"Tipped Off by Your Memory Allocator": Device-Wide User Activity Sequencing from Android Memory Images

Rohit Bhatia*, Brendan Saltaformaggio†, Seung Jei Yang‡, Aisha Ali-Gombe§,
Xiangyu Zhang*, Dongyan Xu* and Golden G. Richard III¶
*Purdue University, {bhatia13, xyzhang, dxu}@cs.purdue.edu
†Georgia Institute of Technology, brendan@ece.gatech.edu
‡The Affiliated Institute of ETRI, sjyang@nsr.re.kr
§Towson University, aaligombe@towson.edu
¶Louisiana State University, golden@cct.lsu.edu

***Abstract*—An essential forensic capability is to infer the sequence of actions performed by a suspect in the commission of a crime. Unfortunately, for cyber investigations, user activity timeline reconstruction remains an open research challenge, currently requiring manual identification of datable artifacts/logs and heuristic-based temporal inference. In this paper, we propose a memory forensics capability to address this challenge. We present Timeliner, a forensics technique capable of automatically inferring the timeline of user actions on an Android device *across all apps*, from a single memory image acquired from the device. Timeliner is inspired by the observation that Android app Activity launches leave behind key self-identifying data structures. More importantly, this collection of data structures can be temporally ordered, owing to the predictable manner in which they were allocated and distributed in memory. Based on these observations, Timeliner is designed to (1) identify and recover these residual data structures, (2) infer the user-induced transitions between their corresponding Activities, and (3) reconstruct the device-wide, cross-app Activity timeline. Timeliner is designed to leverage the memory image of Android's centralized ActivityManager service. Hence, it is able to sequence Activity launches across all apps — even those which have terminated. Our evaluation shows that Timeliner can reveal substantial evidence (up to an hour) across a variety of apps on different Android platforms.**

I. Introduction

One of the critical steps in a forensic investigation is deriving a timeline of a suspect's activities. As described in [33], this task "involves evaluating the context of a scene and the physical evidence found there in an effort to identify what occurred and in what order it occurred." In the physical world, this is often modeled as inferring causal and temporal relations between events involving the suspect(s) and victim(s).

In cyber investigations inferring a suspect's temporal sequence of activities on his/her mobile device remains a challenging problem. Currently, investigators must manually coalesce datable (often modifiable) forms of application-specific evidence: e.g., call/message databases [19], [20] or web browsing logs [21] saved on a mobile device's SD-card. Since no semantic links readily exist between these evidence sources, the activity timelines reconstructed in this way are often heuristic and incomplete at best. Further, while Android captures coarse-grained information about user actions, major Android phone manufacturers routinely disable these features [6].

To illustrate this challenge, consider the variety of mobile apps utilized in the commission of even a simple espionage crime: Upon receiving a go-ahead call from a conspirator, the criminal uses his smartphone camera to take photographs of sensitive documents and forwards them via a secure messaging app to the conspirator. Being wary of his safety, the criminal immediately deletes the photographs and terminates the messaging app. Finally, the criminal opens his banking app to verify the payment from his conspirator in his account. Each of these actions alone does not suggest a pattern of espionage, but the causal relationship between the various user actions (derived from their temporal ordering) indicates the commission of the crime. Unfortunately, in order to reconstruct the temporal sequence of these actions, investigators currently have to perform three manual steps: (1) recover application-specific evidence from the confiscated device, (2) infer temporal ordering relations among the user actions, and (3) derive a global timeline to reveal the causal relationships between user actions.

In this paper, we will show how Android memory forensics — performed on a single memory image *without* temporal logs — can achieve high-accuracy user/app action sequencing. Through an in-depth analysis of mobile app activity handling within Android, we have identified a set of *application-independent* in-memory artifacts which represent state and display changes across all applications. These artifacts, aptly named *Activities*, are generated and managed by the Android subsystem (specifically the ActivityManagerService) — putting them out of the reach of any app's execution and making their recovery and interpretation generic with regard to the app/user actions they represent. Further, each *Activity launch* that an app performs is unique and leaves behind a

Network and Distributed Systems Security (NDSS) Symposium 2018
18-21 February 2018, San Diego, CA, USA
ISBN 1-1891562-49-5
http://dx.doi.org/10.14722/ndss.2018.23324
www.ndss-symposium.org

signature, namely a collection of *residual data structures* that are indicative of a specific user action on the device.

Leveraging the power of app-generic Activities, we then turn our attention to the *automatic, device-wide temporal sequencing of app Activities*. Again, we glean clues from the Android subsystem's in-memory artifacts. By modeling the operation of the Android memory allocator, we found that these residual data structures are allocated memory locations (called "slots") in a sequentially increasing ordering. Put simply: if it is possible to recover the *spatial ordering* of these slots for a sequence of historic Activities, then we can perform inference of the specific *temporal ordering* of those Activities.

Inspired by these findings, we develop Timeliner, a memory forensics technique which automatically performs inference of an Android device user's past actions from Activities *across different apps (even those which have terminated)* found in one memory image. The development of Timeliner overcomes a number of challenges during the identification and subsequent temporal sequencing of Activities: (1) Frequent allocation and garbage collection (GC) induces fragmentation in memory, causing consecutive allocations to become non-contiguous, making identification and segregation of allocations into Activity launches difficult. (2) In a fragmented memory, identification of spatial ordering among the Activity-identifying residual data structures is also challenging. (3) Android's memory allocator utilizes thread-local buffers for small allocations, spreading some residual data structures across several memory locations, making the spatial ordering potentially ambiguous. (4) Long temporal gaps between Activity launches can lead to spatial gaps between their residual data structures, requiring Timeliner to join multiple separated spatial orderings.

To address these challenges, Timeliner works in three stages. First, Timeliner identifies and recovers all *residual data structures* (several hundred per Activity launch) from a subject Android memory image (Section III-A). This step is akin to an investigator first identifying as many crime-related events as possible. Next, like an investigator finding causal relationships between the crime-related events, Timeliner uses spatial ordering of the recovered Activity launches to infer transitions between pairs of Activities (Section III-B). Lastly, similar to an investigator reconstructing the expected timeline, Timeliner orders the pairwise transitions to derive the global ordering of Activities (Section III-B). Note that Timeliner does not compete with forensic tools that recover evidence based on content [51], [52], [58], [60], but instead complements them by providing contextual meaning to the evidence they recover by establishing a device-wide, inter-app temporal sequence of user actions.

We have evaluated Timeliner, using micro-benchmarks and recreations of real criminal investigations, across multiple commercially available Android phones and a wide range of applications covering messaging, voice calls, banking, email, file management, and video streaming. Our results show that Timeliner is highly accurate (our case studies recover as many as 18 prior device-wide Activities) and provides convincing evidence to investigators through the reconstructed timelines of user actions. We also show that the techniques behind Timeliner are neither limited to the Android platform (by applying Timeliner to the jemalloc allocator) nor to only user-actions (by applying Timeliner to inter-application interactions called Broadcasts).

