AmpleDroid Recovering Large Object Files from Android Application Memory

Sneha Sudhakaran
Louisiana State University
ssudha1@lsu.edu

Aisha Ali-Gombe
Towson University
aaligombe@towson.edu

Augustine Orgah
Louisiana State University
aorgah1@lsu.edu

Andrew Case
Volatility Foundation
andrew@dfir.org

Golden G. Richard III
Louisiana State University
golden@cct.lsu.edu

[1] ***Abstract*—Analysis of app-specific behavior has become an increasingly important capability in the fields of digital forensics and incident response. The ability to determine the precise actions performed by a user, such as URLs visited, files downloaded, messages sent and received, images and video viewed, and personal files accessed can be the difference between a successful analysis and one that fails to meet its goals. Unfortunately, proper analysis of volatile app-specific evidence, especially the recovery of large objects such as multimedia and large text files stored in memory has not been explored. This is mainly because the allocation function in the various Android memory management algorithms handles large objects differently and in separate memory regions than small objects. Thus, in this paper our effort is focused on developing an app-agnostic memory analysis tool capable of recovering and reconstructing large objects from process memory captures. We present *AmpleDroid*, a tool that identifies and extracts large objects loaded in an application memory space. Our methodology involves the inspection of the process image to identify vital Android runtime data structures utilized during large object allocation. *AmpleDroid* is evaluated on a number of apps and the results shows the recovery of almost 91% of the allocated large objects from process memory.**

***Index Terms*—Android Runtime, Large Object, Multimedia, app memory**

I. Introduction

Android is the most popular mobile operating system with a market share of 73.3% [1]. People use a wide variety of mobile or personal digital assistant (PDA) applications (app) for day-to-day activities, which involve the processing, sharing and manipulation of both textual data and multimedia objects like photos, video, and audio [2]. All these activities leave behind evidence in the device memory. Extraction of such evidence from memory requires a unique approach called memory analysis, a vital technique employed by practitioners for the identification and extraction of evidence from Android devices [3]. Recent research efforts, such as [4]–[6], target the recovery of forensic evidence from the new Android Runtime (ART). Soares et al described a technique to analyze objects in memory within the ART execution environment using volatile memory [7] data extraction [5]. Soares et al performed an in-depth study of the runtime and built extensions for the Volatility framework [8] [9] for devices compliant with the ARM (Advanced RISC Machine) architecture [5]. Timeliner is a forensics technique capable of automatically extracting the timeline of user actions from a single memory image acquired from an Android device for all the installed apps [6]. The forensic technique in Timeliner is designed to leverage the memory image of Android's centralized ActivityManager service [6]. DroidScraper is another app analysis tool for Android version 8 that recovers small objects allocated in a process memory image using RegionSpace memory management. RegionSpace is a memory management space in Android that consists of equal-sized regions [4]. DroidScraper analyzes these regions and recovers the allocated small objects like int, bool etc. [4]. However, all of this work has limitations that prevent recovery of large size multimedia objects like images, audio, video and large textual data. Hence, we propose a tool called *AmpleDroid* that can recover large objects allocated in the Large Object Space (LOS). LOS is a region of memory in the new ART reserved for objects larger than a particular size [10] [11]. *AmpleDroid* embodies a new memory analysis technique that analyzes large object memory allocations to recover large text files and multimedia artifacts from process memory dumps. In experimental studies, the tool exhibits high accuracy ($\approx 91\%$) for large objects recovery and decoding.

A. Contribution

- The tool *AmpleDroid* can be used to analyze process memory to identify and extract allocated large objects like multimedia.
- The tool is evaluated on multiple applications to successfully recover 91% of the allocated large objects.

The rest of the paper is organized as follows: Section 2 presents the background of this paper; Section 3 provides an overview of our design and implementation; Section 4 presents the evaluation of the proposed approach; Section 5 presents summary of related work; Section 6 summarizes our findings and conclusions.

