DroidScraper: A Tool for Android In-Memory Object Recovery and Reconstruction

Aisha Ali-Gombe
Towson University
aaligombe@towson.edu

Sneha Sudhakaran
Louisiana State University
ssudha1@lsu.edu

Andrew Case
Volatility Foundation
andrew@dfir.org

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

Abstract

There is a growing need for post-mortem analysis in forensics investigations involving mobile devices, particularly when application-specific behaviors must be analyzed. This is especially true for architectures such as Android, where traditional kernel-level memory analysis frameworks such as Volatility [9] face serious challenges recovering and providing context for user-space artifacts. In this research work, we developed an app-agnostic userland memory analysis technique that targets the new Android Runtime (ART). Leveraging its latest memory allocation algorithms, called region-based memory management, we develop a system called *DroidScraper* that recovers vital runtime data structures for applications by enumerating and reconstructing allocated objects from a process memory image. The result of our evaluation shows *DroidScraper* can recover and decode nearly 90% of all live objects in all allocated memory regions.

1 Introduction

In recent years, there has been a significant increase in the adoption and reliance on memory forensics for incident response and malware analysis. While traditionally memory analysis was used to supplement disk forensics in the recovery of critical data such as deleted messages from volatile storage, memory forensics has evolved into an advanced methodology for investigating and identifying memory-resident, kernel-level attacks and malicious behaviors that do not leave an identifiable footprint on the disk [16].

In mobile devices, the advancement and sophistication in application development and the reliance on these devices by many users make them a critical source of evidence for digital investigations. The ability to recover and reconstruct pieces of application data which otherwise may be difficult to identify using traditional static and dynamic analysis can provide investigators with substantial evidence to leverage in cybercrime and malware analysis. However, due to the layers of abstraction between the kernel and the application in the mobile architecture, recovering in-memory application data is not feasible from the residual in-memory kernel data structures. On the other hand, userland memory artifacts can provide sufficient information for investigators to recover application functionality and outline the actions, strategy, and attack evidence without the need for prior knowledge of application logic.

In this research, we target userland memory analysis on Android. The objective is to develop an app-agnostic, per-process technique that can perform post-mortem analysis on Android userland memory dumps to recover and reconstruct vital in-memory evidence for cybercrime investigations and malware and vulnerability analysis. Our effort leverages the new Android Runtime (ART) to trace all objects allocated within a process' address space and then make a best effort to rebuild those objects. This approach is built upon ART's newest garbage collection algorithm, Concurrent Copying collector, which uses region-based memory management for object allocations. In this algorithm, objects are allocated in program specific memory regions, and during garbage collection, an entire region is collected if its number of live objects is less than a certain threshold. With this algorithm, objects tend to live longer in memory when allocated in a region with high live threshold even if the specified object is marked for deallocation. The design of our approach involves identifying crucial ART structures, and subsequently, all objects allocated at runtime in a target process' address space.

In comparison with traditional static app analysis systems, which are often affected by obfuscation such as class and data encryption, dynamic class and Java reflection, etc., this approach has the advantage of recovering de-obfuscated runtime artifacts. Furthermore, unlike dynamic analysis methodologies, our technique relies only on having a process memory dump and is therefore less likely to be affected by anti-analysis techniques.

The contributions of this research work are: (1) We propose a new memory forensics technique, *DroidScraper*, that relies on the design of Android's ART region-based memory allocation to recover and reconstruct in-memory runtime artifacts.

(2) The proposed *DroidScraper* can extract running threads, enumerate objects allocated in the heap region, and then decode objects based on their class definitions. (3) Our evaluation of *DroidScraper* shows that it can recover in-memory objects with an almost 90% success rate and can be applied to behavioral post-mortem monitoring of Android applications.

2 Background

Android applications are typically written in Java and compiled into bytecode which is then executed on a Dalvik Virtual Machine (for older devices) or Android Runtime (ART) (from Android 5.0 and beyond). These Java applications can also be integrated with native code written in C/C++, with the help of Java Native Interface (JNI). ART provides a number of new features, most notably enhanced garbage collection (GC) algorithms, which affect object allocations in the new runtime.

The new Android runtime - ART has two groups of garbage collectors based on the ActivityManager process state [11]. The *ForegroundCollector* handles GC when an app is in the foreground, while the *BackgroundCollector* handles GC when an app is running in the background. A variable for the *ForegroundCollector* comes preconfigured as part of the default runtime options on stock Android as shown in Listing 1. At heap initialization, the system reads the runtime options to determine which collectors to use. Also, this variable defines the memory allocation scheme *(AllocatorType)* to be employed, and subsequently, the memory space *(Space)* to be created and how objects are organized in these spaces. Listing 2 shows all the available allocators on ART.

```
bool Runtime::Init(RuntimeArgumentMap&&
 ↪  runtime_options_in) {
  .....
  XGcOption xgc_option =
   ↪  runtime_options.GetOrDefault(Opt::GcOption);
  heap_ = new gc::Heap
   ↪  (runtime_options.GetOrDefault
   ↪  (Opt::MemoryInitialSize),
    .....
    runtime_options.GetOrDefault
     ↪  (Opt::ImageInstructionSet),
    // Override the collector type to CC
     ↪  if the read barrier config.
    kUseReadBarrier ? gc::kCollectorTypeCC :
     ↪  xgc_option.collector_type_,
    kUseReadBarrier ? BackgroundGcOption
     ↪  (gc::kCollectorTypeCCBackground)
    runtime_options.GetOrDefault
     ↪  (Opt::BackgroundGc),
    .....);
}
```

Listing 1: Runtime Initialization in runtime.cc

```
enum AllocatorType {
    kAllocatorTypeBumpPointer,
    kAllocatorTypeTLAB,
    kAllocatorTypeRosAlloc,
    kAllocatorTypeDlMalloc,
    kAllocatorTypeNonMoving,
    kAllocatorTypeLOS,
    kAllocatorTypeRegion,
    kAllocatorTypeRegionTLAB,
};
inline constexpr bool IsTLABAllocator
 ↪  (AllocatorType allocator) {
    return allocator == kAllocatorTypeTLAB ||
     ↪  allocator ==kAllocatorTypeRegionTLAB;
}
```

Listing 2: Available memory allocator types for ART

ART is designed with four major garbage collection algorithms with each one utilizing one or more of the *AllocatorTypes* in Listing 2. The complexity of choosing collectors and mapping them to one or more allocator types makes the memory allocation and GC on the new Android runtime a very complicated process. As mentioned above, the chosen collector at system startup determines the garbage collection algorithm, which in turn determine the specific *AllocatorType* to be used by the runtime environment. Below is a detailed description of the GC algorithms in ART.

- Semi-Space - SS - this algorithm uses the semi-space garbage collection to copy movable objects between two Bump Pointer spaces. In the SS algorithm, objects are allocated in free memory space using *BumpPointer* and *DlMalloc* for mutable and non-mutable objects, respectively. When the use of Thread Local Allocation Buffers is enabled, this algorithm defaults to using the *TLAB* allocator. On newer Android versions, the BumpPointer spaces are currently only used for ZygoteSpace construction.

