against a Ubuntu desktop 'suspect' machine. In the case studies, DSCRETE performed at expectations as was able to uncover and render valid data structure instances with 100% accuracy for most cases. Additionally, DSCRETE was able to represent several key types of evidence that would be nearly impossible to reconstruct with traditional memory forensic systems.

Paper Discussion

- Hitakshi Annayya

- In old days memory forensics used to investigating by signature based scanning of memory images to uncover data structure -- Reverse Engineering. The disadvantage of this method is not be able to interpret the content of data structure fields. The paper presents new method called DSCRETE data structure content reverse engineering technique, which is a system that enables automatic interpretation and rendering of inmemory data structure contents. 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 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.

- Assumptions made for DSCRETE workflow: first - DSCRETEbased memory the subject binary can be executed. Second - the OS kernel's paging data structures in the subject memory image are intact. Many phases completes the design of DSCRETE- Dynamic data dependency tracing (a data dependence graph is generated using the trace gathered during dynamic instrumentation.), next identifying functional closure, to find scanners entry point, and finally memory image scanning.

Reminders

- Next class: Android Security
- Proposal revision
- Paper summary is required when presenting