[1]This material is based upon work supported by the National Science Foundation under Grant 1703683

WIFS'2020, December, 6-11, 2020, New York, USA. 978-1-7281-9930-6/20/$31.00

```
class AllocationInfo;

enum class LargeObjectSpaceType {
  kDisabled,
  kMap,
  kFreeList,
};
```

Fig. 1. Source Code Listing for the Type of Large Object Space

II. Background

Memory analysis can provide unique insights into runtime system activity, including information about and data from recently executed applications. A forensic investigator requires OS-specific mechanisms to identify and extract allocated objects from memory. In the Android OS, identification of allocated objects from LOS in the heap is essential for evidence extraction, as this region of memory is specifically reserved for allocating large objects such as images, text, audio and video. The ART organizes virtual memory of an app into several categories including heap space (for Java objects), zygote shared object space, and space for large objects [4]. RegionSpace memory allocation is the overall mechanism used in ART to allocate both small and large objects in regions of memory [4]. This memory management algorithm allocates objects at the top of the region, and a new top is determined by adding the size of the object to the top address. The RegionSpace algorithm has two allocation functions that determine how objects are laid out in memory. The Alloc() function is used to allocate small objects such as small primitives, strings, arrays, and other complex objects such as InetAddress. On the other hand, the AllocLarge() function handles the allocation of objects that are above a certain threshold of 12KB. At the start of a process runtime, a heap region is created, which in turn creates multiple regions of equal size, some of which are reserved as LOS and are used by the AllocLarge() function to allocate large objects. Typically, the LOS is located in a unique region in the process memory called Dalvik Large Object Allocation [12]. In ART, LOS uses discontinuous memory mapping, where object allocation regions are not contiguous [13]. Primarily objects stored in the LOS are allocated in arrays of types such as byte, char, string, float, and int [12]. ART categorizes the LOS into three types: FreeList, Map, and Disabled as shown in Figure.1 [12].

During heap initialization, the system creates a LOS structure based on whether the runtime option for LOS is type Freelist or MAP. A Freelist LOS creates a continuous space called the FreeListSpace, which is designed to hold free blocks and handle holes. A MAP LOS creates a discontinuous space called the LargeObjectMapSpace [13], a shared memory region that allocates objects using memory maps instead of the typical Linux malloc. During a large object allocation, the allocator inspects the availability of the size required for the new large object in memory [12] and then creates a memory mapping matching that size. The beginning offset of the mapping is then chained to the allocation tracking map of the LargeObjectMapSpace. Similarly, after the freeing

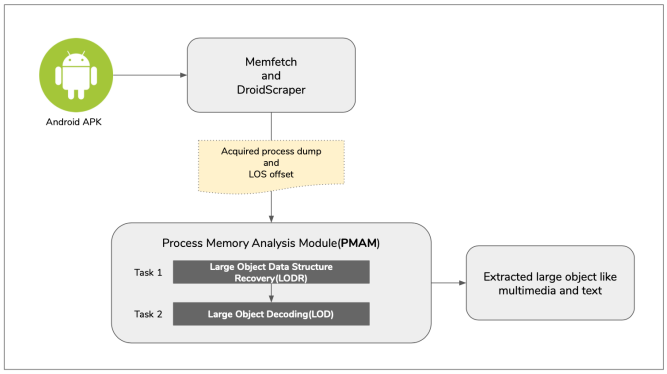

Fig. 2. AmpleDroid's Component Workflow

of a large object, the blob of memory gets evacuated to a unique region called the evacuation region, where it remains until the next garbage collection(GC) cycle [11]. Objects that are moved to the evacuation region are not recovered from the LOS region, hence, the retrieval of such objects are not within the scope *AmpleDroid*. The Android Open Source Project(AOSP) source code was analyzed to identify that large objects like images, video, and text files are allocated in LargeObjectMapSpace, thus, *AmpleDroid* focuses on LargeObjectMapSpace. The structure and class definition of the LargeObjectSpace and LargeObjectMapSpace are explained in detail in the system design section. Apart from FreeListSpace and LargeObjectMapSpace, there is another LOS category called Disabled that allocates runtime-specific large objects that are never garbage collected.