- Generational Semi-Space - GSS - This algorithm leverages heap organization to optimize the simple Semi-Space GC above. The generational hypothesis states that most objects die young [28] and as such where long-lived reachable objects exist, they are relocated to a large RosAlloc space. The default object allocator for GSS is the *BumpPointer* for mutable objects.

- Concurrent Mark Sweep - CMS - this collector uses the concurrent mark-sweep algorithm to collect allocated objects unreachable from their roots from only the region of memory modified since the last GC operation [4]. The default memory *AllocatorType* for CMS is the *RosAlloc* - which is used to allocate mutable objects in runs-of-slots of the same sizes. It also uses the default C malloc *DlMalloc* for non-mutable objects. CMS was introduced

in Android 5 as the default CG algorithm for the Android runtime environment. It is designed to improve app performance through Concurrent collection. While this algorithm has significantly improved GC effort, however, it causes two long pauses during each collection cycle that often adversely affect UI responsiveness [17].

- Concurrent Copying - This technique utilizes an efficient concurrent and moving garbage collection algorithm. Utilizing region-based memory allocation, allocated objects are evacuated from a region and subsequently destroyed if and only if the region has live objects whose count is less than some percentage threshold. Furthermore, this algorithm creates a compacting heap and introduces very short pauses during collection [1]. It also utilizes a read barrier configuration that ensures mutators never see old versions of objects [3]. This configuration allows threads to efficiently and concurrently access heap objects during collection. The CC algorithm uses the *RegionSpace* allocator and if the use of TLAB is enabled, the system uses the *RegionSpaceTlab* allocator for movable objects. On newer Android versions, *RegionSpaceTlab* the is the default for most small object allocations [20].

In the earlier version of *libart* (5,6,7), the CMS collector was favored among the other collectors, thus defaulting to the use of RosAlloc for moving objects. However, in newer Android versions (8,9,10), the development and subsequent enforcement of the read barriers in the runtime options as shown in Listing1 which favors concurrent access to the heap during GC overrides the default CMS collector type to the CC [12, 17]. Furthermore, the introduction of *RegionSpaceTLAB* for per-thread objects makes it an ideal allocation mechanism for small objects.

In this research, our focus is on recovering objects allocated using the *RegionSpace* and *RegionSpaceTlab* allocators, which are based on the Concurrent Copying Collection algorithm. To the best of our knowledge, this is the first work that explores in-memory data recovery from RegionSpace memory maps for the new Android Runtime.

3 System Design

DroidScraper is an Android in-memory object recovery and decoding system that analyzes process address spaces (in the form of per-process memory dumps) for remnants of runtime objects. This system leverages low-level data structure definitions as well as generic class and references constructs provided by the Android runtime library (*libart.so*) to recover and reconstruct objects within a target process' address space. The design of *DroidScraper* provides investigators and malware analysts access to well-structured and forensically interesting data across threads and various process components such as activities and services.

As shown in Figure 1, *DroidScraper*'s workflow begins with process memory dump acquisition. This element of the workflow utilizes any available open-source tools such as Memfetch [6] to generate per process Android memory map. In cases where a complete memory image is acquired using tools like AMD [39] and Lime [36], *DroidScraper* can utilize the output of Volatility's *memdump* [10] plugin to access a process' address space. The memory acquisition process is then followed by the Runtime Data Structure Recovery (RDS) and the Object Recovery and Reconstruction (ORR) modules. These two elements of the workflow constitute the main contribution of our system. The RDS module leverages the per-process memory dump to identify and recreate major runtime structures that are essential for object allocations and deallocation such as Runtime, Heap, Heap Regions, and Threads. The ORR module then utilizes the metadata definitions in the recovered data structures to enumerate and decode reachable live objects within the process memory region.

3.1 Runtime Data Structure Recovery - *RDS*

DroidScraper's RDS is built upon the low-level data structures defined in *libart.so*. These structures are essential for building and maintaining the runtime environment, object allocation and accounting, as well as garbage collection. The *RDS* module begins by identifying the main runtime object of a target process, followed by the identification and extraction of all Linux threads belonging to the target process. It also consists of other sub-modules for heap structure recovery, the identification of the Region Space structure, and the allocated heap regions.

3.1.1 Identification of the Runtime Object

The *Runtime* object is the most crucial data structure required for Android process execution. Its members constitute essential components needed by the process for memory allocation, thread creation, JNI calls, and it serves as a link between the running process and outside environment. On Android, every process executes in its own runtime environment, which itself is forked from the zygote process. The zygote process is started by the init process at system boot together with the default Android runtime. The zygote then listens for a connection on its socket for a request to start up a new application. Upon receipt of such a request, it will fork a new Linux process, which forces the creation of a *Runtime* object that establishes the runtime environment. The zygote process then maps a copy of its shared library (*libart.so*) into the new process' address space.

```
_ZN3art7Runtime9instance_E offset = 0070a980
Runtime Base Address = 0xf233aa80
```

Listing 3: Instance address offset and the base address of the Runtime object recovered

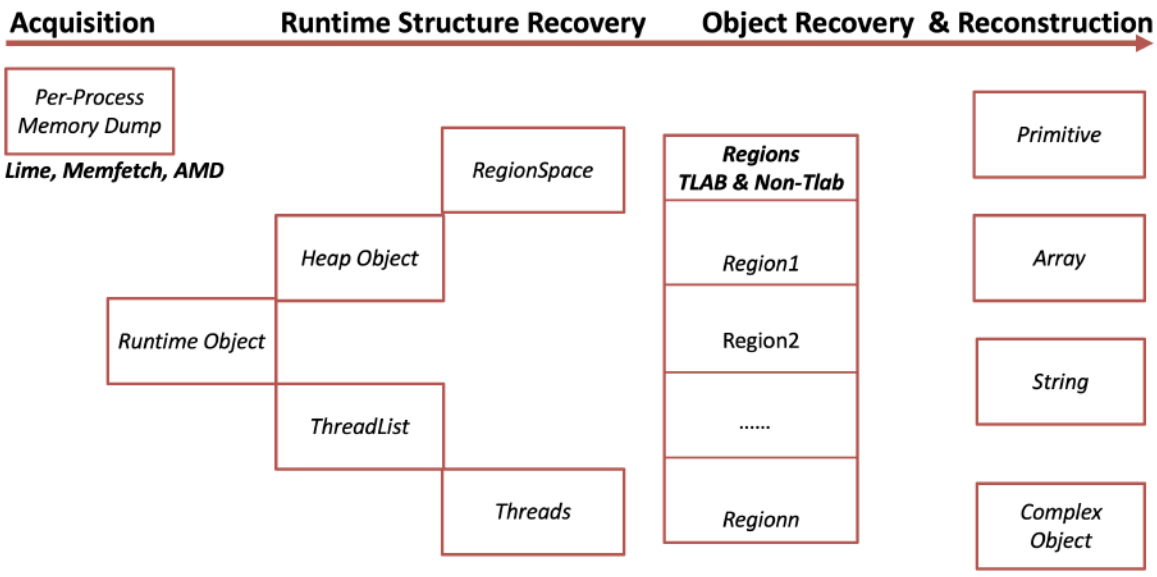

Figure 1: *DroidScraper* Workflow - Showing Acquisition, Runtime Data Structure Recovery and Object Recovery and Reconstruction modules.

