App-Agnostic Post-Execution Semantic Analysis of Android In-Memory Forensics Artifacts

Aisha Ali-Gombe*
Towson University, MD
aaligombe@towson.edu

Alexandra Tambaoan
Towson University, MD
atamba1@students.towson.edu

Angela Gurfolino
Towson University, MD
agurfo1@students.towson.edu

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

ABSTRACT

Over the last decade, userland memory forensics techniques and algorithms have gained popularity among practitioners, as they have proven to be useful in real forensics and cybercrime investigations. These techniques analyze and recover objects and artifacts from process memory space that are of critical importance in investigations. Nonetheless, the major drawback of existing techniques is that they cannot determine the origin and context within which the recovered object exists without prior knowledge of the application logic.

Thus, in this research, we present a solution to close the gap between application-specific and application-generic techniques. We introduce *OAGen*, a post-execution and app-agnostic semantic analysis approach designed to help investigators establish concrete evidence by identifying the provenance and relationships between in-memory objects in a process memory image. *OAGen* utilizes Points-to analysis to reconstruct a runtime's object allocation network. The resulting graph is then fed as an input into our semantic analysis algorithms to determine objects' origin, context, and scope in the network. The results of our experiments exhibit *OAGen*'s ability to effectively create an allocation network even for memory-intensive applications with thousands of objects, like Facebook. The performance evaluation of our approach across fourteen different Android apps shows *OAGen* can efficiently search and decode nodes, and identify their references with a modest throughput rate. Further practical application of *OAGen* demonstrated in two case studies shows that our approach can aid investigators in the recovery of deleted messages and the detection of malware functionality in post-execution program analysis.

*Aisha Ali-Gombe is the corresponding author.

ACSAC 2020, December 7–11, 2020, Austin, USA

ACM ISBN 978-1-4503-8858-0/20/12…$15.00
https://doi.org/10.1145/3427228.3427244

CCS CONCEPTS

• **Applied computing** → **Evidence collection, storage and analysis**; **Data recovery**; • **Security and privacy** → *Software reverse engineering*.

KEYWORDS

Userland memory forensics, android, object allocation graph, visualization, semantic analysis

ACM Reference Format:
Aisha Ali-Gombe, Alexandra Tambaoan, Angela Gurfolino, and Golden G. Richard III. 2020. App-Agnostic Post-Execution Semantic Analysis of Android In-Memory Forensics Artifacts. In *Annual Computer Security Applications Conference (ACSAC 2020), December 7–11, 2020, Austin, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3427228.3427244

1 INTRODUCTION

Traditionally, the use of memory forensics is adopted by practitioners and the scientific community as an alternative technique for the carving of volatile data such as passwords from volatile memory, which otherwise cannot be recovered using traditional disk forensics. In the last decade, this simplistic idea of string and textual data carving from raw memory images began to change dramatically with the advent of memory-resident malware. More sophisticated semantic analysis capabilities that recover and reconstruct the state of a system under investigation from the volatile memory image were developed. Tools, techniques, and algorithms were designed to inspect operating system structures for traces of malware and malware effects, construct system and user timelines of events, perform program inspection, as well as other complex cybercrime evidence reconstruction [8–10, 13, 18, 22, 24, 26, 30, 40].

The increasing dominance of smartphone usage across the general population for everyday communication and data processing makes them a critical source of evidence in cybercrime investigations and threat analysis. Thus, beyond traditional computers, the spatial aspect of memory forensics has been extended to generally cover mobile devices. Specifically, on Android, the recovery of the system-wide logs [25] and content providers such as Contacts and SMS [28] are leveraged by investigators to infer user and program activities. Various other application-specific memory analysis techniques that recover forensically interesting artifacts from well-known applications such as Facebook, default messaging apps and Telegram were also presented in the literature [3, 5, 14, 21, 29].

Although these techniques are increasingly becoming useful and often adopted by practitioners to provide forensics evidence, their methodologies are conceived based on an individual app's specific logic. Thus, their resulting recovery algorithm cannot be generalized to other applications or different versions of the same app. To illustrate this challenge, consider a scenario where a member of a criminal syndicate is busted, and an Android device is confiscated. The mobile device reveals that an in-house encrypted instant messaging application is installed. The app features an auto-delete function that destroys messages after a specific time. The forensics examiners are faced with two important tasks: 1) to examine the volatile memory for the footprint of any possible evidence, such as deleted messages; 2) to determine the provenance of the recovered evidence. Due to the app-specific requirements of the techniques mentioned above, they cannot be utilized to examine the application under investigation. Other more established kernel-level memory forensics frameworks such as Volatility will not be sufficient in this scenario, primarily because of their practical limitations in recovering application's concrete execution traces from the recovered kernel data structures, as discussed in [11].

Recently, there have been some efforts proposed in the literature to overcome the limitation of recovering valuable forensics evidence directly from userland process spaces. Notably, the work of [36, 38, 39] developed techniques for application data structure, GUI, and image reconstruction respectively by training the system to understand application logic. In [6], researchers proposed a low-level plugin for user timeline generation based on object allocations. While these contributions provide a significant improvement in this area of research, the challenge is they are limited to only the devices modified with the developed plugin or applications in which logic has been trained. As such, they are not easily deployed in real investigations. Furthermore, their recovery efforts only applied to particular scenarios, and are thus too limited in general use cases.

In this paper, we seek to overcome the gap in this spatial aspect of mobile memory forensics by developing an app-agnostic, post-execution semantic analysis technique called *OAGen*, which can be utilized by investigators for generalized path exploration, context generation, and overall object scope determination. Unlike existing methodologies that recover specific artifacts such as images and activity components, our approach is an app-generic technique that recreates any application's concrete runtime activity without prior knowledge of its execution logic. The goal is to aid analysts and forensics investigators find valuable evidence and determine their origin within the app's allocation trace. Similar to the field of archaeology, where human activities are determined by recovering and piecing together material from a culture [47], *OAGen* leverages post-mortem Points-to analysis of a process memory image to recreate the runtime's object allocation network. The resulting graph network is then used to establish relationships, context, and scope for object references, field objects, threads, GCRoots, and other Android components. We designed *OAGen* as an extension to the work of [2], which introduces the recovery and reconstruction of objects allocated in the new Android Runtime (ART) environment, based on the current default Android's Region-based memory management algorithm. With the limited ability of the tool DroidScraper [1] to dump the heap content from an Android target process, *OAGen* utilizes the heap content to recreate an entire allocation network that occurred at runtime with nodes representing live objects and edges representing the relationships between objects. Furthermore, we developed five semantic analysis modules that utilize the object allocation graph to efficiently find the relationships between objects of interest and other program components. In summary, this paper makes the following contributions:

- Development of an app-agnostic userland memory analysis technique that recovers valuable forensics evidence from a process' memory space and establishes provenance for such evidence using the application's concrete runtime allocations.
- Presentation of a a robust algorithm that constructs a network from the runtime allocations with objects as nodes and the relations between object references as edges in the graph.
- Development of efficient and effective semantic analysis modules that explore concrete allocation paths for objects of interest to determine origin, context, and scope across the entire network.

We have evaluated the efficiency of *OAGen* in recreating the allocation network across fourteen different applications. Our results indicate that *OAGen*'s throughput efficiency in searching and decoding a node, finding all of its references to be approximately 0.19Mbps. Furthermore, utilizing our semantic analysis modules, *OAGen* can find the flow path between a source and a destination or between a target object and other program components. In two case studies with real applications, we have demonstrated how an analyst can effectively use our modules to find valuable forensics evidence as well as to determine the origin of the evidence by finding the relationships amongst objects of interest and establishing context and scope for those objects within the network. The implementation of *OAGen* will be made open source upon publication of this paper.