III. System Design

AmpleDroid is a large object evidence recovery and decoding tool that analyzes an app's memory capture to extract large objects. The tool inspects various data structure definitions and class templates to retrieve and recreate large objects allocated during runtime. Recovery of large objects can be performed for either textual data or multimedia. As shown in Figure.2, *AmpleDroid*'s workflow consists of - The Process Memory Analysis Module (PMAM), which includes two major tasks: Task1: Large Object Data Structure Recovery and Task2: Large Object Decoding to recover and decode the large objects stored in process memory.

Workflow of *AmpleDroid* begins with the extraction of the process memory image and the recovery of its main runtime structures. The design of our approach is generic in that the system can utilize any of the available process memory imaging techniques/tools [14], [15]. For illustrative purposes, we used Memfetch to extract the memory images of targeted processes [14]. Memfetch utilizes the /proc filesystem to dump available memory maps from a running app. The memory of the targeted userspace process is dumped into a directory containing individual mapping and allocation files. The mem*.bin files are anonymous memory blocks like the stack and the heap. The map*.bin files are a copy of all maps, including the shared libraries and large object space

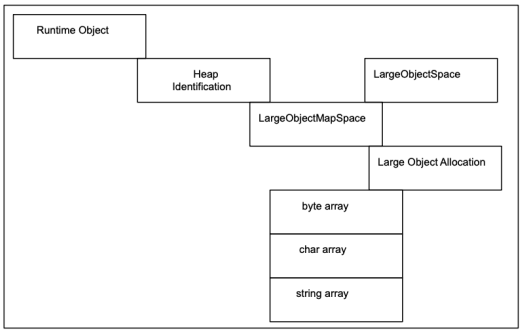

Fig. 3. Process Memory Analysis Module - PMAM

allocations. Map files acquired using Memfetch [14] are then decoded with *AmpleDroid* to extract the large object data. The acquired process dump is then analyzed to identify the ART data structures using the tool DroidScraper [4]. The identified data structures constitute the main runtime's object, process heap, threads, and stacks that are crucial during app execution and therefore vital to understand large object allocations. The *heap* plugin in DroidScraper is used in *AmpleDroid* to identify the LOS offset and its attributes, illustrated in Figure.4. The app dump along with the ART data structures (used to recover large objects) are passed to the PMAM for further inspection.

A. Process Memory Analysis Module - PMAM

The PMAM module analyzes the output acquired from the process memory extraction tool Memfetch to recover raw data for every large object allocated in the process memory space during runtime [14]. As shown in Figure. 3, our analysis begins with the inspection of the recovered heap structure to find the LargeObjectMapSpace - which is implemented using the Linux mmap() function [13]. In ART, the class LargeObjectMapSpace is derived from LargeObjectSpace and thus inherits all of its fields. As shown in Figure.4, the LargeObjectSpace in Android version 8 contains the following attributes: num_bytes_allocated at offset 32; num_objects_allocated at offset 40; total_bytes_allocated at offset 48; total_objects_allocated at offset 56; begin_ at offset 64, and end_ at offset 68 [13].