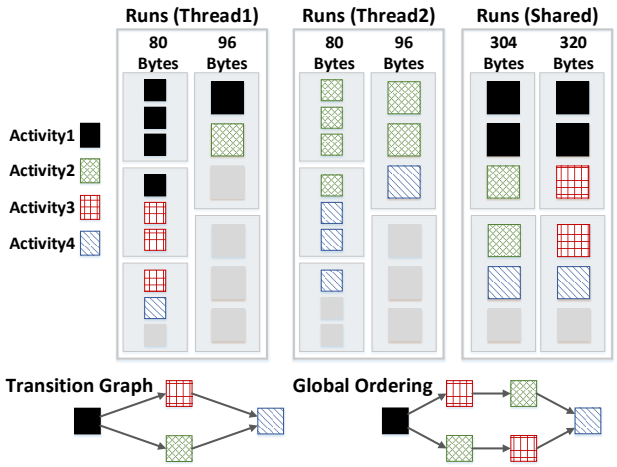

Fig. 1. Spatial Layout of Allocations in RosAlloc.

II. Background

Activity is the fundamental abstraction for a user action provided by the Android framework. In particular, an Activity is described as a "single, focused thing that the user can do" in the Android developer documentation [1]. To showcase how each Activity within an app models such a "single, focused thing," Table I in Section IV lists a number of Activities we encountered during our evaluation. As a user interacts with their device (e.g., clicking a button on the UI or receiving a call), the Activity corresponding to each action will be "launched".

Timeliner aims to reconstruct the temporal sequence of recent Activity launches that occurred on a subject device. To identify the launch of a specific Activity from an input memory image, Timeliner locates and recovers the unique data structures left behind from the execution of the Activity launch's logic. Then, to reconstruct the sequence of these Activity launches, Timeliner infers their temporal ordering from the data structures' spatial ordering (their allocation pattern in memory). We now discuss the enabling principles for these techniques.

A. Memory Allocator Design

Android's memory allocator is a "Run" of "Slots" allocator (named *RosAlloc*), where *slots* are individual memory locations for object allocations and a *run* is a list of slots of the same size. Like other Run of Slots allocators (e.g. phkmalloc, jemalloc), allocation is handled through a per-run bitmap. Each thread in a process manages its own thread-local runs for smaller slots, along with shared runs for larger slots. As an example, Figure 1 shows a few sample thread-local and shared runs of different sizes for two threads.

An allocation request is first assigned to the run whose slots best fit the requested size. A slot is chosen from this run based on a "first-available" algorithm which assigns it to the first empty slot picked from the bitmap, and subsequently the object is instantiated at this location. When the run is completely

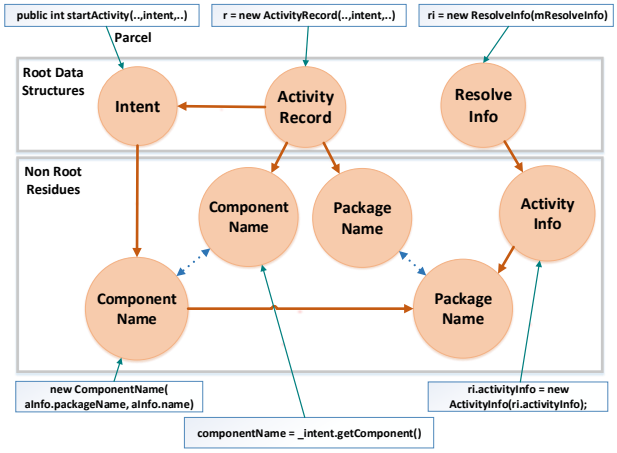

Fig. 2. Example Residual Data Structures Generated During Activity Launch.

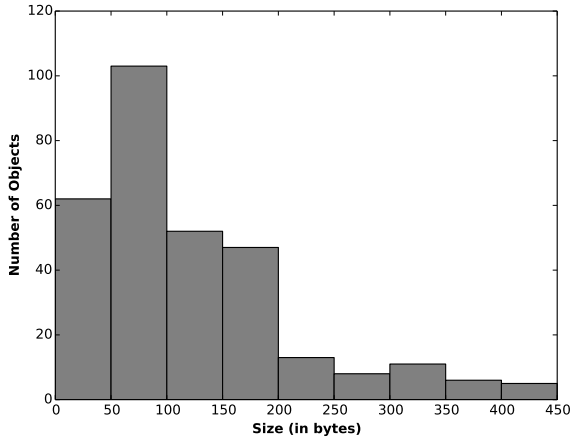

Fig. 3. Sample Size Distribution for Residual Data Structures.

filled, a new run is used, with a similar "first-available" algorithm choosing the run with the lowest address. These algorithms are deliberately designed to reduce fragmentation, preferring low addresses for allocations.

The implication of these "first-available" algorithms is as follows: If an allocation a (immediately) temporally precedes an allocation b of the same size, then a will be assigned a slot preceding b. Put simply, allocations that have a temporal ordering will be assigned memory locations that have a corresponding spatial ordering. This property is key to Timeliner, which attempts to solve the reverse problem: Identify the original temporal ordering of the allocations from their spatial ordering in a memory image.

B. Identifying an Activity Launch from Allocations

As an Activity is launched, the transition from the previous Activity to the newly launched Activity is centrally handled by the *ActivityManagerService*. As such, the ActivityManagerService receives several RPCs (Remote Procedure Calls) when an Activity launch takes place. While executing the RPCs, the ActivityManagerService will allocate several key data structure "clusters" (i.e., a network of interconnected data structures), hereinafter referred to as the residual data structures of an Activity. An example of this can be seen in Figure 2, where a specific Activity's Intent object is allocated as an RPC argument which links to other objects that are allocated through routine execution.

An important property of these residual data structures is that they are highly inter-connected. As the objects in residual data structures are allocated during the same Activity launch, they share a number of field values and are interconnected via pointers. This is highlighted in Figure 2, where the ActivityRecord, Intent, and ResolveInfo objects are required to share references (both directly and through their fields). There is another required value equivalence between the PackageName fields of ActivityRecord and ResolveInfo objects.

Another key property of these residual data structures is that they are organized as trees rooted at *application-generic* objects. This is noticeable in Figure 2, where the ActivityRecord, Intent, and ResolveInfo (application-generic) objects can be utilized to identify (and traverse) the entirety of the residual data structures. We call these top-most, application-generic data structures *root* data structures, and Timeliner utilizes 14 different root data structures to identify the residual data structures that are leftover from each Activity launch.

As shown in the sample size distribution in Figure 3, residual data structures are mostly composed of a large number of small allocations in thread-local runs, and a small number of large allocations in shared runs. Both types of allocations provide the spatial information utilized by Timeliner. While being limited to thread-local runs, the large number of allocations ensures that these allocations are spread out over several runs across various threads. Similarly, while fewer in number, the allocations in shared runs provide more robust spatial ordering. In this way, while RosAlloc's implementation includes both thread-local and shared runs, Timeliner's design is not dependent on it. We demonstrate this by extending Timeliner to another memory allocator (jemalloc) during our evaluation — which utilizes only thread-local runs.