The rest of this paper is organized as follows: Section 2 discusses the background of Points-to analysis and its application in the post-execution analysis. Section 3 describes the design of *OAGen*. Section 4 presents the implementation and evaluation of our approach. This section also discusses our results, performance, and limitations of our algorithms. Section 5 presents the literature review, and Section 6 concludes the paper.

2 BACKGROUND

One of the most fundamental design principles of the Android platform is application sandboxing. This concept requires applications to run and execute in their own runtime environment with minimal and heavily supervised interprocess communications. The new runtime environment (ART) is a very rich container that incorporates all that a process needs to execute. It is responsible for mediating with the lower-level kernel and the library layer, providing functionality such as thread, heap, and stack creation. In part, it is also responsible for object allocation and garbage collection. The newer Android versions use the Region-based memory management algorithm for object allocations. With this algorithm, the ART runtime instance divides the available memory into equal size regions and create a RegionSpace object to hold metadata of available regions of memory. An object is allocated at the top of an available region

as long as its size does not exceed the region's end.
Prior work [2, 32, 34, 35, 42] has involved developing techniques for deconstructing a process memory image and reconstructing the memory allocation metadata, dumping the heap objects, and decoding the objects. However, the limitation of these approaches is the developed tools aggregate the allocation, but do not establish the relationships between allocated objects, and therefore they cannot be used to determine the origin or scope within which an object exists at runtime. In real investigations, however, identifying the overall context of an object is required to establish concrete evidence. Thus, in this work, we seek to utilize the Android memory allocations mechanism defined in the ART library and deconstructed in [1] and the Heap-context Points-to analysis proposed by [44] to develop a robust algorithm for defining relationship paths for allocated objects in a post-execution process memory dump. To the best of our knowledge, this is the first work that explores the application of Heap Context Points-to analysis to recreate an object allocation network in a post-execution process memory image.

2.1 Points-to Analysis

Pointer analysis was originally developed as a static, compile-time technique that establishes the relationships between pointer variables and the memory locations they point to at runtime [45]. In recent years, researchers have explored different pointer analysis techniques to address concerns in program examination, security, and vulnerability analysis [44]. In this paper, we leverage Points-to analysis as a methodology for exploring and recreating the relationships between allocated objects and their references within the post-execution memory image of a process. The resulting relationship network is then utilized for an in-depth semantic analysis.

In OOP, a class defines a template for an abstraction. This template can be used to create an instance of the class, which is then referred to as an object. For example, the Java programming language represents complex ideas as objects, and most programs written in Java create thousands of objects. By design, objects in Java are created using the *new* keyword. The statement containing the new keyword or the function within which an object is created generally is referred to as the object's **allocation site**. Other objects referred to as the **receiver objects** are the objects on which the allocation site's method is called [33]. These objects are called the **allocator** objects. Variables created as objects hold references or the address location of the object's definition. Java objects can range from simple objects with no members to complex objects with other objects as members. The members defined in an object's class template are called fields. A class field is a variable (which can also be an object) that is part of the class instance's state or represents the instance's context.

To establish the relationship between objects in traditional static Points-to analysis where tentative references are not known, [44] defines a Heap-context relation as: a flow path from *a1* to *a2* signifies that *a1* is the receiver object of the method that made the allocation of *a2*, and thus *a1* is the allocator of *a2*.

Applying this concept in post-execution Points-to analysis where the execution allocation is concrete and object references in the process image are finite, we will consider every entry in the heap region to represent an allocation site. Although we may not necessarily know the method that creates the receiver object, from the Heap-context relation above, we know that an object that holds the reference to another object is the allocator object. Hence we can establish a flow path from the allocator to the allocated object. To determine the context for any allocation site, we will need to explore its allocator and all of the chain of the allocator's predecessors. On the other hand, to determine the scope of an object within a specific program component, we will track the object's allocation sites, its references, and its successors' allocation sites within the component.

3 SYSTEM DESIGN

This research work proposes a methodology for the post-execution recreation of object allocation networks and algorithms for efficient semantic analysis of forensically interesting artifacts from an Android process memory image. *OAGen* utilizes remnants of in-memory objects to develop object allocation relations. This relationship network is further utilized by the semantic algorithms to aid investigators in identifying tentative object allocation paths and determining contexts and scopes of objects in a process memory dump.

3.1 Generating Object Allocation Graph - OAG

To best model the objects' allocation relations, we represent the object allocation chain in a graph data structure. Using the Heap-context Points-to analysis discussed in Section 2, we identify relationships between objects based on their class definitions. We construct an Object Allocation Graph (OAG) to capture all the object allocation relations starting with the objects currently allocated in a heap. Our OAG is an unweighted directed graph of interconnected nodes and edges $G = (N, E)$, where a node denotes an allocated object and an edge refers to the relationship between a referenced and referred object or simply an object allocation relation.

We leverage the HeapDump utility from [1] to create a list of objects in memory and add these objects as initial nodes to the graph. HeapDump walks a process heap and identifies entries for objects in the non-free regions of memory. Due to the sandboxing mechanism we discussed in Section 2, Android processes create thousands of objects even for a very small program with few lines of code. Thus, the output of the HeapDump is fed into the OAG generator as the initial nodeList.

The generator recursively iterates over each node, decodes its object, and discovers if the object holds any reference to another object or if it has any field reference(s). If it does, irrespective of whether the newly identified reference resides in the heap, stack, initialized data region, or the code, our algorithms add it to the nodeList and an edge is then created between the current node and the new node.

Each node in the OAG has three attributes:

- ID - this represents the object allocation site.
- Label - this represents the object class.
- Data - this represents the value or field values of the objects. When an object holds a single String or primitive data,

the Data attribute is represented as a String. When the object contains members that are objects or combinations of objects/primitives, Data is a list with values for each field.

Algorithm 1 illustrates how we create the nodeList and initial graph generation. The algorithm adds a node to the graph using the *addNode()* function with corresponding attributes for ID, Label and Data. It also iterates over all the initial nodes in the graph to add edges using the *iternodes()* function. The function *recurseObject()* as shown in Algorithm 2 recursively prunes each node, then establishes a relation if a reference exists. If a reference does not exist, the generator adds data and a label to the node and proceeds to the next node. If a new object is found that is not part of the initial objects, the *recurseObject()* function calls the *decodeObject()* with the node and heap start addresses as parameters to find the class, size and the members of the object. Complex objects with non-primitive members are further deconstructed and the member references are recovered by walking the heap using the *getAllRefs()* function.

Algorithm 1: Illustrating the Initial Phase of Object Allocation Graph (OAG) Generation

```
1  parameters: String:path
2  nodeList = HeadDump(path)
3  G=Graph(strict=False,directed=True)
4  for node in nodeList do
5  |  G.addNode(node.id, node.label, node.data)
6  end
7  for node in G.iternodes() do
8  |  recurseObject(G, node)
9  end
```

3.2 Semantic Analysis

Semantic analysis is the discovery of meaning in a program [41]. In post-execution forensics examination and program analysis, identifying semantic information of data can be crucial in evidence gathering. Thus, our next research target after the OAG generation phase is the semantic analysis phase that seeks to find information about objects of interest. Our aim in this segment is to develop algorithms that can efficiently aid analysts and investigators in performing path exploration, identify the scope of an object within the OAG network, and determine the object's context. In every graph, nodes can have successors, predecessors, and neighbors. Particularly in an OAG, these relations make up the Heap Context for an object. Each one of these relations could be explored to provide some form of semantic and contextual information about data of interest. *OAGen*'s overarching objective is to develop three algorithms for semantic examinations of paths, contexts, and objects' scopes, and one search utility function.