The LargeObjectMapSpace further contains attributes of both LargeObject (at location 112) and the AllocationTrackingSafeMap (at location 120) structures as shown in Figure.4. The LargeObject structure contains attributes like MemMap (a pointer to the large object allocated) and is_zygote (a boolean to indicate if the allocated large object is a zygote or not). The AllocationTrackingSafeMap structure inherits all the attributes and functions from SafeMap [16]. The SafeMap allocation is used after determining if the pointer acquired from the MemMap attribute is a new pointer or a stored template value (a pointer of LargeObject - MemMap stored initially in the memory). The two structures hold the corresponding pointer to the memory map and the metadata structure for allocation tracking [16]. The combination of members of all these structures define how the Android memory allocator lays out every object larger than 12KB in memory.

```
        'LargeObjectSpace' : [ 0x48, {
                'Discontinuous_Space': [0x1c,{
                'Space':[0,['']],
                'live_bitmap_':[20,['']],
                'mark_bitmap_':[24,['']],
                }],
                'AllocSpace' : [0x4, {
                '_vtr$AllocSpace':[28,['']],
                }] ,
                'num_bytes_allocated_':[32,['']],
                'num_objects_allocated_' :[40,['']],
                'total_bytes_allocated_' :[48,['']],
                'total_objects_allocated_': [56,['']],
                'begin_' :[64,['']],
                'end_' :[68,['']],
    }],
        'LargeObjectMapSpace':[0x80,{
                'Inheritance':[0, ['LargeObjectSpace']],
                '_lock':[72,['']],
                'LargeObject': [112, ['LargeObject']],
                'AllocationTrackingSafeMap':[120,['']],
        }],
        'LargeObject':[0x8,{
                'MemMap':[0, ['Pointer']],
                'is_Zygote':[4,['Boolean']],
        }],

}
```

Fig. 4. The 'Large Object Space' object for Android 8, listing as a C structure

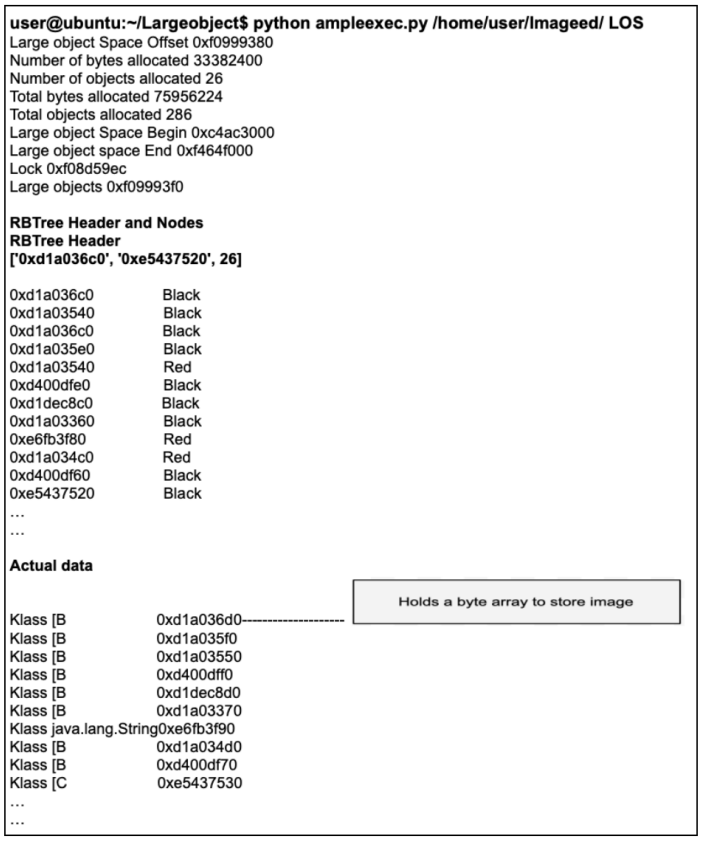

Fig. 5. *AmpleDroid* LOS plugin recovering objects with memory address

The design of PMAM module involves two tasks as shown in Figure.2:

- The Large Object Data Structure Recovery (LODR)
- The Large Object Decoding (LOD)

1) The Large Object Data Structure Recovery (LODR): The LODR extracts the blob of bytes for each object allocated in the LargeObjectSpace. Examining the AOSP source code for Android version 8, we have identified that the ART allocates and initializes storage for an object instance using the AllocObject() function, which in turn calls AllocObjectWithAllocator() [17]. For large objects the AllocObjectWithAllocator verifies that the template value parameter kCheckLargeObject is true and the byte count for the object is greater than or equal to the large object threshold (greater than or equal to 12KB). If these two requirements are met then the heap defaults the allocator type to kAllocatorTypeLOS and allocates the object using a generic AllocLarge(). If for any reason one of the parameters is false or the large object allocation fails, then the heap will default to one of the small object allocators. The large object allocation in LargeObjectSpace created by the function AllocLarge() is based on an attribute large_object_space_type to finalize if the LOS allocated is a LargeObjectMapSpace or not [18] [19].