In this task, our objective is to identify the location the process's *Runtime* object in the memory dump and then reconstruct the structure based on its class template definition, as given in the *libart.so*.

```
'Runtime' : [ 0x340, {
 'callee_save_methods_': [0],
 'pre_allocated_OutOfMemoryError_': [32],
 'pre_allocated_NoClassDefFoundError_': [36],
 'resolution_method_':[40],
 ....
 'heap_':[244],
 'jit_arena_pool_':[248],
 'arena_pool_':[252],
 'low_4gb_arena_pool_':[256],
 'linear_alloc_':[260],
 ....
 'monitor_list_':[268],
 'monitor_pool_':[ 272],
 'thread_list_':[ 276],
 ......}]
```

Listing 4: The Runtime object for Android 8, represented as a C structure.

Utilizing a static exploration of *libart.so* with Linux *nm* command, we identify an address offset *_ZN3art7Runtime9instance_E* in the list of symbols. This offset held the address of the *Runtime* instance when the zygote forked a new process. Because this step is critical to the entire recovery process, we verified on two different devices (Unrooted Samsung S9+ and Samsung S8 AVD) that the object file - libart.so has some basic symbols compiled in it by default and that includes the Runtime instance address. Located in the uninitialized data section of the memory where *libart.so* is mapped in process space, we identify and dereference the address in this offset to locate the beginning of the active *Runtime* instance as shown in Listing 3. We represent the Runtime class as defined in the Android documentation [12] using a C structure, with the class's fields represented as members of the structure. Listing 4 illustrate a partial Runtime object with size and member offsets for Android 8.0.

3.1.2 Enumerating Threads in User Processes

When an Android process begins executing, it spawns its first thread, called the main thread. Depending on how the program is designed, other components of the application such as activities, receivers, providers, and services may create other threads. The **ThreadListing** sub-module is tasked with enumerating all the living threads owned by the user process. Identifying and enumerating running threads is essential for virtually every memory forensics effort. Specifically for *DroidScraper*'s in-memory data recovery and reconstruction, identifying running threads is essential because the system often uses thread local allocation buffer *TLAB* for faster and more efficient object allocations. *TLAB* is a memory region assigned to a single thread for its own objects and can be used without the need to acquire and/or release locks. When the *use_tlab* option is enabled during heap creation, objects are allocated on a per thread basis and thus recovering those objects will require obtaining and analyzing the thread metadata structure.

```
Threads      TID     Name
0xe3e66e00  20392   hwuiTask1
0xe9137c00  20391   Binder:20370_3
0xe9134000  20388   RenderThread
0xe67fb000  20386   Thread-3
0xe663e400  20384   Profile Saver
0xe911a800  20383   Binder:20370_2
0xe407e400  20382   Binder:20370_1
0xe663d200  20381   HeapTaskDaemon
0xe8f7de00  20380   FinalizerWatchdogDaemon
0xf674be00  20378   ReferenceQueueDaemon
0xe8f7d800  20379   FinalizerDaemon
0xe910bc00  20377   JDWP
0xf674ac00  20376   Signal Catcher
0xe9108000  20375   Jit thread pool worker thread 0
0xf674a000  20370   main
```

Figure 2: *DroidScraper* Runtime Data Structure Recovery - ThreadListing.

From the process *Runtime* object, the **ThreadListing** finds the pointer to the beginning of the *ThreadList* object, which is at offset 0x276 on Android 8.0. The threads in the *ThreadList* structure are organized in a cyclical double-linked list. The list header contains the pointers to the first and last *Thread* objects in the list as well as the size of the list. After the identification and subsequent dereferencing of each individual thread pointer to recover all the *Threads* objects, the **ThreadListing** uses the definition of the *Threads* object in the *libart.so* documentation to retrieve each thread ID (tid) and thread name for all the living threads in the user process. Figure 2 illustrates the output of the **ThreadListing** sub-module of the *DroidScraper*'s *RDS*.

3.1.3 Recovering the Heap Structure

As the *Runtime* object initializes, it creates the *Heap* object, which is tasked with managing memory space creation, object allocation mechanisms, and accounting. At initialization, most of the arguments passed to the *Heap* are manufacturer's or design standard's runtime_options such as instruction sets, heap limit, etc. But two vital runtime_options that are crucial to *DroidScraper*'s object recovery are the *UseTlab* and *kUseReadBarrier* option. The *use_tlab* option, if present, causes threads to request and utilize the Thread local allocation buffer (Tlab) for small object allocations. As referenced in Listing 1, the *kUseReadBarrier* forces the system to use the Concurrent Copying (CC) collector which then causes the heap to allocate a different *NonMovingSpace* for non-mutable objects and a *RegionSpace* for mutable objects. On newer versions of Android Open Source Project AOSP (8,9 and 10), *kUseReadBarrier* is enabled by default thus making the Concurrent Copying the default garbage collection technique, and the memory allocations use the region-based memory management, which is the target for *DroidScraper*.

In ART's Concurrent Copying garbage collection, the heap creates a *RegionSpace* structure, which is a continuous space memory map used for the creation of equal-sized memory regions and storing their metadata. As shown in Listing 5, the *RegionSpace* holds the tally of the number of regions created by the *Heap*, the number of non-free regions, and the pointer to the array of all available *Regions*.

```
'RegionSpace' : [ 0xa8, {
        'ContinuousMemMapAllocSpace' : [0],
        'region_lock_': [56],
        'time_': [96],
        'num_regions_': [100],
        'num_non_free_regions_': [104],
        'regions_': [108],
        'non_free_region_index_limit_': [112],
        'current_region_': [116],
        'evac_region_': [120],
        'full_region_': [124],
        'mark_bitmap_': [164],
}]
```

Listing 5: The *RegionSpace* object for Android 8, represented as a C structure.

Each available *Region* structure, as shown in Listing 6, holds pointers to the beginning and end of a region, as well as the top of the region (the address of the last object allocated). It also holds the total number of objects allocated in the region and a boolean value that shows whether this region is a TLAB region or not. If a region is TLAB, then the pointer to the thread offset will be non-zero.

```
'Region' : [ 0x28, {
    'idx_' : [0],
    'begin_': [4],
    'top_': [8],
    'end_': [12],
    'state_': [16],
    'type_': [17],
    'objects_allocated_': [20],
    'alloc_time_': [24],
    'live_bytes_': [28],
    'is_newly_allocated_': [32],
    'is_a_tlab_': [33],
    'thread_': [36],
}]
```

Listing 6: The Region object for Android 8, represented as a C structure.