3.2.1 Search Utility. It is essential that before we begin to explore the OAG for semantic information, we have a utility for performing a simple search for data of interest. This utility is a search engine that can help analysts understand the value, type, and location of data within the OAG. The search utility provides options for dumping all allocated sites for a specific object. This tool supports

Algorithm 2: Recursively Searching and Adding Nodes and Edges to the Graph

```
1  parameters: Node:node, Address:heapBegin_
2  decoded = decodeObject(node, heapBegin_)
3  if decoded is String then
4  |  node.label= "String Class"
5  |  node.data= decoded
6  else if decoded is Array then
7  |  if decoded is PrimitiveArray then
8  |  |  node.label= "Primitive Array Class"
9  |  |  node.data= decoded
10 |  else if decoded is ObjectArray then
11 |  |  node.label= "Object Array Class"
12 |  |  arr = getAllRefs(decoded)
13 |  |  for x in arr do
14 |  |  |  G.addEdge(node, G.addNode(x, node.id=x))
15 |  |  end
16 |  |  node.data= decoded
17 else if decoded is Object then
18 |  node.label= "Object Class"
19 |  ObjectFields=getAllRefs(decoded)
20 |  for x in ObjectFields do
21 |  |  if x is ref then
22 |  |  |  G.addEdge(node, G.addNode(x, node.id=x))
23 |  end
24 |  node.data= decoded
```

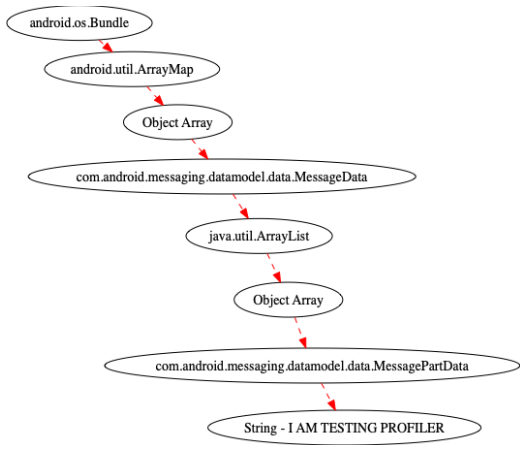


Figure 1: Illustrating path exploration from a source node to a destination node using edge_disjoint paths with the shortest augmenting path maximum flow function.

both string value search and other objects' search using their fully-qualified Java class names, which can be very handy, especially in a dense and noisy network. For example, when performing vulnerability analysis on an application to detect password persistence, an examiner might want to search for all the occurrences of the inputted password in the memory image. This tool can provide the result, including the memory address of the allocated password. The object search can be used to lookup objects of interest, such as HTTPConnection. An analyst can quickly identify where an

object of interest lies and can use the address as a target for the semantic analysis techniques. The listing below is a sample output for the search utility with the string search option "PROFILER". The output shows the memory address where each string containing the word "PROFILER" is allocated in the memory image extracted for the Android Messaging process.

```
0x12e1f3b0 String - I AM TESTING PROFILER
0x12e20150 String - PROFILER
0x12ec0760 String - I AM TESTING PROFILER
```

3.2.2 Path Exploration. This path exploration algorithm establishes connectivity between two nodes of interest. Given a source and destination node in the OAG, the path exploration algorithm finds the maximum flow path using the Edge-disjoint Paths. Kleinberg and Tardos[27] define Edge-disjoint Paths as:

> *Given directed graph G = (N, E), and two nodes s and t, k-Edge-disjoint Paths are paths with no common edge.*

The theory of Edge-disjoint Paths also establishes that the number of Edge-disjoint paths between a source and destination is equal to the connectivity between the two nodes. Applying this theory to our OAG, the k-Edge-disjoint Paths between a source allocated object and a target allocated object will return all the k connected paths between the source and target. The nodes in each path will represent all the allocators of the allocators of the target nodes that are on the path between the source and the target.

We implemented this technique using the Networkx algorithm for Edge_disjoint_paths [23]. However, due to the density of our OAGs, returning multiple paths for the path exploration will not be efficient, thus we overwrite the Edmonds-Karp flow default flow function with the shortest augmenting path flow path. This is a maximum single-commodity flow function that will force the algorithm to return the shortest of the k paths. Figure 1 is a sample output for the path exploration between source node - 0x12ec0360 and destination node - 0x12ec0760 in the OAG of the sample Messaging process memory image.

3.2.3 Context Determination. An object context defines the state of an object. As discussed in Section 2, given object allocation path *a1...a3...root*, where *a1* is an allocator object of *a2*, a2 is an allocator object of a3, and so on. [44] argues that the context of the object a1 is a simple path from a1 to root. This simple definition, while accurate, limits the understanding of object context in a very complex architecture like Android. We argue that expanding the search to incorporate all adjacent edges for each node found in the path will be beneficial, as demonstrated in Figure 2.

Our context determination algorithm as illustrated in Algorithm 3 starts with a target node and then recursively iterates all of its predecessors, adding their nodes to the nodeList until it gets to the topmost root node. We then use the nodeList to get all the edges for the corresponding nodes in the nodeList from the OAG. Finally, we create a subgraph with the resulting edgeList, which corresponds to the broader context of the target object.

The problem of finding the top-most root and expanding edges can quickly escalate and can lead to scalability problems. Thus, we recommend a depth value, which represents the maximum pruning the recursive search will do before returning the last node as the root node. Figure 2 is a cropped plot that shows the output

Algorithm 3: Algorithm for Context Determination

```
1  parameters: Node:target, Graph:G
2  nodePred=[target]
3  for n in nodePred do
4      for i in G.iterpred(n) do
5          nodePred.append(n)
6      end
7  end
8  edgeList=G.edges(nodePred)
9  H = G.subgraph(edgeList)
10 return H
```

of the Context Determination algorithm starting at a target node 0x12ec0760 (String - I AM TESTING PROFILER), with a depth of 10. The subgraph visualizes the relation between adjacent objects starting at a target node to an identified topmost root node. In Figure 2, we can see that the expanded context search algorithm helped us make a connection between the target string and the SQL query referenced in an adjacent SQLiteStatement. The edges in the network illustrate further the flows and connectivity between the nodes. It is also likely that a target object may be referenced by more than one object, thus having multiple contexts. The expanded search will merge the multiple contexts and render a larger subgraph that incorporates all the available contexts.

3.2.4 Object Scope. In OOP, the position at which a variable is declared determines its scope in a program. For post-execution analysis, examining the scope of an allocated object can help us infer further meaning as to why an object may be available in another context beyond its immediate context. In this task, we developed an algorithm to examine the scope of a target object within the boundaries of threads, program components, and its connectivity to Garbage Collection Roots (GCRoots). This more extensive search is particularly important when trying to examine how multiple objects from different sources relate.

Algorithm 4: Algorithm for Object Scope Search

```
1  parameters: Node:target, Graph:G
2  nodeList = G.getJThreads() || G.getGCRoot() || G.getComponents()
3  for node in nodeList do
4      nodeSuccessors=[node]
5      for n in nodeSuccessors do
6          for i in G.itersucc(n) do
7              nodeSuccessors.append(n)
8          end
9      end
10     H = G.subgraph(nodeSuccessors)
11     if H.has_node(target) then
12         Flow(target, node, H)
13 end
```

Scope in Threads : At application startup, a Zygote process forks the new application process, which then creates an instance of the runtime environment and maps all needed shared libraries. The runtime instance then creates the main application thread and the

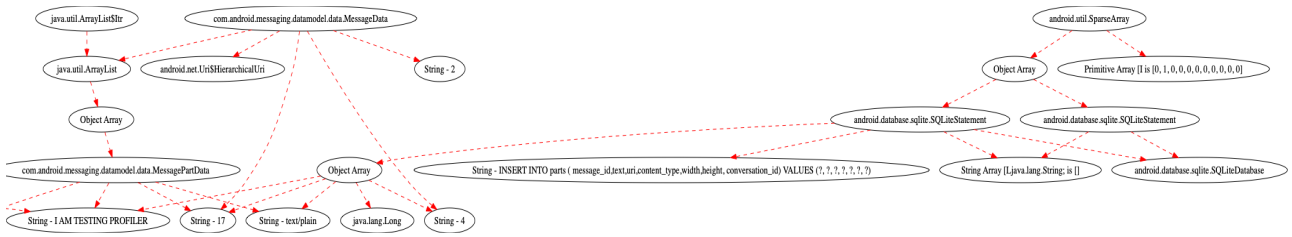


Figure 2: Illustrating context determination from a target object to a topmost root node with a recursive search depth of ten.

heap object, amongst other components. The runtime adds the main thread to its thread listing structure and transfers control to it. At any time during the program execution, new threads may be created and added to the thread listing.