Lastly, note that the lifespan of residual data structures is dependent on the Activity they represent. Further, the Activities belonging to the current Activity stack survive garbage collection. However, even those that survive get diminished in number. This is because many objects in the residual data structures are utilized only temporarily during Activity launch execution, making them candidates for garbage collection. Despite their reduced numbers, these *diminished* residual data structures are still recoverable and identifiable (i.e., they reveal the original Activity's name). This allows Timeliner to identify Activities that occurred before the last garbage collection, which we call *garbage collected Activities*.

C. Inferring Temporal Ordering from Spatial Ordering

As described above, each Activity launch generates residual data structures, produced through the execution of the launch logic. Figure 1 provides an example of the allocations from four Activity launches, where all slots of the same color represent the residual data structures of a single Activity launch. The most important property to note from the layout of the residual data structures is that they are of varying allocation sizes and therefore occupy slots in multiple runs, both thread-local and shared.

In Section II-A we described how *within a single run* the temporal ordering of allocations results in a corresponding spatial ordering. This is the first step to inferring the temporal ordering of the residual data structures from two Activity launches — enabling Timeliner to solve for the above principle (temporal ordering leads to spatial ordering) in reverse. Timeliner applies the above principle across multiple runs and recovers the temporal ordering for a pair of Activity launches, which we refer to as the *transition* between two Activities. Timeliner models the transitions as a directed edge starting from a node for the former Activity and directed towards a node for the latter. The nodes and edges for all Activities recovered from a memory image are organized into a *transition graph*. Figure 1 includes an illustration of the transitions between various pairs of Activities.

A timeline of Activities that satisfies the transition graph should satisfy each edge individually, i.e. for every transition $u \rightarrow v$, from Activity u to Activity v, u should occur before v in the timeline. This ordering, known as the *topological ordering* of a graph, allows Timeliner to solve for the timeline by topologically sorting the transition graph.

Unfortunately, within individual runs, fragmentation of memory (i.e., a new allocation fills a slot before an existing allocation) can mislead Timeliner's reconstruction. This will result in an erroneous edge originating from one Activity and pointing to another. Handling these misleading (i.e., incorrect) transitions will require Timeliner to prune such edges from the transition graph. Figure 4 describes two examples of such pruning.

Notice also that Timeliner's transition graph can entirely miss edges which should exist. This can happen in two ways: (1) If the current set of runs becomes filled, then the spatial ordering information inferred from those runs (i.e., one transition between two Activities) is lost. As a result, the transition graph is partitioned into several connected components, the temporal orderings of which are termed *local orderings*. Luckily, as the runs are chosen just like slots, i.e. with a "first-available" algorithm, the different local orderings can be joined later based on the spatial ordering of their runs into a single *global ordering*. (2) If two successive Activities do not share a common run, then there is no evidence of the transition between them. In this case, an ambiguity exists in their spatial ordering, and hence temporal ordering, leading to multiple possible timelines for those Activities. Hence, Timeliner needs to find all the topological orderings for the given transition graph. An example is shown in Figure 1, where there are two possible timelines.

III. Timeliner Design

Timeliner operates on only a single memory image from an Android device. From this memory image, Timeliner isolates and inspects the ActivityManagerService process's dynamic memory allocation space. Note that because Timeliner only relies on generic framework defined objects like Activities, Timeliner has no application-specific requirements in its design or implementation. Further, Timeliner's operation is entirely automated with no supervision required from an investigator — allowing it to be immediately deployable in practical investigations. In the remainder of this section, we will present the three phases of Timeliner's design.

Algorithm 1 Segregating Residual Data Structures.

Input: Object List O, RootClass List $Roots$
Output: ResidueObjectSet List $Residues$

▷ Identify and Add the Root Data Structures
Object List $rootObjs \leftarrow \varnothing$
for Object $o \in O$ **do**
  if $o.class \in Roots$ **then**
    $rootObjs \leftarrow rootObj \cup o$
▷ Segregate into different partial ResidueSets
ResidueObjectSet List $PartialResidues \leftarrow \varnothing$
for Object $rootObj \in rootObjs$ **do**
  ResidueObjectSet $newResSet \leftarrow rootObj$
  ▷ Match each root to identified partial ResidueSets
  for ResidueObjectSet $resSet \in PartialResidues$ **do**
    for Object $singleRes \in resSet$ **do**
      if $MATCH(rootObj, singleRes)$ **then**
        ▷ Merge partial residueSets for the same Activity
        $newResSet \leftarrow newResSet \cup resSet$
        $PartialResidues \leftarrow PartialResidues - resSet$
        break
  ▷ Add the new partial residueSet back to PartialResidues
  $PartialResidues \leftarrow PartialResidues \cup newResSet$
▷ Recurse to get ResidueSets
ResidueObjectSet List $Residues \leftarrow \varnothing$
for ResidueObjectSet $resSet \in PartialResidues$ **do**
  ResideObjectSet $fullResSet \leftarrow RECURSE(resSet)$
  $Residues \leftarrow Residues \cup fullResSet$

A. Identifying Residual Data Structures

As Section II-B introduced, highly-interconnected sets of objects called residual data structures are left over from the execution of past Activity launches. Therefore, Timeliner must first recover objects and segregate them into residual data structures. This procedure is shown in Algorithm 1. As a running example, we shall recall the structures shown in Figure 2 throughout this section.

Timeliner first scans the input memory image to identify all objects previously allocated by the ActivityManagerService, whether still active or deallocated but waiting for garbage collection ("dead objects"). This step is accomplished with the help of the runtime type information included in the managed runtime of Android (ART), which includes type information for objects and their fields. These are included in every process's memory space, so Timeliner can recover them directly. Note that Timeliner also recovers dead objects since Android's memory management is automatic and slots remain allocated until a garbage collection event. This list of recovered objects is given as the input to Algorithm 1.

Next, Timeliner parses this list of objects and identifies the root data structures. Defined in Section II-B, these data structures are crucial for the identification of residual data structures (the "fingerprints" left by Activity launches). In Figure 2, we can see that the Intent, ActivityRecord, and ResolveInfo objects are three instances of such root data structures. An important point to note is that these root data structures are highly inter-connected, and as such, they can be used to segregate the recovered objects into distinct residual data structures.

This is exactly the approach used in Timeliner, as explained in Algorithm 1, the list of root data structures is segregated into distinct partial residual data structures. Note that these resultant residual data structures (named ResidueSets in Algorithm 1) are called partial, as they do not (yet) include the various

non-root residual data structures reachable from the recovered instances of root data structures.

This segregation is affected by the "MATCH" function, which contains predefined application-generic relationships between the root data structures. In Figure 2, Intent and ActivityRecord are matched by their predefined ComponentName field values, and also a direct pointer from ActivityRecord to Intent. Further, ResolveInfo is linked to Intent via their predefined PackageName and ComponentName field values. Timeliner also leverages value equivalence in the PackageName fields of the ResolveInfo and ActivityRecord objects.