For this task, we developed a sub-module called the **Heap** that identifies the beginning of the process heap in the Runtime object. Dereferencing the heap address, we walk the structure to determine the offsets for the *RegionSpace* and then the pointer to the array of regions. For each recovered *Region* structure,

```
Heap Offset 0xf2297400
RegionSpace Offset 0xf6730300
Number of Regions 2048
Number of Non Free Regions 13
Region Array Offset 0xf22c06c0
TLAB Regions
Index   Begin       Top     End     Thread      Objects     Tid Thread_Name
0   0x12c00000  0x12c40000  0x12c40000  0xe8f7d800  3       20379   FinalizerDaemon
2   0x12c80000  0x12cc0000  0x12cc0000  0xe9108000  7       20375   Jit thread pool worker thread 0
3   0x12cc0000  0x12d00000  0x12d00000  0xe9134000  5       20388   RenderThread
4   0x12d00000  0x12d40000  0x12d40000  0xf674a000  630     20370   main
5   0x12d40000  0x12d80000  0x12d80000  0xe911a800  11      20383   Binder:20370_2
6   0x12d80000  0x12dc0000  0x12dc0000  0xe407e400  10      20382   Binder:20370_1
7   0x12dc0000  0x12e00000  0x12e00000  0xe9137c00  5       20391   Binder:20370_3
8   0x12e00000  0x12e40000  0x12e40000  0xe3e66e00  5       20392   hwuiTask1
9   0x12e40000  0x12e80000  0x12e80000  0xe910bc00  416     20377   JDWP
Non-TLAB Regions
Index   Begin       Top     End     Objects
1   0x12c40000  0x12c80000  0x12c80000  2472
70  0x13d80000  0x13da1ec8  0x13dc0000  1508
71  0x13dc0000  0x13e00000  0x13e00000  650
118 0x14980000  0x14982828  0x149c0000  48
```

Figure 3: *DroidScraper* Runtime Data Structure Recovery - Heap plugin output showing non-free regions in *RegionSpace*.

we walk its metadata to retrieve its index, begin address, end, top, and the number of objects allocated. Where a region is a Tlab region, the **Heap** recovers the thread offset and then utilizes the **ThreadListing** to identify the thread name and ID. The output of the **Heap** sub-module as illustrated in Figure 3 shows the offsets of the Heap and RegionSpace. The output also shows the process has a total of 2048 regions with only thirteen in use. The output further indicates the distribution of the thirteen occupied regions as having nine Tlab regions and four Non-Tlab regions.

3.2 Object Recovery and Reconstruction - *ORR*

The objective of ORR module is to perform the actual recovery of the dynamically allocated objects. This module leverages the extracted *Runtime* data structures above to identify reachable and live runtime objects and then makes the best effort to reconstruct each object.

3.2.1 HeapDump - Object Recovery

Our next sub-module is called **HeapDump**. This component is tasked with the identification and subsequent recovery of all reachable and live objects in the non-free heap regions. At creation, every object is associated with an object tree which has one or more root objects. This new object is marked reachable if it can be referenced from another reachable object. Android uses direct references to manage Java objects and indirect references for JNI code. Every instance of an object allocated is derived from the *Object* class. The object class contains two members - the class member which is defined as a *HeapReference* to the class description of the object and four bytes of monitor/hash information. The class definition determines the type of the object allocated, which in turn specifies its size. Using the algorithm defined in Algorithm 1, the **HeapDump** traces each object by decoding its object class reference to get its name and class flag which is then used to determine the object type. The sub-module then computes the size of the object based on the defined type. The location of the next object is calculated based on the size of the previous object. The output in Figure 4 prints an object offset, the class name for the object, and the object size.

Algorithm 1: HeapDump Algorithm

```
1  pos = regionBegin
2  while pos < regionTop do
3      if clazz− >ResolveClass(pos)!= nullptr then
4          obj = GetObject(pos, clazz)
5          size = GetSize(obj)
6          pos = pos+size+kAlignment
7      else
8          pos = pos+kAlignment
9      end
10 end
```

3.2.2 Object Decoding

When a process or thread allocates an object, the class and the size of the object are provided as parameters to the allocation function. At a higher level, the classes can be broadly classified into four types - Primitives, Arrays, Strings, and Complex classes. Objects in each of these classes are allocated in a unique way and as such our decoding algorithm handles each one of them differently, as shown in Algorithm 2.

```
Address 0x12c738c0 java.lang.Class - [C 18
Address 0x12c738d8 java.lang.Class - java.nio.charset.CharsetDecoderICU 72
Address 0x12c73920 java.lang.Class - [I 24
Address 0x12c73938 java.lang.Class - libcore.util.NativeAllocationRegistry$CleanerThunk 24
Address 0x12c73950 java.lang.Class - sun.misc.Cleaner 36
Address 0x12c73978 java.lang.Class - libcore.util.NativeAllocationRegistry$CleanerRunner 12
Address 0x12c73988 java.lang.Class - java.nio.HeapByteBuffer 47
Address 0x12c739b8 java.lang.Class - [B 15
Address 0x12c739c8 java.lang.Class - java.lang.String 19
Address 0x12c739e0 java.lang.Class - android.app.ActivityThread$ProviderKey 16
Address 0x12c739f0 java.lang.Class - android.app.ContentProviderHolder 21
Address 0x12c73a08 java.lang.Class - android.content.pm.ProviderInfo 95
Address 0x12c73a68 java.lang.Class - java.lang.String 59
Address 0x12c73aa8 java.lang.Class - java.lang.String 47
Address 0x12c73ad8 java.lang.Class - android.content.pm.ApplicationInfo 234
Address 0x12c73bc8 java.lang.Class - java.lang.String 47
Address 0x12c73bf8 java.lang.Class - java.lang.String 47
Address 0x12c73c28 java.lang.Class - java.lang.String 33
Address 0x12c73c50 java.lang.Class - java.util.UUID 24
```

Figure 4: *DroidScraper* Runtime Data Structure Recovery - HeapDump.

[String Object]

```
Aishas-MBP-2:ART aishacct$ python artProj.py mal8/ Heap decodeObject 0x13014130
@ Address 0x13014130
0x6f2ca108 is a Global Reference
++++++++++++++++++++++++++++++++++++++++++++++++
Reference Class is a Class Instance
Number of Reference Instance Fields = 0
The data for java.lang.String is plbslog.umeng.com
```

[Complex Object]

```
Aishas-MBP-2:ART aishacct$ python artProj.py mal8/ Heap decodeObject 0x12fa23a0
@ Address 0x12fa23a0
No reference for 0x6f6dc2e0
++++++++++++++++++++++++++++++++++++++++++++++
Reference Class is a Class Instance
Number of Reference Instance Fields = 2
android.app.ContextImpl$ApplicationContentResolver kClassFlagNormal
Super Class Offset android.content.ContentResolver
Object Size 32
FieldName - mContext - Landroid/content/Context; offset 8
Data --- 0x12fa22d8
FieldName - mPackageName - Ljava/lang/String; offset 12
Data --- 0x12f82010
FieldName - mRandom - Ljava/util/Random; offset 16
Data --- 0x12fa3388
FieldName - mTargetSdkVersion - I offset 20
Data --- 26
FieldName - mMainThread - Landroid/app/ActivityThread; offset 24
Data --- 0x12f809a8
FieldName - mUser - Landroid/os/UserHandle; offset 28
Data --- 0x6f61da20
```

Figure 5: *DroidScraper*'s DecodeObject sub-module - Recovering a String object and an instance of java.net.URL.