To find the scope of a target object in the available threads in the memory image, we walk the thread listing pointer of the runtime instance and find all the user threads. For each user thread, we find the instance of *java.lang.Thread*. The *opeer* member of a thread object holds the reference to the Java thread. Using Algorithm 4, each Java thread instance from the thread listing is added to a nodeList. Starting at each Java thread, we recursively search for its successor nodes and add them to the nodeSuccessors list. When there are no more nodes to add, we return the nodeSuccessors list and use it to create a subgraph. Finally, the Path Exploration algorithm is used to find the flow paths between the target node and the Java thread node in the new subgraph.

Scope in Program Component : Application components are a fundamental building block of application development in the Android platform. The four major Android application components are Activity, Service, Content Provider, and Broadcast Receiver.

Android maintains a ClientRecord class that holds the record of all the local running Activities and Content Providers. The runtime class instance ActivityClientRecord and ProviderClientRecord corresponds to single instances of Activities and Content Providers, respectively. The information about currently activated Services and Receivers components created by an Intent, on the other hand, are provided using the ServiceDispatcher and ReceiverDispatcher objects, respectively. Thus, we created a Component function that searches the OAG for all the instances of these four classes and returns their corresponding nodes into a nodeList. Using the nodes in the nodeList and Algorithm 4, we apply the same steps we used in finding the scope of a target object in threads to find the scope of a target object in the Android components.

Scope in Garbage Collection Roots: A special kind of object that holds references to objects in a heap and can be accessed from outside the heap is called the Garbage Collection Root (GCRoot) [12]. GCRoots are crucial objects in the heap allocation chain because they determine whether an object stays or is disposed of by the garbage collector. Every GCRoot maintains a retained set of objects for which each individual or sets of objects are only marked for collection when they cannot be traced back to the GCRoot. In the new Android Runtime (ART), GCRoots are classified as either global, local, or weak references.

The runtime environment holds GCRoot references in an Indirect Reference Table -*IRT* structure. The *IRT* is a table of pointers with each entry holding a unique *gc-root* pointer for a reference. As part of the runtime instance, two key data structures *JavaVM* and *JNIEnv* are defined to hold the *IRT* structures for all the available GCRoots for the process. Every Android process has only one active *JavaVM*, which holds the *IRT* structures for global and weak global references, while *JNIEnv* holds *IRT* structures for the local references allocated per thread-local storage. By design, threads needing to share references must export them as globals. Thus, global references tend to stay longer in memory until they are explicitly deleted, or the process is terminated. To find the scope of a target object from the GCRoots, we created a GCRoot function that walks the global *IRT* structure and creates a list of GCRoot references. We then utilized Algorithm 4 to find the flow path between a target object and any of the references in GCRoot.

4 IMPLEMENTATION AND EVALUATION

OAGen is written in Python and implemented as a stand-alone memory forensics tool. We utilized some of the base-class data structures for Runtime, Heap, and Thread objects discovered in DroidScraper [2] to identify these structures and their addresses. *OAGen*'s implementation has five main components, each implemented as a module.

The first module is the OAG plugin that takes an Android process memory image as input and outputs a Python dictionary representing a directed graph with nodes and edges. Each node in the graph represents an allocated object, and an edge represents the allocation relation. The output of the OAG module, which is written to a file, is used as an input to the other four components. This module has the option to visualize the graph.

The second module is the Search Utility module that takes the graph file as input and an option to either dump all strings, search for a particular string, or search for an object with its fully-qualified class name.

The third module is the Path Exploration module, which also takes the graph file as input, together with source and target nodes (object references). The output consists of all the nodes that existed in the flow path between source and target. This module also plots the flow path from source to target.

The fourth module is for context generation. This module also takes the main graph as input, together with the target object, and returns a subgraph with all the adjacent nodes along the path of the target to its topmost root node. The module also has the option to visualize the subgraph.

The fifth module is for scope determination. It takes the graph as input together with the target object and one of the three options (GCRoot, Threads, Component). It processes the option first and returns the output in a node list. It then utilizes the path exploration module to find if a path exists between the target and any of the nodes in the node list.

4.1 Evaluation

Setup : *OAGen* is evaluated across fourteen memory images collected from six regular Google Play [20] apps (Facebook, WhatsApp, Signal, EvolveSMS, Keeper, and Chrome) as representative for different categories, including chat apps, social media and vault apps and one system app (Messaging). We also added seven malware samples downloaded from VirusShare [46] to the evaluation as representative of malicious applications. We set up a Samsung Galaxy S8 on Genymotion [19] with emulated GPS location, IMEI, Android ID, a Gmail account, Contacts, and SMS to simulate an instance of a real device. The snapshot of this initial setup is referred to as the clean state. Each app is downloaded, installed, activated, exercised, and its memory is captured on an instance of a clean state. We utilized memfetch [49] as a per-process memory capture utility. However, it is important to note that *OAGen* can work on a variety of Android process memory images captured using different utilities such as Volatility's Memdump plugin [18], AMD [48] - a live main memory acquisition method that relies on data in the recovery partition or PASM [16] - an Android application memory data acquisition method that uses a system-level data migration function.

The objectives of our evaluation are mapped directly to the contribution of this work. First, we seek to test the robustness of the OAG network in generating the relationships in the allocation chain. Second, we seek to evaluate the effectiveness of the semantic analysis modules in performing analysis. Notably, we want to experiment with the use of the developed algorithm to explore allocation paths, find object contexts, and determine the scope of allocated objects. We demonstrated the application of our approach in post-execution semantic analysis components in two case studies.

4.1.1 Performance of OAG Generation. In this experiment, we measure the performance of our algorithm in constructing the OAG across fourteen process memory images as well as evaluate the total processing time. We utilized a MacBook Pro with a 2.6GHz Intel processor and 16GB RAM.

Processing Time : As shown in Table 1, WhatsApp (com.whatsapp) has the largest allocation network with 328,896 nodes and 459,196 edges. com.yxxinglin.xzid has the smallest allocation network, with about 35,832 nodes and approximately 45,294 edges. The size of the memory image and the number of allocations linearly increased the time for the recursive generation of the network. Figure 3 also shows an increase in processing time as the number of nodes increases, which also corresponds to increase in the number of edges. For instance, it takes about 847 minutes to recursively search and add the 282,120 nodes from a 1733MB Facebook (com.facebook.katana) memory image and create all the 411,522 edges. As a performance measure, the throughput of the OAG generation algorithm on our test machine is approximately 0.19Mbps. This rate represents the efficiency of our algorithm to recursively search for an allocated object in a 0.19MB (185KB) memory image, decode the object, find all of its references, add them as nodes in the graph, and create edges between them all in one second. Thus, on our test environment with 2.6GHz speed and 16GB RAM, we consider this throughput to be satisfactory. Appendix A provide a visual illustration of a complete Object Allocation Graph for a test application with 3977 nodes and 6609 edges.