With these segregated partial residual data structures, Timeliner then recursively adds fields of the root objects that link non-root residual data structures from each Activity launch. This is represented in Algorithm 1 as the "RECURSE" function. When applied to the case presented in Figure 2, this would add any additional data structures reachable from the Intent, ActivityRecord, and ResolveInfo objects, leading to a full set of residual data structures (which represent an individual Activity launch).

After this step, Timeliner has obtained a list of Activities (whose launches create distinct residual data structures) and now needs to establish their temporal ordering. In the next section, Timeliner shall build a transition graph for these Activities.

B. Building the Transition Graph

As discussed in Section II-C, two Activities are said to have a transition if they have a corresponding temporal ordering. Simply put, an Activity e has a transition to an Activity f if e is launched before f.

As noted in Section II-A, the residual data structures are assigned slots spread across several runs. Further, recall from Section II-C, that a single run can give misleading information due to fragmentation, and thus Timeliner utilizes multiple runs to infer transitions between two Activities, say Activity e and Activity f.

We define an Activity e as a set of pairs, where each pair consists of a run and a list of corresponding slots occupied by residual data structures of the Activity e. This can be represented as:

$$e = \{(r, s) \mid r \in Runs \wedge s = \{i \mid r[i] \in Residue(e)\}\} \quad (1)$$

where r is a Run and s is the list of indices of occupied slots for Activity e. The "Residue" function represents the residual data structures allocated during the launch of Activity e, which were identified in the previous section.

Identifying Transitions. An Activity e will have a transition to an Activity f if (1) they share runs where all allocations of e precede all allocations of f and (2) they do not share a run where the opposite ordering occurs, that is, any allocation of e succeeds any allocation of f. With these properties in mind, we define the following two functions, allPrecede and anySucceed, counting the common runs that have all allocations of e preceding allocations of f and those with any allocation of e succeeding allocations of f, respectively.

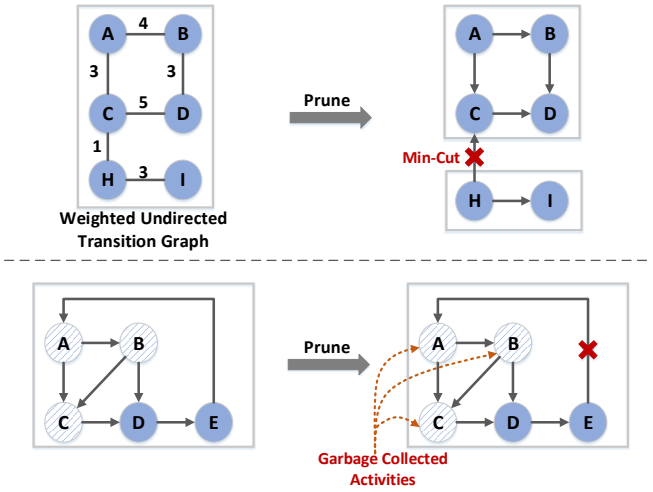

Fig. 4. Prune Erroneous Edges from Transition Graph.

$$\begin{aligned} allPrecede(e, f) &= |\{r \mid (r, m) \in e \ \wedge \ (r, n) \in f \ \wedge \\ &\qquad \max(m) < \min(n)\}| \\ anySucceed(e, f) &= |\{r \mid (r, m) \in e \ \wedge \ (r, n) \in f \ \wedge \\ &\qquad \max(m) > \min(n)\}| \end{aligned} \quad (2)$$

where e and f are Activities, r is a common run, and m and n are lists of slots in the common run r. There exists a transition between e and f if allPrecede(e,f) is positive and anySucceed(e,f) is equal to zero. Timeliner organizes these Activities as nodes and transitions as edges in a graph, called the *transition graph*.

Assigning Weights to Transitions. The higher the number of shared runs between two Activities (while maintaining the correct ordering), the higher the confidence in the transition, as the odds of coincidentally sharing runs decreases exponentially with the number of shared runs. Hence, if a transition exists, we assign the transition from Activity e to Activity f a weight equal to the number of shared runs between the two Activities. Note that the number of common runs is equal to allPrecede(e,f) + anySucceed(e,f), however as anySucceed(e,f) is equal to zero for a transition, the weight a transition is assigned is equal to allPrecede(e,f).

Pruning the Transition Graph. As noted in Section II-C, due to fragmentation, it is possible for a new allocation to fill up a slot before a pre-existing allocation. This means that an erroneous spatial ordering exists from the new allocation to the pre-existing one, and this implies a transition edge from a new Activity to a much older one. Clearly, this erroneous edge in our transition graph can lead to a wrong temporal ordering and must be pruned.

To handle such erroneous transitions, we use the observation that while it is possible for a new Activity to reuse a run and share it with an older Activity, it is improbable for them to share two runs, and unlikely to share three. Hence, we assume

a maximum weight of two for such edges. Further, such an erroneous edge can be classified as one of the following two cases:

1) **Transition between two connected components:** The edge connects two connected components, which contain legitimate transitions and are highly interconnected. In other words, such an edge serves as the *minimum cut* in the graph, and as discussed above, is limited to a maximum weight of two. An example can be seen in the top half of Figure 4, where there is an erroneous edge pointing from node H to node C, serving as the minimum cut between the two subgraphs. Pruning the graph in the example leads to two connected components, one with nodes $\{A, B, C, D\}$ and the other with nodes $\{H, I\}$.
   To implement this, Timeliner utilizes a min-cut algorithm (Stoer-Wagner [57]) on the corresponding undirected version of the transition graph, running the algorithm for each connected component, as explained in Algorithm 2. Assuming that there are n Activities and hence $O(n^2)$ edges, the time complexity of the algorithm is $O(n^3)$.
2) **Transition in the same connected component:** The edge connects two nodes in the same connected component, creating a cycle. Note that this can only happen if a garbage collection event happens between the Activities covered in the transition subgraph, allowing a new Activity to re-use a run used by an older Activity.
   Timeliner utilizes this property to remove the erroneous edge, pointing from an Activity launched after the garbage collection event, to an Activity launched before. As noted in Section II-B, the Activities that occurred before the last garbage collection event, called garbage collected Activities, can be identified by their diminished residual data structures. An example can be seen in Figure 4, with the erroneous edge pointing from node E to node A removed from the transition graph. The complexity of this algorithm is linear with respect to the number of edges.

In the next section, Timeliner utilizes this pruned transition graph and reconstructs the device-wide sequence of Activities, which we call the *timeline*.

C. Reconstructing the Global Ordering for Activities

Before we describe the procedure of finding the temporal ordering, it is important to prove the existence of one. As discussed in Section II-C, Timeliner solves for the topological ordering for a given transition graph. It is a known property of directed graphs that a topological ordering exists if and only if it is acyclic. While it is obvious that a graph with a cycle cannot be topologically ordered, it can also be proven that a depth first search in an acyclic graph (where a node is processed after its children are processed) leads to a reverse topological ordering. Hence, proving the existence of a topological ordering is equivalent to proving that the directed graph is acyclic. In Timeliner's transition graph, because of the "first-available" memory allocator algorithm, cycles can only be erroneous. Therefore, the acyclic property of the transition graph is guaranteed by the pruning described in Section III-B.