1. Primitive - A primitive class stores basic data types. Objects in this category are allocated according to the data type they represent. The component size for primitive ranges from 0 for Void, 1 for Boolean and Byte, 2 for Short and Char, 4 for Integer and Float, and 8 for Long and Double. If an object class is of type primitive, the data is read based on its component size.

2. Array - An object of type array holds a group of data items of the same size beginning at a contiguous memory location. When an object class resolved to an array, the 4 bytes after the *Object*'s inheritance structure is the length of the array followed by the data offset. The total size of an array is the sum of its metadata and the product of its component size and length. For instance, an integer array of size four will have a total size of 32 bytes including alignment.

3. String - An object of type string is allocated by providing the length of the string encoded in a 32-bit integer variable called count. The count is written right after the *Object*'s inheritance structure, followed by the hashcode of the string and then the beginning of the data.

4. Complex Class - This category hold instances of any class that is not primitive, array, or string. Objects in this category have a complex structure called *Class* that holds the name of the object, its size, its class size, superclass, fields, and methods, among other members. To decode an object in this category, we first find the pointer to the *Art_Field* structure. This structure helps us identify and decode all the members of this instance, their names, and offsets.

Algorithm 2: ObjectDecode Algorithm - GetObjectData(obj)

```
1  if clazz==Primitive then
2  |   len = obj− >getType()->getComponentSize()
3  else if clazz==String then
4  |   len = obj− >len
5  else if clazz==Array then
6  |   type = obj− >getType()
7  |   len = obj− >getComponentSize(type)
8  else
9  |   clazzName = obj− >getName()
10 |   fields[] = obj− >getFields()
11 |   len = fields− >getLength()
12 end
13 data = read(obj, len)
```

As shown in Figure 5, we developed a sub-module called the **DecodeObject** that decode and reconstruct live and reachable objects from the process memory dump.

4 Evaluation

DroidScraper is developed as a standalone memory forensics tool that analyzes per-process Android application memory maps for remnants of runtime artifacts. With its generic implementation, *DroidScraper* can be utilized to examine process memory directly extracted from Android devices using tools such as memfetch [6] or indirectly generated by plugins such as Volatility's memdump [10], without any background knowledge of the target application's data structures. The current version of *DroidScraper* is written in Python and works on process memory images obtained from any Android device running versions 8.0 and 8.1. The free and open source version of this tool will be released with the publication of this article.

4.1 Accuracy of Objects Recovery and Reconstruction

To assess the effectiveness of our approach for object recovery, we performed a series of experiments on memory images generated across a wide variety of applications. These include six malware samples from the CICAndMal2017 dataset [22] and VirusShare [37] and six benign applications - *Signal, EvolveSMS, Keeper, Calculator Vault, Clock Vault,* and *Google Chrome* running the *Facebook* web application, all downloaded from Google play [19].

4.1.1 Experimental Setup

Our evaluation used the Genymotion Desktop Android emulator as the execution environment. With over 3000 virtual Android device configurations and full sets of hardware sensors, Genymotion can emulate real Android devices with a high degree of accuracy [5]. We created AVDs for Google Pixel running Android 8.1-API 27 and Google Nexus 6 and Samsung S8 running Android 8.0-API 26. All the emulators have 4GB memory and are equipped with one Gmail account and a couple of fake SMS and contacts to simulate real devices. Each application was then installed on a selected device and interacted with manually by the authors to generate sufficient activity. We then captured the process memory image using Memfetch.

4.1.2 Object Recovery

To analyze the memory dumps using *DroidScraper*, we used a MacBook Pro as the host system. The captured memory images on the AVDs are pushed to the Mac using the Android *adb* [2] utility. We first execute the **ThreadListing** to enumerate all living threads and then run the **Heap** module to itemize the allocated regions (Tlab and non-Tlab) and obtain an object count per region. Finally, we run **HeapDump** to recover the reachable live objects. As shown in Table 4.1, the Total Objects column represents the cumulative total objects count from each region. This value tallies the *objects_allocated_* field of the *region* structure for all the non-Tlab regions and the *thread_local_objects* field of each thread in the Tlab regions. As mentioned in Section 3, the **HeapDump** module makes the best effort to decode the type of object and its size in an allocated memory region. However, not all objects can be recovered as some may have been deallocated even though the region is not collected. Nevertheless, in Android's Region-based memory management, a region will be completely collected only if the live threshold is below 75%. Thus, for each allocated region, the percentage recovery will be higher than 75 which in turn means the total recovery percentage for each of the test apps must be higher than 75.

For better analysis, our recovered objects are grouped into Primitives, Arrays, Strings, and Objects. The total of all the recovered objects is given in the Total Recovered column. For the measure of performance of *DroidScraper*, the recovery percentage is computed as a fraction of the total recovered objects to the cumulative tally of the objects in all the allocated regions. For each test app, the recovery percentage is given in the % column. The average recovery percentage across all the test applications is approximately 89.6%. This means that *DroidScraper* can recover nearly 90% of all objects allocated within the process memory space in well-structured formats. This percentage likelihood further buttresses *DroidScraper*'s ability to retrieve crucial and forensically interesting runtime artifacts created by an Android process.

4.1.3 Object Reconstruction

From the results obtained in the object recovery above, we performed an in-depth analysis of two samples. As a representative of the application dataset, we selected one malware and one benign application and examined them for the presence of any forensically interesting data.

RansomBO (com.yandex226.yandex967) - In the analysis of this malware sample we found evidence that it utilizes an instance of Chromium *org.chromium.content.browser.PopupZoomer* as its main activity View. This View is designed to automatically terminate itself and delete the application icon after a few seconds, changing the *Animating* and *Showing* members of the PopupZoomer View to false. We also found evidence of file activities in the *android.app.SharedPreferencesImpl* object, where the malware opened and read data from a shared preferences XML file called *maxiettings.xml*. This file contains the server *URL http://212.56.214.233/task.php* and other connection information. We also recovered and decoded the *com.android.okhttp.internal.huc.HttpURLConnectionImpl* object, which showed that the malware made a POST request to the server using an instance of *com.android.okhttp.Request* with the following header values:

Applications	Threads	Regions	Total Objects	Primitives	Arrays	Strings	Objects	Total Recovered	%
com.baidu.mbaby	16	3	2780	6	526	671	1235	2438	87.7
com.losg.netpack.BaApp	28	10	12493	77	2555	1929	6436	10997	88.1
com.caf.fmradio	15	7	7707	43	3016	1126	2878	7063	91.6
com.yandex226.yandex967	30	12	10346	161	2126	1547	5895	9729	94
cn.myhug.baobao	30	17	50529	133	6410	8571	12025	44297	87.7
com.easyhin.usereasyhin	31	15	29654	2847	6856	5391	29203	27139	91.5
Keeper	44	102	264237	2623	71823	39507	107348	221301	83.8
CalculatorVault	53	22	44757	271	8464	7320	23225	39280	87.8
ClockVault	87	31	99641	2428	15664	10287	60309	88688	89
EvolveSMS	24	36	33234	169	6565	6195	18530	31459	94.7
Signal	36	48	287650	129599	22671	24369	79528	256167	89.1
Chrome Browser	31	11	20628	208	3563	4566	10315	18652	90.4

'Content-Type', 'multipart/form-data; boundary="===1552903509936==="', 'Accept', 'application/json', 'http.agent', 'User-Agent', 'Dalvik/2.1.0 (Linux; U; Android 8.0.0; Samsung Build/OPR6.170623.017)'

Checking for a response to this connection, we found that the connection did not go through. The DetailedMessage member of the *java.net.SocketTimeoutException* has the following exception message:

failed to connect to /212.56.214.233 (port 80) from /10.0.3.15 (port 47488) after 5000ms.