Macro Benchmark : In other to evaluate the robustness of *OAGen* to accurately add and decode nodes in the OAG, we compare the reconstructed graph against a known ground truth on a smaller benchmark. Specifically, we chose three sample memory images with large, medium, and small allocations - com.whatsapp, com.android.messaging, and com.yandex226.yandex respectively. We compare the number of nodes added and decoded by *OAGen* to the output of the HeapDump utility in DroidScraper as well as the cumulative total objects count from each heap region as provided by the *objects_allocated_* field in the Android heap region structure. This macro benchmark result showed that for the same memory, dump *OAGen* added 318,632 nodes to com.whatsapp OAG compared to 301,467 recovered objects for DroidScraper and 301,742 from the *objects_allocated_* field. For com.android.messaging, *OAGen* added 106,304 nodes, compared to 90,053 for DroidScraper and 92,046 for the *objects_allocated_*. In the memory image with much lower allocations, *OAGen* recovers 35,528 nodes compared to 10,328 for DroidScraper and 11,172 for *objects_allocated_*. The reason *OAGen* recovered and added more nodes than DroidScraper and the *objects_allocated_* is because the Points-to analysis utilized by our algorithm follows pointer relationships, irrespective of whether the object resides in the heap, stack, initialized data or the code section of the memory. The only requirement is that the new object is referenced by an object that resides in the heap. In addition to the nodes of the graph, *OAGen* generated thousands of edges in each memory image to show the relationships between objects. This is a unique and novel feature of *OAGen* that is not available in the existing tools and literature.

4.1.2 Semantic Analysis - Case Studies. The goal of this experiment is to illustrate the application of *OAGen*'s semantic analysis algorithms in real forensics investigations and/or security analysis.

Cyber Crime Investigation : In recent years there has been an increasing trend by terrorist networks in utilizing end-to-end, fully encrypted applications as communication channels. In 2017, the Middle East Media Research Institute published an article that detailed how WhatsApp is specifically gaining popularity amongst extremist groups and their supporters [43]. From its ease of use and secure message delivery and storage, extremist groups are using WhatsApp for messaging and dissemination of news and propaganda materials.

In this experiment, we created a scenario using WhatsApp, where a suspect who has been on the radar was trailed. Furthermore, right before the suspect was arrested, he sent a WhatsApp message to a co-conspirator with details about a potential attack. The message was then deleted almost immediately using the *"Delete from Everyone"* option available in the WhatsApp platform. The suspect,

Table 1: The node- and edge- list of the object allocation graph for the sample applications as generated by *OAGen*.

Applications	Image Size (in Mb)	Number of Nodes	Number of Edges	Processing Time (in Minutes)	Throughput (in Mbps)
com.whatsapp	1338	318632	476586	1025.5	0.03
com.facebook.katana	1733	282120	411522	847	0.04
com.callpod.android_apps.keeper	1237	227776	330240	899.5	0.03
org.thoughtcrime.securesms	1249	150824	229308	381.5	0.06
com.peaksel.gothicwallpapers	1272	128256	183276	206.5	0.11
com.klinker.android.evolve_sms	1208	108248	154014	280	0.08
com.android.messaging	1257	106304	156318	311.5	0.07
com.dd.monkey	1169	80512	116472	133	0.15
com.android.tencent.zdevs.bah	1192	59232	81462	157.5	0.13
com.easyhin.usereasyhin	1192	57504	85212	63	0.32
com.husor.beibei	1211	39320	50964	63	0.33
org.chromium.webview_shell	1289	38560	50070	45.5	0.48
com.yandex226.yandex	1201	38528	49590	56	0.36
com.yxxinglin.xzid	1195	35832	45294	49	0.41

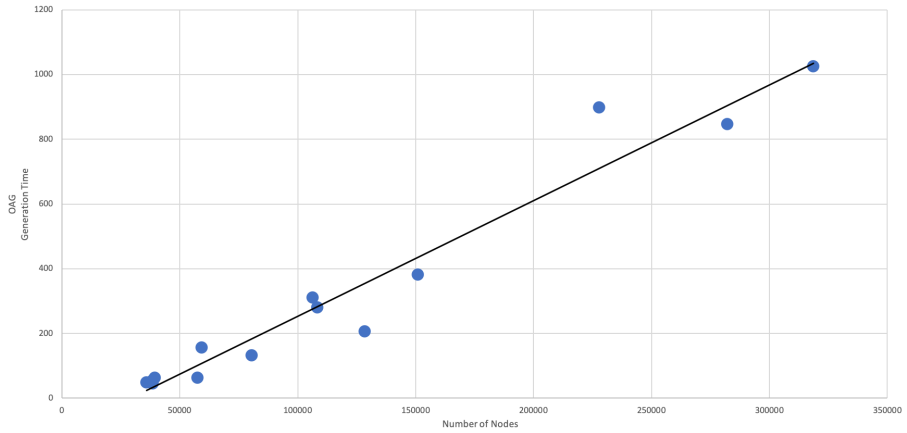


Figure 3: A graph illustrating the linear relationship between the number of nodes generated in each network and the processing time per network.

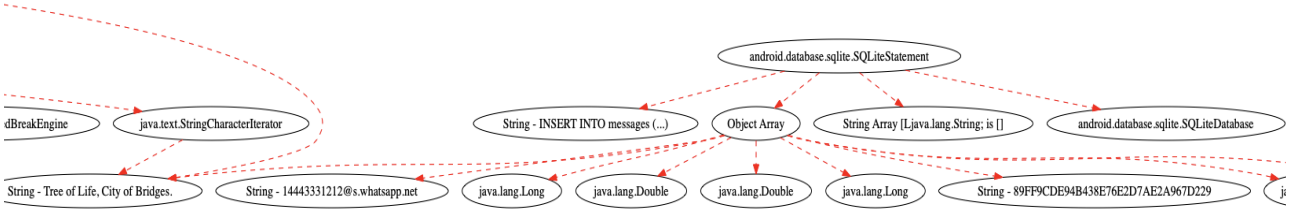


Figure 4: Partial graph showing a broader context of the suspected string content with SQL insert statement, message recipient and the hash of the message.

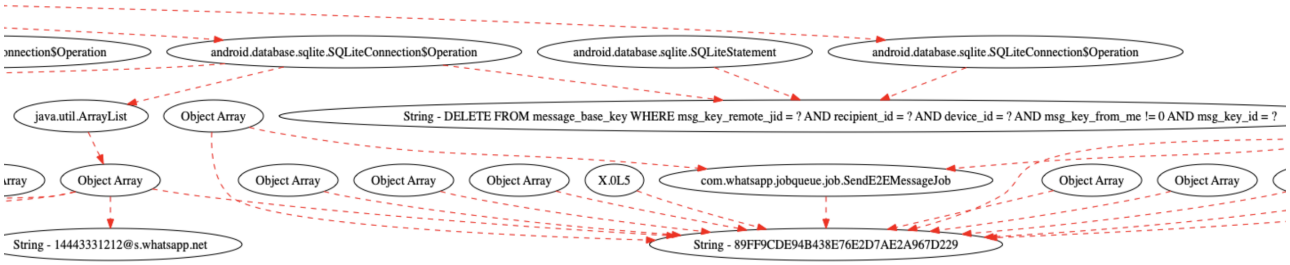


Figure 5: Partial graph showing the SQLiteConnection operation record for the message deletion with the valid message key.

who is willing to give up his life for the cause, refuses to cooperate. Investigators urgently need to find the message content to stop the attack as well as use it as admissible evidence in court.

The investigators have a memory imaging utility for Android, such as [16, 48]. Investigators then begin to process the WhatsApp process data by generating the OAG of the process memory dump. After the completion of this initial phase, they then use the Search Utility with the option to dump all the string objects with their memory locations. They search through the strings and identified a couple of chat messages that were not deleted (as evidence from the correlated chats on disk), but they also found suspected human-readable content that looks like a message. The content reads, "Tree of Life, City of Bridges." They then use the Search Utility again with the option to search a target string. They found 42 occurrences of the suspected content. While the text itself may be a red flag, without knowing its provenance or context within the WhatsApp process, the investigators don't know the recipient of the message, and the simple textual evidence by itself may not be admissible in court. Thus, to give context to the suspected content, the investigators use the *OAGen*'s Context determination option with a depth of 10 to get one subgraph for each of the 42 instances of the string. More than half of the 42 subgraphs are within the context of sun.misc.cleaner. The others were within the context of the program GUI component, except for one instance for which its broader context includes a SQL statement that seems to insert suspected chat into the Message table, as shown in Figure 4. From the graph, we can see that the insertion is also associated with what seems to be a hash value (89FF9CDE94B438E76E2D7AE2A967D229).