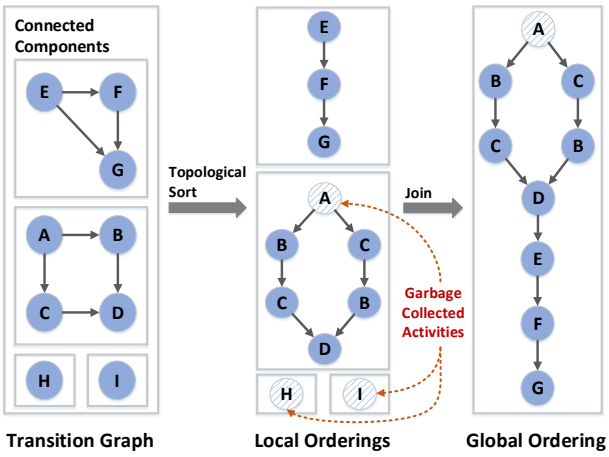

Fig. 5. Deriving the Global Ordering from the Transition Graph.

Algorithm 2 Reconstruct the Global Ordering.

Input: Graph $Transitions$
Output: Graph $Timeline$

▷ Remove Erroneous Edges - O(n^3)
GraphList $Components \leftarrow Transitions_{components}$
for Graph $g \in Components$ **do**
  $(cutSize, cutEdges) \leftarrow$min-cut($undirected(g)$)
  if $cutSize < 3$ **then**
    $Transitions \leftarrow Transitions - cutEdges$
  if $HasCycle(g)$ **then**
    for Transition $(e \rightarrow f) \in g$ **do**
      if *not* $IsGarbageCollected(Residue(e))$ **then**
        if $IsGarbageCollected(Residue(f))$ **then**
          $g \leftarrow g - (e \rightarrow f)$
▷ Topologically Sort into Local Orderings - O(numTimelines*n^2)
GraphList $LocalOrderings \leftarrow \varnothing$
GraphList $Components \leftarrow Transitions_{components}$
for Graph $g \in Components$ **do**
  $LocalOrderings \leftarrow LocalOrderings \cup$ topological-sort(g)
▷ Identify Joinable Local Orderings - O(n)
GraphList $JoinableOrderings \leftarrow \varnothing$
for Graph $g \in LocalOrderings$ **do**
  Activity $a \leftarrow g.lastActivity()$
  if *not* $IsGarbageCollected(Residue(a))$ **then**
    $JoinableOrderings \leftarrow JoinableOrderings \cup g$
▷ Join Local Orderings into Global Ordering - O(n)
Graph $Timeline \leftarrow \varnothing$
$JoinableOrderings.lastActivity().sort()$
for Graph $g \in JoinableOrderings$ **do**
  $Timeline.append(g)$

Local Orderings. Given a list of Activities recovered in Section III-A, Timeliner topologically orders the transition graph from Section III-B to establish various local orderings of Activity launches. However, as discussed in Section II-C, there are possible ambiguities in the temporal orderings of Activities, causing multiple possible timelines. An example can be seen in Figure 5, with the local ordering of nodes $\{A, B, C, D\}$ having two possible solutions. To effectively compute all the solutions, Timeliner performs a depth-first search with backtracking. The complexity of this algorithm is O(numTimelines*n^2), assuming n Activities and numTimelines solutions of temporal orderings.

Joinable Local Orderings. As described in Section II-A, Timeliner uses the property that runs are allocated via a "first-available" algorithm. This implies that, just like allocation

TABLE I. LIST OF SOME APPLICATIONS AND A FEW EXAMPLE ACTIVITIES.

Application	Activities			
WhatsApp	HomeActivity	Conversation	VoipActivity	RecordAudio
	CameraActivity	MediaGallery	ProfileActivity	VoiceMessaging
WeChat	LauncherUI	ChattingUI	AlbumPreviewUI	VideoActivity
	SelectContactUI	ContactInfoUI	GroupCardSelectUI	NearbyFriendsIntroUI
Signal	ConversationListActivity	ConversationActivity	RedPhone	NewConversationActivity
	GroupCreateActivity	ContactSelectionActivity	ShareActivity	SmsSendToActivity
Skype	HubActivity	PreCallActivity	ContactDirectorySearch	ContactProfileActivity
	ContactEditActivity	ContactDetail	ContactAddNumber	AddParticipantsActivity
Messaging	ConversationListActivity	ConversationActivity	WidgetReplyActivity	PeopleAndOptionsActivity
	ApplicationSettingsActivity	ShareIntentActivity	WidgetPickConversation	VideoShareActivity
Dialer	InCallActivity	CallLogActivity	CallDetailActivity	PeopleActivity
	QuickContactActivity	BlockedNumbersActivity	ImportVCard	CallSubjectDialog
Chase	AccountsActivity	BillPayAddStartActivity	BillPayAddVerifyActivity	BillPayHistoryActivity
	QuickPayChooseRecipient	TransferActivity	QuickDepositStartActivity	FindBranchActivity
Gmail	ConversationListActivity	ComposeActivityGmail	AccountSetupFinalActivity	GmailPreferenceActivity
Facebook	SplashScreenActivity	PickerLauncherActivity	ComposerActivity	FbMainTabActivity
File Browser	FileBrowserActivity	TaskProgressActivity	FileConverterActivity	HttpServerActivity
Netflix	HomeActivity	SearchActivity	ShowDetailsActivity	PlayerActivity

slots, the runs for different local orderings are spatially ordered. However, this spatially-increasing ordering does not always hold true, because of garbage collection events. A garbage collection event frees up low memory runs that are used by future Activities, causing a backward jump in the spatial ordering. Hence, Timeliner only joins those local orderings whose last Activities occur after the last garbage collection event. Such *joinable* local orderings can be distinguished by identifying garbage collected Activities as discussed in Section II-B. An example can be seen in Figure 5, where the two local orderings with nodes $\{A, B, C, D\}$ and $\{E, F, G\}$ are joinable, even though node A is a garbage collected Activity. Note that while garbage collection makes it difficult to order garbage collected Activities, it does not entirely prohibit it. For example, a local ordering that includes Activities that span a garbage collection event, from both before and after the event, is a joinable local ordering.

Global Ordering. To join multiple joinable local orderings into a single global ordering, Timeliner first identifies the Activities that were launched after the last garbage collection, as explained in Section II-B. Then Timeliner joins the local orderings which end with these Activities, following the spatial ordering of the runs that hold their allocations, yielding the global ordering. For example, in Figure 5, we see that the local orderings with nodes $\{A, B, C, D\}$ and $\{E, F, G\}$ are joined into a global ordering. The joining of local orderings into a global ordering has a complexity linear with respect to the number of Activities.

The resultant global ordering is returned by Timeliner to the investigators as the device-wide sequence of user actions.