The SafeMap allocation, identified with the MemMap pointer as mentioned above, was studied in detail using AOSP source code analysis [16]. The SafeMap allocation performs binary searches using a red-black tree data structure by comparing the address offset of LargeObject and loads the metadata of large objects effectively [16] [20]. The red-black tree data structure performs binary search insertions and deletions for faster and best fit memory allocation [20]. Therefore, each node holds the large object metadata extracted using the *AmpleDroid* RBtree Algorithm. The RBtree algorithm identifies the addresses of all the nodes that allocate large objects by recovering the NodeDetails. NodeDetails include each node's color and the consecutive node (the next node used to recover the actual large object data). The results of the

Algorithm 1: RBtree Algorithm

```
Parameters: LOS_offset offset
 if offset != '0x0' then
 |  LargeObjectMapSpace LOS = GetNode(offset)
 |  nodeList = []
 |  nodeList.append(LOS.begin_)
else
 |  error: "Not allocated to LOS"
end
for int i=0; i ≤ len(nodeList); i++ do
 |  nodeList = TRAVERSAL(node[i],nodeList)
end
function TRAVERSAL(node, nodeList)
   (parent, left, right, color) = NodeDetails(node)
   nodeList.append(left)
   nodeList.append(right)
   return nodeList
end function
```

plugin to recover attributes of LOS allocation from a sample process dump (Image Editor in this evaluation) are shown in Figure.5. The LODR inspects the process memory image to recover the LargeObjectSpace offset at address 0xf0999380. The AllocationTrackingSafeMap attribute which a safe map that holds all the LargeObject structures corresponding to each large object allocation is also extracted and displayed in Figure.5. The LOS begin address is at memory location 0xc4ac3000, which is the offset where the actual large object analysis starts. The pointer LargeObject in Figure.5at offset 0xf09993f0 is also at analyzed, as it points to the map that holds the the structure of the large object. The RBTree Header in Figure.5 depicts the offsets that point to the MemMap structure that in turn hold large object offsets.

2) Large Object Decoding (LOD): LOD is the second task performed by the PMAM. LODR inspects the process dumps to identify the metadata of each large object which can be decoded here to extract the original file loaded in the process memory. If the size of the object loaded in memory exceeds the attribute value kMinLargeObjectThreshold (three times the page size is 3*4096 = 12KB) [17] then these objects are allocated in LargeObjectSpace as large primitive arrays [17]. Such large primitive arrays include byte array, char array, float array, int array and string array that can be further decoded to recover the actual file loaded in the memory. The actual file recovered on decoding will be a large text file or a multimedia file. The large primitive arrays recovered are explained below:

1) byte array - The byte arrays are optimized data storage mechanisms for storing large multimedia files [21].
2) char array - The char arrays are used to store arrays of characters from large input files like text files [22].
3) string array - The string arrays holds the resources that are referenced from the application [23]. Resources in Android provides a format for storing text strings for applications with optional text styling and formatting.
4) float array, int array - These primitive arrays represent properties like screen pixels, but these are currently ignored as they are outside our current research focus.

To summarize, *AmpleDroid* with LODR analyzes the AllocationTrackingSafeMap and identified offsets of the large object structures that are represented as RBTree offsets. The RBTree offset on further analysis with LOD identified the data structure that allocates the large object stored in the memory. For example in Figure.5 the RBTree offset 0xd1a036c0 is a black node that on analysis with LODR and LOD stores a byte array at offset 0xd1a036d0.