The investigators use this suspected hash value for the second round of context generation. They find 51 instances of the hash value, and only two instances seem viable with more than two edges associated with them. They explore the broader context and discover that in one of the instances, the hash was used as part of the Delete Query Where clause, as shown in a cropped version of the subgraph plot in Figure 5. Both the suspected chat content and the hash value in the Delete query are associated with the same message recipient. Hence, with the semantic analysis capability provided by *OAGen* , investigators find the message and prove that the content was sent by the suspect to his co-conspirator before it was deleted to destroy the evidence. Unlike existing work that recovers only the text data, *OAGen* goes a step further to determine the origin and context of the evidence within the running process. The recovered contextual content can be used to stop the attack before it happened and provide solid evidence for use in court. Given that the WhatsApp process has allocated more than 350,000 objects with about 53,000 strings, *OAGen* was essential for quickly zeroing in on the important evidence.

Detecting Data Exfiltration : Mobile applications are notorious for mishandling and misusing private user data. There is documented evidence that Android malware steals and exfiltrates various device and user data from mobile devices without consent [4, 31]. Furthermore, Android malware continuously evades detection from existing analysis techniques by utilizing various obfuscation mechanisms such as encryption, Java reflection, and dynamic class loading.

In this case study, we run a 2015 variant of the Plankton malware that heavily used Java reflection to hide the request for device and user data. The recreated OAG for the malware process has 128,256 nodes representing the allocated objects and 183,276 edges for the object relations. For better illustration, we limit this case study to tracking network-based data exfiltration from the remnants of in-memory objects. We begin the analysis by searching the graph for the presence of sink objects, which are generally the containers used to send data across the network. We found four instances of DefaultHttpClient (which is the default implementation of the HttpClient object) and its associated DefaultClientConnection object within the scope of the RefQueueWorker thread. Exploring the path of one of the DefaultHttpClient objects, we found its HttpParameters and Connectivity manager. Further pruning of the Client connection using the Context determination algorithm showed the allocation of a URI object which is associated with a *my.mobfox.com* host, a *request.php* path, an *http* scheme and a URI query that consists of different data including 818693586880159, the device version (8.0.0), build (OPR6.170623.017), etc. To find where the number 818693586880159 came from, we utilized the String search utility and found eight instances of the string. Exploring one of the instances using context determination showed that the string is allocated at address 0x12d40168, which is in the broader context of the com.android.internal.telephony.ITelephony$Stub$Proxy at address 0x12d40158. Thus this showed that the string is the device IMEI, which by the standard is requested using the TelephonyManager.getDeviceId() function. By creating the subgraph of these allocations, as shown in Appendix B, with URI as the target and

RefQueueWorker as the thread object using the Scope determination algorithm with a depth of fifty, an analyst can prove that the malware did indeed try to steal the device ID.

One may argue that a simple string search may be enough in some cases to get the query string, but nonetheless, knowing the provenance and context within which the query string exists is undoubtedly very important in documenting the malware's action. Besides, even in the absence of the source code, based on the subgraph alone, the analyst can eliminate the other six instance methods of the HTTPClient class [17] and establish that the malware called the execute() function on the HTTPClient object with a HttpURIRequest parameter. While dynamic analysis techniques such as TaintDroid [15] have proven effective in tracing data exfiltration and can offer similar insights, due to the difficulty in developing sound taint analysis systems, especially where Java reflection and/or the use of dynamic code is involved, these tools are rarely maintained. As an example, the last revision of TaintDroid occurred in 2014.

4.1.3 Discussion and Future Work. In this evaluation, we have shown that *OAGen* can indeed be of immense benefit to analysts and investigators in finding evidence with only a post-execution process memory image. We have shown the robustness of *OAGen* in exploring relationship paths, context, and scope, even for a very dense object allocation network.

In comparison with other userland memory forensics techniques such as Timeliner[6], which targets the recovery of Android activities, and GUITAR[36], which targets the reconstruction of Android GUIs using brute-force signature scanning of the windowing data structures, *OAGen* is a more generic and elaborative technique that goes beyond the reconstruction of a very specific application construct. It provides a flexible approach for analysts to explore and efficiently prune for different data classes, establishes data provenance, execution flow paths, and the context and scope within which the data exists. Furthermore, because *OAGen* is built on an in-depth understanding of Android's memory management algorithm, it can find and utilize partial and deallocated data structures as long as the memory region is not evacuated. This is a very distinct design difference compared to GUITAR, in which scanning algorithms conservatively discard objects with partially corrupted data fields.

Nevertheless, *OAGen* has some limitations, which as part of future work, we will implement. Some of these limitations include;

- More tests need to be carried out to explore the effects of program obfuscation, such as encryption on the network.
- Improvement in OAG generation time. We are working on better threading and an enhanced recursive algorithm that will shorten the current OAG generation time.
- We also believe that creating dynamic graphs where nodes can be explored with a cursor can improve the usability of this system.
- In order to find a tentative execution path for a program, we are currently working on recovering program code from memory and then mapping it to the OAG. This idea will significantly improve our context and scope determination algorithms.
- Utilizing machine learning to examine adjacently allocated objects that have no direct flow to a target may prove useful in recovering and finding relationships with local variables and other remote objects.

5 RELATED WORK

In this paper, we proposed a userland memory forensics technique that generates a runtime objects' relationship network from a post-execution process memory image. Utilizing this relationship graph, we developed three categories of algorithms for path exploration, context, and object scope determination that can be employed by investigators in performing in-depth semantic analysis of a target memory image.

5.1 Userland Memory Forensics

In 2011, [7] made the argument for the importance of utilizing data structures recovered from a process runtime environment in memory forensics. Using the Android Dalvik virtual machine as a use case, this study demonstrated that it is possible to replicate the features of the Android framework to find instances of structured classes and fields from the process memory dump. This line of argument was further explored by[2, 32, 34, 35, 42] to recover and reconstruct in-memory forensics artifacts from different runtime environments that are built using different memory management algorithms. [32] extended the work of [7] to cover a newer version of Android DVM. The papers[42] and [2] presented techniques for the volatile data extraction from the Android runtime (ART) heap space that is based on the reconstruction of the RosAlloc and Region-Based memory allocation algorithms. Pridgen et al.[34, 35], on the other hand, discuss a forensic analysis framework called RecOOP that works to recover managed memory objects from the HotSpot Java VM. Built on the premise of these pioneering studies and in particular [2], we argue that it is not sufficient to simply recover and reconstruct remnants of data from process memory. This is mostly because threats from mobile applications continue to grow, and the data generated from process memory increases exponentially in size and complexity. For practical and effective semantic analysis, we will need a more efficient and robust mechanism to understand the recovered objects' allocation paths, program context associated with objects, and the scope within which the object was manipulated.