IV. TIMELINER EVALUATION

Timeliner is implemented as a plugin for the AOSP (Android Open Source Project) and executes within an Android emulator, utilizing ART's runtime environment to identify crucial data structures for the memory allocator. Timeliner also reuses ART's various libraries to automatically parse and process the definitions of the residual data structures stored in the input memory image.

Setup. Timeliner is evaluated across 3 commercially available smartphones (Samsung Galaxy S4, LG G3 and Motorola Moto G3) using a variety of different applications. These include messaging apps such as WhatsApp, WeChat, Signal (widely renowned for security), each vendor's Messaging app, voice and video telephony apps like the vendors' Dialer apps and Skype. We also include email applications such as Gmail, the Chase Banking personal banking app, a video streaming app (Netflix), a social network app (Facebook), and various utility apps such as File Browser, Downloads, PDFReader, Camera, and Google Maps.

Table I lists a small subset of the Activities that are present in some different apps that we used in our evaluation. As Table I shows, the names of these Activities are very descriptive of the user actions they represent. For example, even within sophisticated apps like Signal, we can see Activities such as *ConversationListActivity* and *ConversationActivity* which describe viewing a list of past conversations versus clicking into a single conversation. Representing the action of making a voice/video call, we see the *VoipActivity* in WhatsApp, *VideoActivity* in WeChat, *RedPhone* (making a secure phone call) in Signal, *PreCallActivity* in Skype, and *InCallActivity* in Dialer. Even fine-grained app-specific actions can be captured, such as *ComposeActivityGmail* for composing an email, *BillPayAddStartActivity* for initiating a bill payment, and *QuickDepositStartActivity* for starting a check deposit. These vivid descriptions are due to the fact that Activities serve as intuitive abstractions for user actions.

However, the most important information for a criminal investigator is not just isolated user actions but the complex *sequencing of Activities*. For example, a *FileBrowserActivity* followed by a *TaskProgressActivity(delete)* showcases that the user deleted a file. This interplay of activities can be used to develop the timeline of a crime: For example, a user first takes a photograph with the *CameraLauncher* Activity in the camera app, followed by sharing the photo via WhatsApp's *ChooserActivity(share)* and *ImagePreview* activities. Finally, the user opens the photo via the *FileBrowserActivity* and deletes it with the *TaskProgressActivity(delete)* Activity. This semantically-meaningful series of user actions can be essential for quickly focusing a developing criminal investigation.

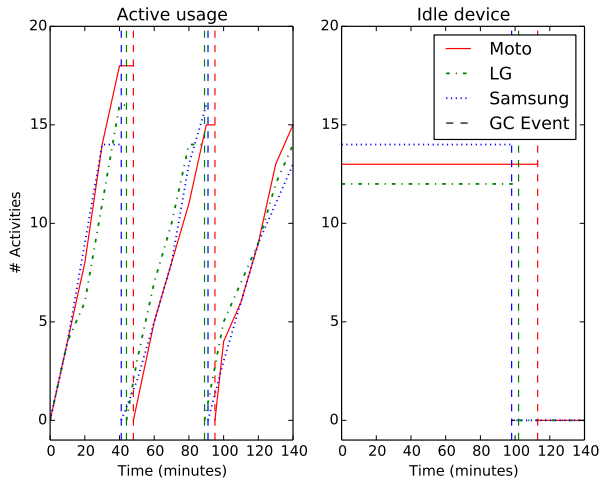

Fig. 6. Duration Between Garbage Collection Events Across Three Devices During Active and Idle Usage.

A. Garbage Collection

Garbage collection events can clear the allocations of Activities that are not alive (on the Activity stack) and hence do not have a reference towards them, causing limited evidence recovery. Further, garbage collection can also break the spatially-increasing ordering (due to the "first-available" algorithms) by causing a jump from high to low memory addresses as those runs become available. Hence, garbage collection events lead to (1) loss of evidence and (2) partial loss of spatial ordering for the remaining evidence in memory. However, note that while garbage collection makes it difficult to order garbage collected Activities, they can still be ordered if they are a part of some joinable local ordering.

To understand the limitations on Timeliner due to garbage collection, we begin the evaluation of Timeliner by evaluating (1) the frequency of garbage collection and (2) Timeliner's recovery after garbage collection events.

We first evaluate garbage collection frequency across different devices and under different usage conditions, while aiming to measure the time duration between different garbage collection events. To do so, we instrumented and measured the frequency of garbage collection on the *ActivityManagerService* process (the subject of Timeliner's reconstruction). Note that this is a service provided by the Android framework, which is largely unaffected by the processing done in any application. Hence, while certain applications are quite memory intensive (causing heavy workload of allocations), the frequent garbage collections are limited to their own processes.

This garbage collection profiling was carried out under two different conditions: (1) the phone was left idle and (2) the phone underwent constant user activity. For this purpose, we first installed all the applications listed in Table I on the three devices. For the idle case, we turned the device's screen off and left the phone idle. However, due to the presence of events raised by various background services (e.g., the AlarmManager service) the memory usage of the ActivityManager process slowly increases with time. For the active usage case, we repeatedly followed the sequence of Activities shown in Set A listed in Table III, raising a new Activity every two to three minutes.

TABLE II. RECOVERY BY TIMELINER UNDER GARBAGE COLLECTION.

Time (minutes)	Total Activities	Activities Since GC	Activities Recovered	Activities Ordered
0	1	0	1	0
10	8	7	8	7
20	11	10	11	10
30	15	14	15	14
40	17	16	17	16
GC_FOR_ALLOC at t = 44 minutes				
50	23	6	8	7
60	28	11	13	12
70	31	14	16	15
80	34	17	19	18
GC_FOR_ALLOC at t = 82 minutes				
90	41	7	10	8
100	46	12	15	13
110	50	16	19	17

Figure 6 presents our profiling results across the three devices under each usage condition, with Activities raised since last garbage collection event shown for each device. In the active usage case, garbage collection events were triggered periodically after 41-50 minutes (that followed raising 14 to 18 Activities). Interestingly, even after repeated triggering of the garbage collection events, this period remained roughly the same. This suggests a stable heap size and memory usage pattern. In the idle case, garbage collection events were triggered after 98 to 113 minutes, again due to the slower increase in memory usage within the *ActivityManagerService* process when the device is idle.

To further confirm Timeliner's recovery under garbage collection, we profiled garbage collection events and Activity launches on the Motorola device. During this time, we captured memory snapshots every 10 minutes (as any one of those could be the one taken when investigators confiscate the device) and use Timeliner to recover the Activities in the memory image. The results are detailed in Table II.

As expected, we can see that the Activities recovered by Timeliner include the set of Activities launched since the last garbage collection event. Timeliner also recovered Activities that survived garbage collection. Out of these Activities that survived the garbage collection event, a few of them also get ordered as they were part of a joinable local ordering. Note that there are two garbage collection events because of memory allocation requests (GC_FOR_ALLOC) at 44 and 82 minutes, respectively.

In a criminal investigation, a device's past usage will blend both idle and active periods, but in either case this is an ample time window to capture the details of a crime carried out on a smartphone.