IV. Evaluation

AmpleDroid is a tool developed to analyze app dumps for retrieving large object files like multimedia and text as shown in Figure.6. This tool is written in Python to perform an in-depth process memory analysis for Android version 8. The free and open source code of this tool will be released with the publication of the paper.

To assess the effectiveness of object recovery of *AmpleDroid*, a series of evaluations were performed on process memory images. The evaluations were conducted on five benign apps: Gallery Vault, Instagram, Image Editor, PDF Reader, and Facebook Lite from the Google Play store [24].

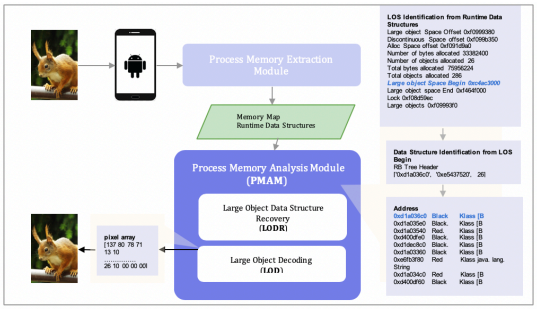

Fig. 6. Flow of *AmpleDroid* describing Recovery and extraction of large files.

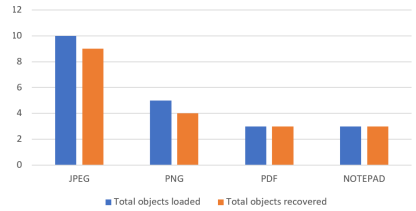

Fig. 7. The statistical view of loaded files in comparison with the recovered files using *AmpleDroid* on Image Editor App.

A. Experimental Setup

Our experimental setup used the genymotion Android emulator as the execution environment [25]. We created Android Virtual Devices (AVD's) for the Samsung S8 emulator running Android 8.0-API 26 and for the Google Nexus 6 running Android 8.0-API 26. All the emulators had 4GB memory and apps were installed and loaded with multiple images, text files, to simulate real devices. We interacted with the app manually to generate sufficient activities for evaluation. Using Memfetch, we captured the process memory of the app [14] and extracted the runtime data structures with DroidScraper [4]. Finally, retrieval of the large objects was accomplished with *AmpleDroid*.

B. Object Recovery and Decoding

We tested *AmpleDroid* on apps from various categories, as mentioned earlier. Recovered objects were grouped into data structures: byte arrays, char arrays and string arrays, as these store images, video frames, text, PDF data, resource data referenced from applications, etc. The performance of *AmpleDroid* is illustrated with Table 1 depicting the recovery percentage. The recovery percentage is the ratio of recovered loaded files to loaded large object files.

For each evaluated app, the recovery percentage is displayed in the % column. *AmpleDroid* can recover an average of 91% of all allocated large objects within the process memory which shows that the tool is useful for Android 8 memory analysis. On further analysis, *AmpleDroid* did not recover the 9% allocated objects due to the acquired process dumps. The significant reason behind the objects not retrieved by the tool was the impact of critical runtime external factors like Garbage collection on the dumps.

Fig. 8. The frames of video viewed by the user

1) Editing apps object recovery: In Figure.6, the emulator was used to install the Image Editor app and a PNG image was loaded in the app [25]. After a few minutes, Memfetch acquired Image Editor's process dump [14]. By executing DroidScraper's Heap plugin, we identified the LOS offset. Next, after executing *AmpleDroid*'s *LOS* plugin, we extracted the RBtree nodes to identify the large object metadata. The metadata upon inspection and analysis with the *DecodeLosObject* plugin generated a byte array which on further decoding retrieved the PNG image loaded earlier.

We also evaluated the tool by loading 21 files: 10 JPEGs, 5 PNGs, 3 PDFs, and 3 Notepad documents into the Image Editor app. The files recovered from memory were byte arrays (images), char arrays (input text files), and string arrays (files referenced from application). In Figure.7, the graph illustrates the efficiency of the *AmpleDroid* tool by identifying and recovering the large object files in comparison to the total files loaded manually. The graph shows at least 91% recovery and decoding efficiency.