Keeper - In the analysis of this Vault app, we found that the application creates a database file using the email address provided by the user and then saves the hash of the user-supplied password in the file. As contents are added to the vault, the application queries the database for the hash, then encrypts the data using the password hash, an initialization parameter, and an encryption key. We found evidence of 115 instances of *javax.crypto.spec.IvParameterSpec* and 152 instances of the *javax.crypto.Cipher$InitParams* objects in the memory dump. The reconstruction of each initialization vector object (IV) reveals the 16 byte IV values in clear text. The IV value is random and unique across all the 115 instances. For the Initialization parameter, the *javax.crypto.spec.SecretKeySpec* members show the data is encrypted using an AES algorithm and the 32-byte key is shared across all the 152 instances. The complete transformation of any content uses a combination of AES/CBC/PKCS7Padding for crypto, feedback mode, and padding respectively. But the most interesting part of this analysis is at the beginning of the first instance of *IvParameterSpec*, we found the database query and hash of the password. This is then followed by the new instance of *IvParameterSpec*, *Cipher$InitParams*, and *Cipher$Transform* with all the bytes in the block stored consecutively in a row. Thus it is easy to figure out and reconstruct the original block, the IV, the key and the transformation algorithm from the recovered objects.

4.2 Case Study

In this part of the evaluation, we will use a case study to illustrate the application of *DroidScraper*'s data recovery and reconstruction for program analysis, specifically, in regenerating program control flow to prove database access based on remnants of runtime allocations. We created a small piece of code that queries three columns (_id, address, body) from the SMS database as shown in Figure 6 - (1) below. The result of this query is populated in a *Cursor*, and then the data is read into a *StringBuilder* by moving the *Cursor* from the first position to the last. As mentioned in Section 2, with region-based memory management, objects are added at the top of a region and the new top is calculated by adding the object byte size to the old top. Thus, with such allocation, it is possible to trace precise control flow of the code by examining the objects allocations as shown in Figure 6.

Mapping the code segment (1) one-to-one with the partial output of the **HeapDump** plugin in (2), it is noticeable that right after the *CursorWrapper* object is allocated, the *StringBuilder* and the subsequent strings that are used to read the data from the *Cursor* were also created in the same region. Thus as shown in (9), the return string of the *StringBuilder.toString()* function, which contains all the messages queried from the SMS database, is allocated at 0x12c748a0. However, to give context to the read messages, we need to understand how Android performs database accesses.

In order to access any database content on an Android device, the requesting application uses its *Resolver* object to perform a CRUD operation. Each of the available CRUD functions requires at least a *URI* and in some cases, valid column names and conditions. Based on the *URI*, the *Resolver* sends the request to its corresponding *Provider*. The task of the *Provider* is two-fold - it validates the URI by matching its authority to the URI's authority, and then performs a permission check on the requesting app, after which it creates and sends a valid *SQLiteStatement* to the native SQLite engine for processing. Thus, based on this code segment, the runtime creates a *URI* object as shown in (3), which authority is "sms" and uriString is "content://sms/inbox". The corre-

sponding *Provider* object whose authority matches that of the SMS URI is shown in (4). Upon approval of the *Resolver*'s database request, the runtime creates a *CursorWrapper* object, which implements the *Cursor object*. The instance of this object is allocated at address 0x12c743d0 as shown in (5). The instance of the *CursorWrapper* contains pointers to the a) *Cursor* object, as shown in (6), which holds the column names, count and the data, b) *Provider*, as shown in (4), points to the database currently being accessed and c) *Resolver* in (7) which points to the client currently accessing the content - in this case, the package of the test application. Thus, by identifying data of interest, we can utilize *DroidScraper* to regenerate program control flow, give context to the data within the flow, and then reconstruct its members and inner members.

4.3 Challenges and Limitations

The primary limitation of any memory forensics techniques on mobile devices is the ability to acquire a memory image. The security design of smartphones limits user access to low-level components and as such almost all the available techniques currently used in practice either require rooting the device to run sudo-based utilities such as Memfetch or creating a custom ROM and/or custom kernel module like Lime to acquire the entire RAM image [36]. Other hardware-based solutions are also available using JTAG [21] and recently the work of [39] explored the use of recovery partitions for acquiring device memory images.

Other limitations of *DroidScraper* include dead Objects that have been deallocated but remain in an allocated region. Although *DroidScraper* has an almost 90% recovery rate, the remaining 10% of unrecovered objects can still pose some limitations in identifying or giving context to data of interest. Also, when garbage collection occurs, regions with less than 75% live percentage are collected, and every object in this region is deallocated. While from our observations the presence of a large corpus of objects needed for rendering GUI and other vital user data often prevents a regions' live data from inching below 75%, there is no obvious way to prevent GC from happening.

5 Related Literature

Volatile memory or RAM is a core component of modern computing devices. Every application requires some chunk of available memory to layout its code, data, and other resources at runtime. As instructions are executed, the runtime environment will consistently create new allocations, update existing ones, and deallocate unused ones, as needed. This process makes the RAM a hive of potentially interesting forensic evidence. Furthermore, mobile phones are highly dynamic execution environments, often driven by external events and user interactions, resulting in many sensitive forensic artifacts appearing *only* in volatile memory. As such, in this research work, we leverage the idea of memory forensics to develop a system that recovers and reconstructs remnants of in-memory application data from Android userland address spaces.

Traditional memory forensics has focused largely on analyzing content in kernel space [8, 9, 16]. These techniques, while effective in recovering important artifacts such as running processes, network sockets, etc., fall short in many ways. Because of the architecture of Android and other object-oriented design models, focusing solely on kernel-based memory analysis fails to target key process components such as *ActivityInfo*, *ActivityThread*, *PackageManager* and other objects like decrypted *HttpRequests*, *Cursor*, *StringBuilder*, and *Arrays*.

Thus more recent memory forensics research has explicitly targeted managed runtime recovery efforts as presented in the work of [15, 25, 28, 35]. These techniques are designed to recover and reconstruct data allocated on a per-process basis. In 2011, Case presented the first approach for examining the contents of the Android Dalvik Virtual Machine (DVM) [15]. This work was extended to cover a newer version of Android DVM in [25]. Soares developed a technique for extraction and analysis of contents in the Android ART runtime allocated using the new RosAlloc memory management scheme [35]. Much like [15,25,35], *DroidScraper* is also an Android runtime-based recovery technique that specifically targets the recovery and reconstruction of objects allocated using the Region-based memory management. The Region-based memory allocation is the default memory management scheme included in the latest revision of libart beginning 2017 [17]. A prior object recovery effort targeting Region-based memory management called RecOOP was presented in the work of Pridgen et al. [28,29]. Unlike *DroidScraper* which is designed for the Android ART, RecOOP was specifically designed for HotSpot Virtual Machine (JVM). In addition to the obvious architectural differences, the memory allocation scheme and its corresponding garbage collection mechanisms, which determine the base runtime data structures and the layout of objects, are entirely different.