B. Micro-Benchmarks

To evaluate Timeliner's reconstruction capability, this section presents micro-benchmark results measured during Timeliner's recovery across a variety of memory images. For this recovery, the authors interacted with the sets of Activities described in Table I. The activities in the applications were launched following one of ten random sequences, the exact sequences of launches are detailed in Sets A through J in Table III. We performed six experiments on each of the three devices, each experiment using a different activity sequence taken from the defined ten random sequences. Each Activity

TABLE III. TIMELINE RECONSTRUCTION FOR MICRO-BENCHMARK EXPERIMENTS ACROSS DIFFERENT DEVICES.

Device	Experiment Set	Activity Count	Duration (Minutes)	Activities Recovered	Activities Ordered	Root Structures	Residual Structures	Local Orderings	Number of Timelines	Kendall-Tau Distance
Samsung S4	Set A	15	39	17	16	91	6526	4	1	0
	Set B	13	37	14	14	77	5584	2	1	0
	Set C	16	51	18	16	95	6218	5	1	0
	Set D	12	22	13	12	65	4881	4	1	0
	Set E	13	34	15	14	81	5079	3	1	0
	Set F	15	45	16	15	86	6049	5	1	0
LG G3	Set G	14	38	16	15	85	5427	5	1	0
	Set H	16	42	18	16	94	6395	4	1	0
	Set A	15	33	17	16	92	6241	4	1	0
	Set I	14	28	16	14	83	5280	5	1	0
	Set J	15	37	16	16	97	6429	3	1	0
	Set B	13	28	15	14	78	5227	3	1	0
Moto G3	Set G	14	35	15	14	83	5016	4	1	0
	Set D	12	28	15	13	70	4589	3	1	0
	Set C	15	57	16	15	88	5782	5	1	0
	Set I	14	39	16	14	87	5296	4	1	0
	Set A	15	41	17	16	85	5721	5	1	0
	Set H	14	48	15	14	83	5192	3	1	0

Experiment Set	Sequence of Activities (Randomly Chosen for Micro-Benchmarks)
Set A	Launcher → WhatsApp{ Conversation → HomeActivity → Conversation → VoipActivity → Conversation → CameraActivity → Conversation } → Launcher → Gmail{ ConverastionListActivity → ComposeActivityGmail } → Launcher → Downloads{ FilesActivity → ShareActivity } → Launcher
Set B	Launcher → Skype{ HubActivity → ContactDirectorySearch → ContactProfileActivity → ContactEditActivity } → Launcher → Dialer{ CallLogActivity → CallDetailActivity → PeopleActivity → InCallActivity } → Launcher → Netflix{ HomeActivity → PlayerActivity }
Set C	Launcher → WeChat{ LauncherUI → ChattingUI → VideoActivity → SelectContactUI → SingleChatInfoUI → ContactInfoUI } → Launcher → Signal{ ConversationList → Conversation → GroupCreate } → Launcher → Signal{ GroupCreate → ContactSelection } → Launcher → Gmail{ ConversationList → GmailPreference } → Launcher
Set D	Launcher → Messaging{ ConversationList → Conversation } → Launcher → Messaging{ Conversation → PeopleAndOptions } → Dialer{ InCallActivity } → Launcher → Chase{ AccountsActivity → BillPayAddStartActivity → BillPayAddVerifyActivity } → Launcher
Set E	Launcher → Netflix{ HomeActivity → SearchActivity → ShowDetailsActivity → PlayerActivity } → Launcher → WhatsApp{ HomeActivity → GroupMemberSelector → NewGroup → Conversation } → Launcher → Downloads{ FilesActivity } → Launcher
Set F	Launcher → Signal{ ConversationListActivity → ConversationActivity → DocumentsActivity → RedPhone } → Launcher → Skype{ HubActivity → PreCallActivity → HubActivity → ContactProfileActivity } → Launcher → Chase{ AccountsActivity → BillPayHistoryActivity → QuickPayTodoListActivity } → Launcher
Set G	Launcher → Gmail{ ConverastionListActivity → ComposeActivityGmail } → Launcher → Dialer{ CallLogActivity → PeopleActivity } → Launcher → Dialer{ PeopleActivity → BlockedNumbersActivity → InCallActivity } → Launcher → Downloads{ FilesActivity → UploadActivity } → Launcher
Set H	Launcher → Chase{ HomeActivity → AccountsActivity → AlertsHistoryActivity → FindBranchActivity → LocationInfoActivity → AccountsActivity } → Launcher → Dialer{ CallLogActivity → CallDetailActivity → PeopleActivity } → Launcher → WhatsApp{ HomeActivity → Conversation → VoipActivity → CameraActivity }
Set I	Launcher → Skype{ HubActivity → PreCallActivity → ContactProfileActivity } → Launcher → Signal{ ConversationListActivity → ConversationActivity → RedPhone → NewConversationActivity → ConversationActivity → DocumentsActivity } → Launcher → Gmail{ ConversationListActivity → GmailPreferenceActivity }
Set J	Launcher → WhatsApp { HomeActivity → GroupMemberSelector → NewGroup → Conversation } → Launcher → Skype{ HubActivity → ContactDirectorySearch → ContactProfile → ContactEdit } → Launcher → Netflix{ HomeActivity → SearchActivity } → Launcher → Netflix{ SearchActivity → ShowDetailsActivity → PlayerActivity }

in a sequence is started and left on the screen for a varying amount of time, around two to three minutes. To mitigate the effect of garbage collection on these micro-benchmarks, we initiated a garbage collection before starting each experiment. Memory images were captured by a custom handler invoked at the next garbage collection event, implemented by instrumenting the internal garbage collection event handler.

To verify the accuracy of Timeliner's recovery, we compared the reconstructed timeline with the original list of Activities. The ground truth about Activity launches was captured by profiling the *ActivityManagerService* process. We stored the addresses of the allocated Activity-launch related data structures along with the original timeline of the activities. The allocated data structures were stored to correctly identify recovered activities from the original sequence and this timeline was used to verify Timeliner. Note that Timeliner did not need nor have access to this ground truth information and reconstructed Activity timelines completely oblivious to our external measurement.

Table III provides a summary of the micro-benchmark results from these experiments. The first column shows the device the experiment was run upon and the experiment set used, followed by *Activity Count* and *Duration* of the ground truth timeline. The fourth and fifth columns list the number of Activities recovered and the number of Activities ordered by Timeliner. Next two columns present recovery metrics: the total number of roots and residual data structures recovered. The eighth and the ninth columns show the number of local orderings recovered and the number of possible timelines in the global ordering and the last column compares the original ground truth to the recovered timeline (minus Activities not in the ground truth) via *Kendall-Tau* distance [10]. Kendall-Tau distance compares two ordered lists and calculates the number of pairwise disagreements between them. This is a good measure for a timeline as the more displaced an activity is from its correct position, the higher the Kendall-Tau distance (therefore a minimal distance value is best). Finally, the exact sequence of Activities in the 10 experiment sets is presented at the bottom of Table III.