2) Social media apps object recovery: The Android emulator was used to install the Facebook Lite app. We viewed a 1 min video, loaded images to a Facebook account and posted a story. After a few minutes, Memfetch acquired the process dump of the Facebook app (with the metadata of the video viewed, story posted in memory) [14]. The metadata on inspection with *AmpleDroid* displayed traces of webp and webm files that on decoding generated a series of byte arrays loaded in the memory. On further decoding, we could extract the frames of the video viewed. The results of the extracted video frames are shown in Figure.8.

V. Related Literature

Extraction of all types of large objects (text, image, audio, video) from smartphone memory is a significant contribution to Android memory forensics. This is difficult due to the variations made in the Android smartphone [26]. There are many works associated with the recovery of forensic data from memory images of which related work like Guitar [27], Timeliner [6], Discrete [28], VCR [26] and DroidScraper [4] are more useful in this work. The Visual Content Recognition (VCR) tool is one such tool that recovers large object data by unveiling an intermediate service used by Android device

Apps	Large Objects Loaded	String Array	Char Array	Byte, Int and Float Array	Large Object Recovered	%
Gallery Vault	82	15	35	26	76	93
Instagram	33	8	16	6	30	91
Image Editor	21	3	7	11	19	91
PDF Reader	30	6	11	10	27	90
Facebook Lite	72	17	24	25	66	92

TABLE I
EVALUATION RESULTS

cameras to recover visual content images [26]. Saltaformaggio et al, in their work, VCR, extracted multimedia contents from different apps that used the Android device camera. VCR does not have access to such runtime information and operates on only the input static memory image [26]. But, *AmpleDroid* is different from VCR as our tool recovers all the multimedia and text files stored in process memory without the involvement of any standard service like the camera. Also, Ali-Gombe et al proposed DroidScraper, a tool for analyzing the ART RegionSpace memory allocation to extract small objects [4]. This tool was used to recover runtime data structures with special attention to the recovery of small objects allocated in the RegionSpace [4]. The class of data recovered by *AmpleDroid* is completely different, and the large object data recovered by our tool could contain critical digital evidence to support legal proceedings or cyber investigations.

VI. CONCLUSION

In this paper, the tool presented is used to analyze an Android app dump to extract the large objects allocated in the LargeObjectSpace. The LargeObjectSpace allocation is studied in detail to identify how the large objects are stored in memory. *AmpleDroid* is tested on benign apps with at least 91% accuracy. This tool supersedes other Android tools because *AmpleDroid* performs a complete process memory analysis on Android version 8 large object memory allocation and extracts multimedia and text files which other tools cannot currently process. Currently, *AmpleDroid* is fully automated and will be made open source when the paper is published. This tool can be instrumental in forensic investigations as it provides an overall idea of how large object files (text, video, image, etc.) are mapped in an app's memory along with high recovery and decoding rate.

REFERENCES

[1] S. Türker and A. B. Can, "Andmfc: Android malware family classification framework," in *2019 IEEE 30th International Symposium on Personal, Indoor and Mobile Radio Communications (PIMRC Workshops)*. IEEE, 2019, pp. 1–6.

[2] M. Sun, T. Wei, and J. C. Lui, "Taintart: A practical multi-level information-flow tracking system for android runtime," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016, pp. 331–342.

[3] A. Case, "Memory Analysis of the Dalvik (Android) Virtual Machine," *Source Seattle*, 2011.

[4] A. Ali-Gombe, S. Sudhakaran, A. Case, and G. G. Richard III, "Droidscraper: A tool for android in-memory object recovery and reconstruction," in *22nd International Symposium on Research in Attacks, Intrusions and Defenses ({RAID} 2019)*, 2019, pp. 547–559.

[5] A. M. M. Soares and R. T. de Sousa Jr, "A technique for extraction and analysis of application heap objects within android runtime (art)." in *ICISSP*, 2017, pp. 147–156.