Other widely adopted memory forensics techniques in practice include generic memory scanning with utilities such as *strings* and *jbgrep*. These methods are often employed for identifying textual data in allocated or insecurely deallocated memory spaces [26, 40]. However, due to various program obfuscation techniques and the general complexity of programs, the data of interest may not be layed out sequentially. In addition, considering that these techniques are oblivious of the memory allocation scheme, the recovered data often lacks context. With *DroidScraper*, the recovery and reconstruction effort is designed based on a specific Android memory allocation scheme. This system identifies and decodes crucial runtime data structures that hold the definitions, metadata, and accounting information of allocated objects. Other specialized memory scanning techniques have targeted more specific data structure other than strings [7, 14, 18, 23, 24, 27, 34, 38]. These

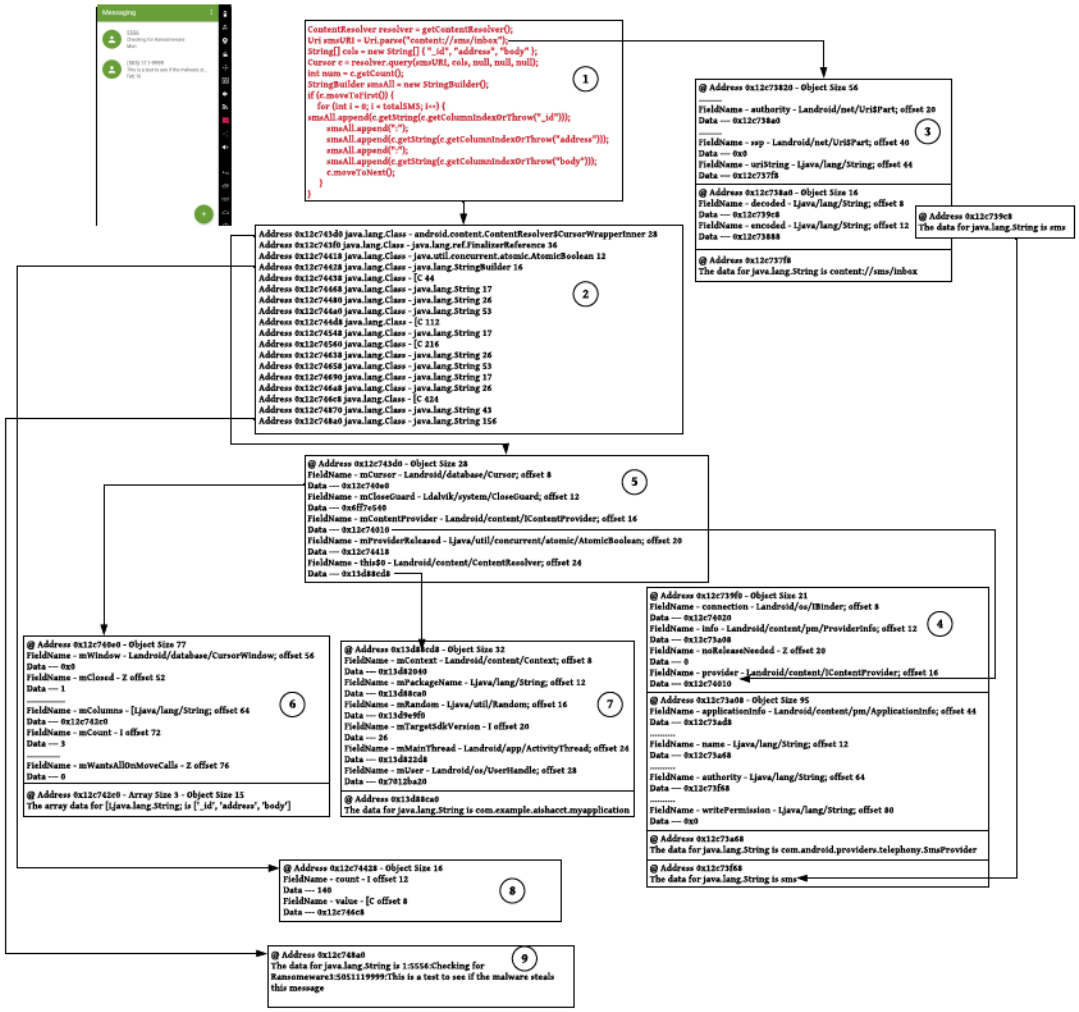

Figure 6: Proving database access using *DroidScraper*'s Object Recovery and Reconstruction modules.

approaches require prior definitions of the data structures or the profile of its members. With *DroidScraper*, no knowledge of the application data structures or objects is required.

As an alternative to raw memory analysis, [13, 30–33] have employed system instrumentation or special-purpose runtime tracing of object allocations. DSCRETE is a content reverse engineering technique that reuses application logic from a target application extracted previously using Intel PIN to scan and render in-memory data structures [33]. [31] leverages memory images of Android device cameras to recover photographic images. Saltaformaggio et al. proposed GUITAR - a tool that rebuilds and redraws an app GUI from smartphone memory images based on the low-level definition of the Android GUI framework [30]. The authors further extend this work with a more advanced memory forensic technique that performs spatial-temporal recreation of screens of Android apps from memory images [32]. Bhatia et al. presented Timeliner - an AOSP plugin that identifies and recovers residual data structures [13]. Timeliner infers user-induced transitions between corresponding activities by building a transition graph and then reconstructing a cross-app Activity timeline. In comparison with these related efforts, our contribution is an app-agnostic technique that is not limited to any special-purpose scenario. *DroidScraper* is a generic approach that can recover *any* kind of in-memory object such as activity data, network structures, messages etc. without the need for instrumentation or modification of the AOSP code. Furthermore, *DroidScraper* does not require prior definitions of low-level GUI structures to extract in-memory GUI objects such as *Views*.

6 Future Work

Currently, *DroidScraper*'s object recovery mechanism extracts and dumps nearly 90% of reachable and live objects from a process memory space. Depending on the type of ap-

plication, these objects can range from as little as 2000 to as many as 250,000 objects. With such large sets of objects, looking for forensically interesting data can be a tedious process. Although the layout of the region-based allocation helps tremendously in tracing and predicting control flow, a more automated approach to program reconstruction is needed. Thus as part of our future work, we will develop an automated system that can reconstruct an application's components by mapping the allocated objects to the code section of the application. This automated process will help us generate a proper execution path, reconstruct the GUI, and trace other program segments, such as background services. Furthermore, we are currently working to extend *DroidScraper* to cover Android 9 and 10.