Other specialized scenario-based userland memory analysis were published in the works of [6, 36–39]. Saltaformaggio et al., presents a system called DSCRETE that allows automatic interpretation and restructuring of data structure contents from process memory using an already extracted application execution logic for the target application[39]. This work, while very close in purpose to *OAGen* , however, is application-dependent and requires prior knowledge of the application execution logic, which is not feasible in some investigations. Saltaformaggio et al., [36] proposed an app-agnostic GUI reconstruction methodology. Their system, titled GUITAR, utilizes the low-level definition of the Android GUI framework for reassembling partial GUIs from the smartphone's memory image. The same authors further proposed VCR - a photographic evidence recovery from mobile device memory image[37]. In 2016, RetroScope was

presented in [38]. Using spatial-temporal forensics, RetroScope recovers and renders previous screens of an Android application from a mobile device memory image. Like GUITAR, RetroScope requires no prerequisite knowledge of the application that it is recovering or reconstructing. In 2018, Bhatia et al. discuss a novel memory forensics tool that addresses the challenge of the reconstruction of a timeline of user activity[6]. The authors developed a tool called Timeliner to reconstruct the past Android Activities of users across a wide range of applications from a mobile device memory image. Much like this related work, *OAGen* is also app-agnostic, however with an entirely different objective, *OAGen* is not limited to specific scenarios. It is generically designed to aid analysts and investigators to examine the Android post-execution memory image for any process and can find different allocation paths from different objects, identify multiple contexts, and determine scopes for different objects.

Furthermore, much different from the related work, our generic object allocation network and the semantic analysis algorithms can be applied in different security and digital forensics examinations, ranging from user activity exploration in cybercrime investigations, the identification of malware behavior by retracing object allocation paths, enumerating execution paths in symbolic and concolic execution, to data tracing in fuzzing analysis.

6 CONCLUSION

As userland memory forensics continues to be a practical and crucial alternative to kernel-level memory forensics and traditional disk forensics in mobile program analysis and forensics investigations, the need to develop better and enhanced techniques cannot be overemphasized. The limitations of the current state-of-the-art userland memory forensics approaches to scenario-specific data recovery, such as images and GUIs, make them inapplicable to a wide variety of investigations.

Thus, in this paper, we present *OAGen* - an approach that uses Heap Context Points-to analysis for post-execution semantic investigation of an Android process memory image. *OAGen* creates a graph of the runtime's object allocation network (OAG) that represents the relationships between objects and their allocators. The implementation establishes objects' references as nodes and the relationships between them as edges. Utilizing the OAG for semantic analysis, we developed three algorithms that explore in depth allocation paths for a target object, identify the object's broader context, and determine its scope of execution in the allocation network. *OAGen* was tested against fourteen different applications. The results showed that *OAGen* took an approximate 0.19Mbps on a modest system to search and decode a node, and locate all of its references in a process memory image. Further evaluation of *OAGen* in two case studies illustrates its practical application in a post-mortem recovery of deleted messages in the WhatsApp application and the detection of data exfiltration for a malware sample.

ACKNOWLEDGMENTS

This work is supported by the National Science Foundation under Grant Number 1850054.

REFERENCES

[1] Aisha Ali-Gombe. 2019. DroidScraper. https://github.com/apphackuno/DroidScraper [Online; accessed 10-January 2018].
[2] Aisha Ali-Gombe, Sneha Sudhakaran, Andrew Case, and Golden G Richard III. 2019. DroidScraper: A Tool for Android In-Memory Object Recovery and Reconstruction. In *22nd International Symposium on Research in Attacks, Intrusions and Defenses ({RAID} 2019)*. 547–559.
[3] Aisha Ibrahim Ali-Gombe. 2012. *Volatile Memory Message Carving: A" per process basis" Approach.* Master's Thesis. University of New Orleans, LA.
[4] Aisha I Ali-Gombe, Brendan Saltaformaggio, Dongyan Xu, Golden G Richard III, et al. 2018. Toward a more dependable hybrid analysis of android malware using aspect-oriented programming. *computers & security* 73 (2018), 235–248.
[5] Cosimo Anglano, Massimo Canonico, and Marco Guazzone. 2017. Forensic analysis of telegram messenger on android smartphones. *Digital Investigation* 23 (2017), 31–49.
[6] Rohit Bhatia, Brendan Saltaformaggio, Seung Jei Yang, Aisha I Ali-Gombe, Xiangyu Zhang, Dongyan Xu, and Golden G Richard III. 2018. Tipped Off by Your Memory Allocator: Device-Wide User Activity Sequencing from Android Memory Images.. In *NDSS*.
[7] Andrew Case. 2011. *Memory analysis of the dalvik (android) virtual machine.* Source Seattle.
[8] Andrew Case, Mohammad M Jalalzai, Md Firoz-Ul-Amin, Ryan D Maggio, Aisha Ali-Gombe, Mingxuan Sun, and Golden G Richard III. 2019. HookTracer: A System for Automated and Accessible API Hooks Analysis. *Digital Investigation* 29 (2019), S104–S112.
[9] Andrew Case and Golden G Richard III. 2015. Advancing Mac OS X rootkit detection. *Digital Investigation* 14 (2015), S25–S33.
[10] Andrew Case and Golden G Richard III. 2016. Detecting objective-C malware through memory forensics. *Digital Investigation* 18 (2016), S3–S10.
[11] Andrew Case and Golden G Richard III. 2016. Memory forensics: The path forward. *Digital investigation* (2016), 1–11.
[12] IBM Knowlegge Center. 2015. Garbage collection roots. https://www.ibm.com/support/knowledgecenter/en/SS3KLZ/com.ibm.java.diagnostics.memory.analyzer.doc/gcroots.html
[13] Yoan Chabot, Aurélie Bertaux, Christophe Nicolle, and Tahar Kechadi. 2014. Automatic timeline construction and analysis for computer forensics purposes. In *2014 IEEE Joint Intelligence and Security Informatics Conference*. IEEE, 276–279.
[14] Jusop Choi, Jaewoo Park, and Hyoungshick Kim. 2017. Forensic analysis of the backup database file in KakaoTalk messenger. In *2017 IEEE International Conference on Big Data and Smart Computing (BigComp)*. IEEE, 156–161.
[15] William Enck, Peter Gilbert, Seungyeop Han, Vasant Tendulkar, Byung-Gon Chun, Landon P Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N Sheth. 2014. TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones. *ACM Transactions on Computer Systems (TOCS)* 32, 2 (2014), 1–29.
[16] Peijun Feng, Qingbao Li, Ping Zhang, and Zhifeng Chen. 2019. Private Data Acquisition Method Based on System-Level Data Migration and Volatile Memory Forensics for Android Applications. *IEEE Access* 7 (2019), 16695–16703.
[17] The Apache Software Foundation. 2020. Interface HttpClient. https://hc.apache.org/httpcomponents-client-ga/httpclient/apidocs/org/apache/http/client/HttpClient.html [Online; accessed 1-June 2020].
[18] Volatility Foundation. 2017. Volatility Command Reference. https://github.com/volatilityfoundation/volatility/wiki/Command-Reference#memdump [Online; accessed 21-March 2018].
[19] Genymotion. 2019. Genymotion Desktop. https://www.genymotion.com [Online; accessed 10-January 2020].
[20] Google. 2019. Google Play. https://play.google.com/store?hl=en
[21] George Grispos, William Bradley Glisson, and Tim Storer. 2015. Recovering residual forensic data from smartphone interactions with cloud storage providers. *arXiv preprint arXiv:1506.02268* (2015).
[22] Kristinn Guðjónsson. 2010. Mastering the super timeline with log2timeline. *SANS Institute* (2010).
[23] Aric Hagberg, Pieter Swart, and Daniel S Chult. 2008. *Exploring network structure, dynamics, and function using NetworkX.* Technical Report. Los Alamos National Lab.(LANL), Los Alamos, NM (United States).
[24] Christopher Hargreaves and Jonathan Patterson. 2012. An automated timeline reconstruction approach for digital forensic investigations. *Digital Investigation* 9 (2012), S69–S79.
[25] Andrew Hoog. 2011. *Android forensics: investigation, analysis and mobile security for Google Android.* Elsevier.
[26] MNA Khan and Ian Wakeman. 2006. Machine learning for post-event timeline reconstruction. In *First Conference on Advances in Computer Security and Forensics, Liverpool, UK*. Citeseer.
[27] Jon Kleinberg and Éva Tardos. 2005. *Algorithm Design.* Pearson.
[28] Jeff Lessard and Gary Kessler. 2010. Android Forensics: Simplifying Cell Phone Examinations. (2010).