[6] R. Bhatia, B. Saltaformaggio, S. J. Yang, A. I. Ali-Gombe, X. Zhang, D. Xu, and G. G. Richard III, "Tipped off by your memory allocator: Device-wide user activity sequencing from android memory images." in *NDSS*, 2018.

[7] D. Apostolopoulos, G. Marinakis, C. Ntantogian, and C. Xenakis, "Discovering authentication credentials in volatile memory of android mobile devices," in *Conference on e-Business, e-Services and e-Society*. Springer, 2013, pp. 178–185.

[8] M. Auty, A. Case, M. Cohen, B. Dolan-Gavitt, M. H. Ligh, J. Levy, and A. Walters, "Volatility-an advanced memory forensics framework."

[9] H. Macht, "Live Memory Forensics on Android with Volatility," *Friedrich-Alexander University Erlangen-Nuremberg*, 2013.

[10] M. Chartier, "Performance and memory improvements in android run time (art)," in *Google I/O '17*, 2017.

[11] R. Jones, A. Hosking, and E. Moss, *The garbage collection handbook: the art of automatic memory management*. CRC Press, 2016.

[12] AndroidXRef, "Androidxref oreo 8.0.0_r4," http://androidxref.com/8.0.0_r4/xref/art, 2018.

[13] A. XRef, "Androidxref oreo 8.0.0_r4," http://androidxref.com/8.0.0_r4/xref/art/runtime/gc/space/large_object_space.h, 2018.

[14] "Memfetch," https://github.com/citypw/lcamtuf-memfetch, [Online; accessed 17-March 2018].

[15] "Memparser Analysis Tool by Chris Betz," http://old.dfrws.org/2005/challenge/memparser.shtml, [Online; accessed 15-March 2020].

[16] "Androidxref oreo 8.0.0_r4," http://androidxref.com/8.0.0_r4/xref/art/runtime/safe_map.h#171, 2018.

[17] "Androidxref oreo 8.0.0_r4," http://androidxref.com/8.0.0_r4/xref/art/runtime/gc/heap.h, 2018.

[18] A. XRef, "Androidxref oreo 8.0.0_r4," http://androidxref.com/8.0.0_r4/xref/art/runtime/gc/heap.cc#486, 2018.

[19] "Androidxref oreo 8.0.0_r4," http://androidxref.com/8.0.0_r4/xref/art/runtime/gc/space/region_space.h#58, 2018.

[20] Boost, ""memory allocation algorithms "," https://www.boost.org/doc/libs/1_47_0/doc/html/interprocess/memory_algorithms.html, 2017.

[21] A. Developers, "Android developers byte array," http://androidxref.com/8.0.0_r4/xref/external/skia/src/core/SkWriteBuffer.cpp#37, 2018.

[22] *Android Developers Char Array*, 2018 (accessed July 20, 2020). [Online]. Available: https://developer.android.com/reference/java/io/CharArrayWriter

[23] *Android Developers String Array*, 2018 (accessed July 20, 2020). [Online]. Available: https://developer.android.com/guide/topics/resources/string-resource

[24] Google, "Google Play," https://play.google.com/store?hl=en, year=2018.

[25] "Genymotion Desktop," https://www.genymotion.com, [Online; accessed 10-January 2018].

[26] B. Saltaformaggio, R. Bhatia, Z. Gu, X. Zhang, and D. Xu, "Vcr: App-agnostic recovery of photographic evidence from android device memory images," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, 2015, pp. 146–157.

[27] ——, "Guitar: Piecing together android app guis from memory images," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, 2015, pp. 120–132.

[28] B. Saltaformaggio, Z. Gu, X. Zhang, and D. Xu, "{DSCRETE}: Automatic rendering of forensic information from memory images via application logic reuse," in *23rd {USENIX} Security Symposium ({USENIX} Security 14)*, 2014, pp. 255–269.