7 Conclusions

As mobile devices continue to evolve, program analysis remains crucial for forensics investigations. From cybercrime to malware and vulnerability analysis, userland memory forensics can provide a better alternative to traditional techniques especially in multi-stack architecture. In this paper, we presented *DroidScraper* - a userland in-memory object recovery and reconstruction system that recovers the remnant of runtime artifacts from Android process memory space. The evaluation of *DroidScraper* has shown that it can recover in-memory data allocated using Android's region-based memory allocation with a recovery percentage of almost 90%. In addition, *DroidScraper* can reconstruct and give context to the extracted objects, which in practice can be utilized for detecting evidence of file and network activities, database accesses as well as recovery of cryptographic keys.

Acknowledgment

This material is based upon work supported by the National Science Foundation under Grant #1703683 and #1850054.

References

[1] Android 8.0 ART Improvements. https://source.android.com/devices/tech/dalvik/improvements. [Online; accessed 03-January 2019].

[2] Android Debug Bridge (adb). https://developer.android.com/studio/command-line/adb. [Online; accessed 16-March 2019].

[3] Concurrent and Parallel Garbage Collection. http://www.cs.umd.edu/class/fall2013/cmsc433/lectures/concGC.pdf. [Online; accessed 25-February 2019].

[4] Concurrent Mark Sweep (CMS) Collector. https://docs.oracle.com/javase/8/docs/technotes/guides/vm/gctuning/cms.html. [Online; accessed 11-December 2018].

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

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

[7] Memparser Analysis Tool by Chris Betz. http://old.dfrws.org/2005/challenge/memparser.shtml. [Online; accessed 15-March 2019].

[8] Rekall Forensics. http://www.rekall-forensic.com. [Online; accessed 03-March 2019].

[9] The Volatility Framework. https://www.volatilityfoundation.org. [Online; accessed 15-January 2019].

[10] Volatility Command Reference. https://github.com/volatilityfoundation/volatility/wiki/Command-Reference#memdump. [Online; accessed 21-March 2018].

[11] Android. Debugging ART Garbage Collection. https://source.android.com/devices/tech/dalvik/gc-debug#invalid_root_example, 2015.

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

[13] Rohit Bhatia, Brendan Saltaformaggio, Seung Jei Yang, Aisha Ali-Gombe, Xiangyu Zhang, Dongyan Xu, and Golden G Richard III. Tipped Off by Your Memory Allocator: Device-Wide User Activity Sequencing from Android Memory Images. In *Proceedings of the Network and Distributed Systems Security Symposium (NDSS'18), San Diego*, 2018.

[14] Chris Bugcheck. Grepexec:Grepping Executive Objects From Pool Memory. *Annual Digital Forensic Research Workshop (DFRWS)*, 2006.

[15] Andrew Case. Memory Analysis of the Dalvik (Android) Virtual Machine. *Source Seattle*, 2011.

[16] Andrew Case and Golden G Richard III. Memory Forensics: The Path Forward. *Digital Investigation*, 20:23–33, 2017.

[17] Matheiu Chartier. Performance and Memory Improvements in Android Run Time (ART)(Google I/O '17), 2017.

[18] Brendan Dolan-Gavitt, Abhinav Srivastava, Patrick Traynor, and Jonathon Giffin. Robust Signatures for Kernel Data Structures. In *Proceedings of the 16th*

ACM Conference on Computer and Communications Security, pages 566–577. ACM, 2009.

[19] Google. Google Play. https://play.google.com/store?hl=en, year=2019.

[20] GoogleGit. Git Repositories on Android. https://android.googlesource.com/platform/art/+/master/runtime/gc/allocator_type.h, 2018.

[21] Keonwoo Kim, Dowon Hong, Kyoil Chung, and Jae-Cheol Ryou. Data Acquisition from Cell Phones Using a Logical Approach. In *Proceedings of the World Academy of Science, Engineering and Technology*, volume 26, 2007.

[22] Arash Habibi Lashkari, Andi Fitriah A Kadir, Laya Taheri, and Ali A Ghorbani. Toward Developing a Systematic Approach to Generate Benchmark Android Malware Datasets and Classification. In *2018 International Carnahan Conference on Security Technology (ICCST)*, pages 1–7. IEEE, 2018.

[23] Zhiqiang Lin, Junghwan Rhee, Chao Wu, Xiangyu Zhang, and Dongyan Xu. Dimsum: Discovering Semantic Data of Interest from Un-mappable with Confidence. In *in: Proceedings of the 19th Annual Network and Distributed System Security Symposium (NDSS'12*. Citeseer, 2012.

[24] 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 *NDSS*, 2011.

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

[26] Tilo Müller and Michael Spreitzenbarth. Frost. In *International Conference on Applied Cryptography and Network Security*, pages 373–388. Springer, 2013.

[27] 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(4):197–210, 2006.

[28] Adam Pridgen, Simson Garfinkel, and Dan S Wallach. Picking up the Trash: Exploiting Generational GC for Memory Analysis. *Digital Investigation*, 20:S20–S28, 2017.

[29] Adam Pridgen, Simson L Garfinkel, and Dan S Wallach. Present but Unreachable: Reducing Persistent Latent Secrets in the HotSpot JVM. In *Proceedings of the 50th Hawaii International Conference on System Sciences*. University of Hawai'i at Manoa, 2017.

[30] Brendan Saltaformaggio, Rohit Bhatia, Zhongshu Gu, Xiangyu Zhang, and Dongyan Xu. GUITAR: Piecing Together Android App GUIs from Memory Images. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 120–132. ACM, 2015.

[31] Brendan Saltaformaggio, Rohit Bhatia, Zhongshu Gu, Xiangyu Zhang, and Dongyan 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*, pages 146–157. ACM, 2015.

[32] 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 *USENIX Security Symposium*, pages 1137–1151, 2016.

[33] Brendan Saltaformaggio, Zhongshu Gu, Xiangyu Zhang, and Dongyan Xu. DSCRETE: Automatic Rendering of Forensic Information from Memory Images via Application Logic Reuse. In *23rd USENIX Security Symposium (USENIX Security)*, pages 255–269, 2014.

[34] Andreas Schuster. Searching for Processes and Threads in Microsoft Windows Memory Dumps. *Digital Investigation*, 3:10–16, 2006.

[35] Alberto Magno Muniz Soares and Rafael Timóteo de Sousa Jr. A Technique for Extraction and Analysis of Application Heap Objects within Android Runtime (ART). In *ICISSP*, pages 147–156, 2017.

[36] Joe Sylve. Lime-Linux Memory Extractor. In *Proceedings of the 7th ShmooCon Conference*, 2012.

[37] VirusShare. VirusShare.com - Because Sharing is Caring, 2017.

[38] Robert J Walls, Erik G Learned-Miller, and Brian Neil Levine. Forensic Triage for Mobile Phones with DEC0DE. In *USENIX Security Symposium*, 2011.

[39] Seung Jei Yang, Jung Ho Choi, Ki Bom Kim, Rohit Bhatia, Brendan Saltaformaggio, and Dongyan Xu. Live Acquisition of Main Memory Data from Android Smartphones and Smartwatches. *Digital Investigation*, 23:50–62, 2017.

[40] Li Ying. Where in your RAM is "python san_diego.py"? - PyCon 2015. https://www.youtube.com/watch?v=tMKXcc2-xO8, 2015.