[29] Alex Levinson, Bill Stackpole, and Daryl Johnson. 2011. Third party application forensics on apple mobile devices. In *2011 44th Hawaii International Conference on System Sciences*. IEEE, 1–9.

[30] Nathan Lewis, Andrew Case, Aisha Ali-Gombe, and Golden G Richard III. 2018. Memory forensics and the Windows Subsystem for Linux. *Digital Investigation* 26 (2018), S3–S11.

[31] Yuping Li, Jiyong Jang, Xin Hu, and Xinming Ou. 2017. Android malware clustering through malicious payload mining. In *International Symposium on Research in Attacks, Intrusions, and Defenses*. Springer, 192–214.

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

[33] Ana Milanova, Atanas Rountev, and Barbara G Ryder. 2002. Parameterized object sensitivity for points-to and side-effect analyses for Java. In *Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis*. 1–11.

[34] Adam Pridgen, Simson Garfinkel, and Dan Wallach. 2017. Present but unreachable: reducing persistentlatent secrets in hotspot jvm. (2017).

[35] Adam Pridgen, Simson Garfinkel, and Dan S Wallach. 2017. Picking up the trash: Exploiting generational GC for memory analysis. *Digital Investigation* 20 (2017), S20–S28.

[36] Brendan Saltaformaggio, Rohit Bhatia, Zhongshu Gu, Xiangyu Zhang, and Dongyan Xu. 2015. GUITAR: Piecing together android app GUIs from memory images. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. 120–132.

[37] Brendan Saltaformaggio, Rohit Bhatia, Zhongshu Gu, Xiangyu Zhang, and Dongyan Xu. 2015. 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*. 146–157.

[38] Brendan Saltaformaggio, Rohit Bhatia, Xiangyu Zhang, Dongyan Xu, and Golden G Richard III. 2016. Screen after previous screens: Spatial-temporal recreation of android app displays from memory images. In *25th {USENIX} Security Symposium ({USENIX} Security 16)*. 1137–1151.

[39] Brendan Saltaformaggio, Zhongshu Gu, Xiangyu Zhang, and Dongyan Xu. 2014. {DSCRETE}: Automatic Rendering of Forensic Information from Memory Images via Application Logic Reuse. In *23rd {USENIX} Security Symposium ({USENIX} Security 14)*. 255–269.

[40] Bradley Schatz, George Mohay, and Andrew Clark. 2004. Rich event representation for computer forensics. In *Proceedings of the Fifth Asia-Pacific Industrial Engineering and Management Systems Conference (APIEMS 2004)*, Vol. 2. 1–16.

[41] Michael L. Scott. 2009. *The Java Native Interface: Programmer's Guide and Specification*. Morgan Kaufmann.

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

[43] Steven Stalinsky and R. Sosnow. 2017. Jihadi Use Of Encrypted Messaging App WhatsApp. https://www.memri.org/cjlab/jihadi-use-of-encrypted-messaging-app-whatsapp [Online; accessed 04-April 2020].

[44] Tian Tan, Yue Li, and Jingling Xue. 2016. Making k-object-sensitive pointer analysis more precise with still k-limiting. In *International Static Analysis Symposium*. Springer, 489–510.

[45] Cetus Team. 2004-2011. *The Cetus Compiler Manual*. ParaMount Research Group, Purdue University.

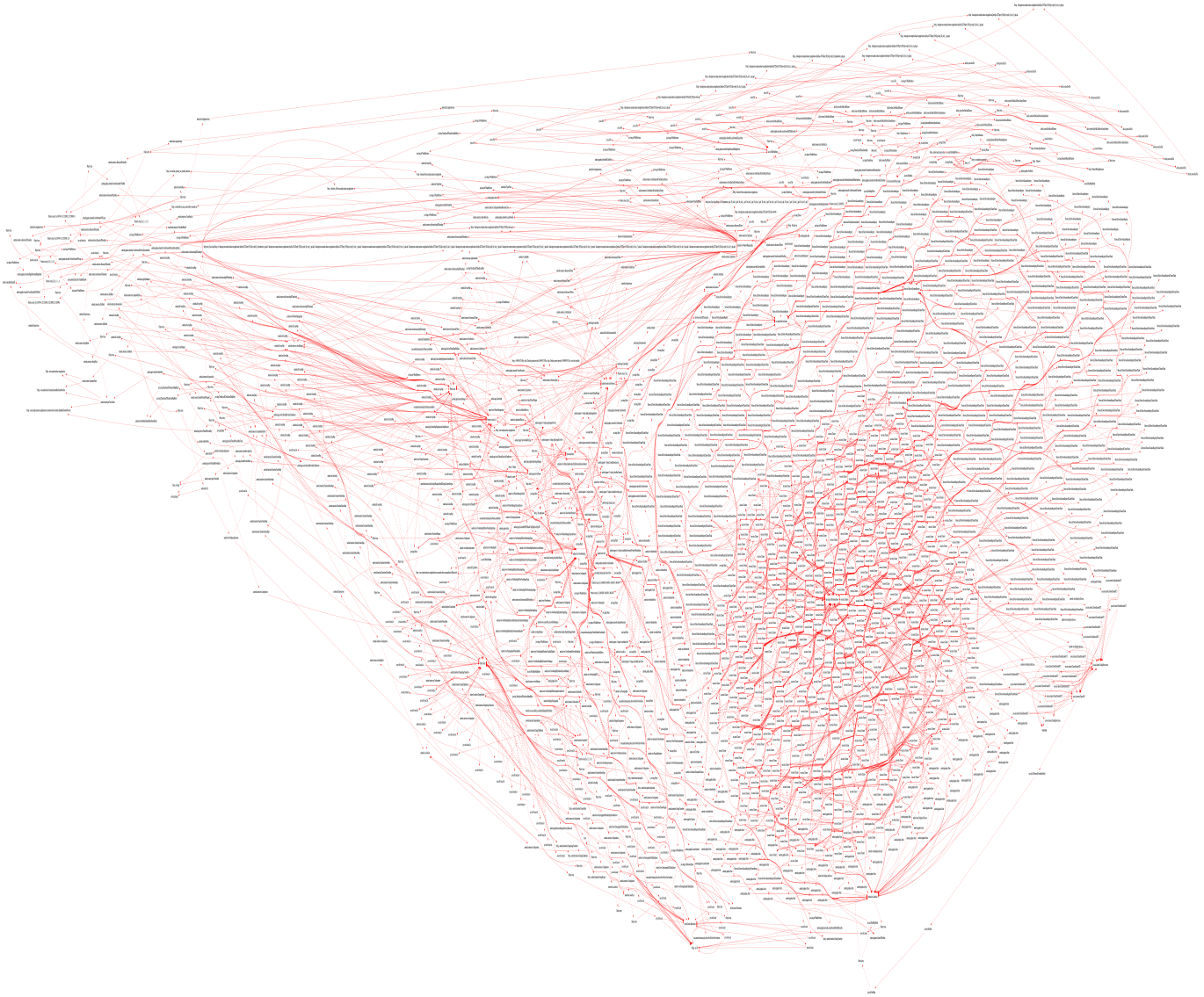
[46] VirusShare. 2017. VirusShare.com - Because Sharing is Caring. https://virusshare.com

[47] Wikipedia. 2020. Archaeology. https://en.wikipedia.org/wiki/Archaeology#cite_note-Society_for_American_Archaeology-1 [Online; accessed 1-June 2020].

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

[49] Michal Zalewski. 2003. Memfetch. https://github.com/citypw/lcamtuf-memfetch [Online; accessed 17-March 2018].

Appendix A. Object allocation graph for a sample application with 3977 nodes and 6609 edges.



Appendix B. Partial subgraph showing the broader context of the URI object and its relationship with the DefaultClientConnection.