First, from Table III, observe that for some cases like Set C in Samsung, Set J in LG, and Set C and Set H in Motorola, the Activity count in the experiment is less than the number of Activities in the experiment set. This is because in these cases, a garbage collection event was triggered before the sequence of Activities was finished. Hence, a smaller set of Activities is taken from the experiment set. For other cases, if the sequence was finished without a garbage collection event, one was triggered manually.

Even though garbage collection is triggered manually in some cases, the workload in Table III is quite similar to the one in Section IV-A. For example, the number of Activities in the original timeline varies from 12 to 16 with an average of 14.16 Activities per experiment, similar to what was observed in Section IV-A. Similarly, the time duration varies from 22

to 57 minutes with an average of 37.88 minutes, again similar to the observed results in Section IV-A.

The results of the micro-benchmarks are also quite similar to the results from Section IV-A. For example, just like in Table II, Timeliner recovers more Activities than the ones that were raised after the last garbage collection event because some Activities will survive garbage collection. While not all Activities can be ordered because of loss of spatial and temporal information, some local orderings are joinable. This allows Timeliner to order more Activities than were in the experiment set. For the micro-benchmarks, Timeliner recovers 13 to 18 Activities with an average of 15.83 Activities per experiment. The global orderings generated by Timeliner contain 12 to 16 Activities with an average of 14.67 Activities per experiment.

Looking at the data structure metrics, we see that Timeliner recovers an average of 84.44 root data structures per experiment, which equals 5.33 roots per recovered Activity. Similarly, Timeliner recovers an average of 5607.33 residual data structures per experiment, which leads to 354.15 residual data structures per recovered Activity. These averages are roughly constant across the various experiments, implying that the residual data structures are (roughly) application-generic.

We also compare the metrics across the three devices. On average, the Samsung device yields 14.33 Activities over 38 minutes, while LG has 14.83 Activities over 34.33 minutes, and Motorola has 14.83 Activities over 41.33 minutes. We also observe that (per-Activity) the Samsung device yields an average of 369.21 residual data structures, LG has an average of 357.13 residual data structures, and Motorola an average of 336.12 residual data structures, which follow the earlier observations that there are roughly similar number of residual data structures per activity even across devices. The similarity of these results gives us confidence that vendor-customizations rarely affect both low-level primitives of memory allocation and application-generic residual data structures.

Finally, we compare the metrics that pertain to ordering, namely local orderings, number of possible timelines in the global ordering and the Kendall-Tau distance. As we can see, we get a few local orderings, varying from 2 to 5 for different experiments. From the other two metrics, it is visible that Timeliner is highly successful in ordering the Activities. Not only is Timeliner highly precise, with 1 unique timeline in the global ordering of every experiment, it is also highly accurate with the Kendall-Tau distance for all the experiments being equal to zero. In other words, Timeliner is able to perform perfect recovery of the Activity timeline.

The accuracy of Timeliner, while surprising, is intuitive as there are no spatial (and hence temporal) ambiguities because of the following two properties: (1) application-generic residual data structures contain a large number of objects spread across shared and thread-local runs, ensuring unambiguous spatial ordering, and (2) a complete global ordering of the Activities after the last garbage collection event is ensured by their local orderings being joinable (because of the "first-available" algorithms).

Next, we show that Timeliner's design is generic and applicable across various Android versions and even other memory allocators.

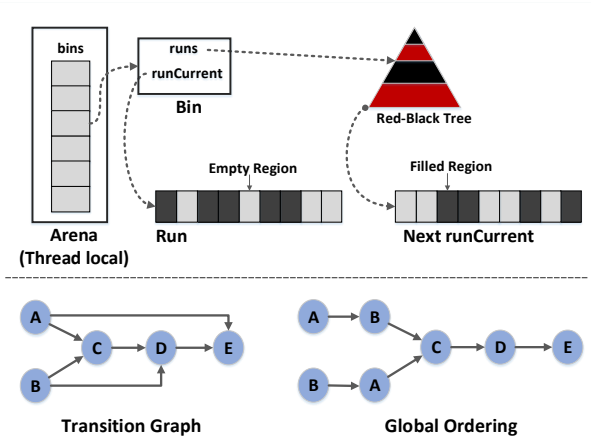

Fig. 7. mozjemalloc Design and Simulated Results.

C. Design Generality

While Timeliner is implemented within a specific Android platform, Timeliner's *design and operation* is generic, and Timeliner is immediately applicable across the newest and most widely used Android versions. The devices in this evaluation all use different Android platforms: the Samsung running Android 5.0, LG running Android 5.1, and Android 6.0 running on the Motorola. These versions comprise 61.5% of the current Android devices and represent a wide variety of available Android smartphones [2].

Timeliner's generic design is due to a robust set of root data structures used to identify the residual data structures. The same set of root data structures is highly efficient because the Activity launch logic is similar across various Android versions. Further, Timeliner can also be applied to memory allocators other than RosAlloc, as many other memory allocators also perform "first-available" allocations.

Extension to jemalloc. jemalloc is a memory allocator widely used across products such as Firefox, Cassandra, Redis, among many others [9]. Our investigation reveals that jemalloc (without thread caching) also utilizes a similar design to RosAlloc, with a "first-available" algorithm for memory allocation. In particular, we discuss the design and extension of Timeliner to mozjemalloc [14], a modified implementation of jemalloc used across various Mozilla products such as Firefox and Thunderbird. The following discussion is based on the mozjemalloc version bundled as the default memory allocator in Firefox 55.0.

The design of mozjemalloc is shown in Figure 7, with allocations stored in *Regions* (slots in RosAlloc) and regions of same size organized in *Runs*. Runs with allocations of the same size are placed in bins, and bins are placed in a (thread-local) *Arena*. Each bin has a current run to allocate from, with the rest of the runs (that have free Regions) organized in a red-black tree. The algorithms utilize a bitmap for "first-available" Region allocation. New runs are also allocated in accordance to a "first-available" algorithm, utilizing a red-black tree.

To evaluate Timeliner's recovery on mozjemalloc, we simulate Activity launches by following the allocation size distribution from Figure 3, spread out evenly across two threads in mozjemalloc. We simulated five Activity launches

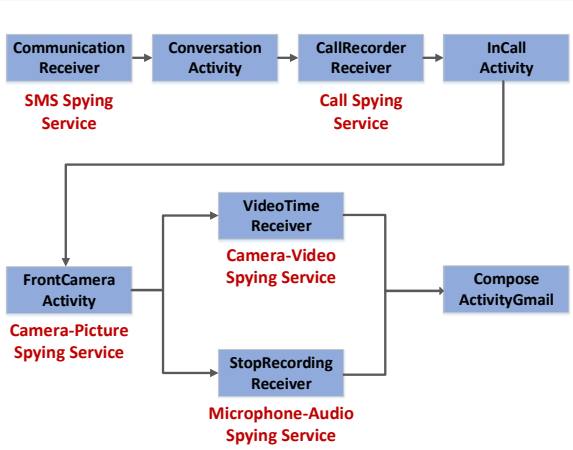

Fig. 8. Recovered Transition Graph for Spyware Attack.

on mozjemalloc, which was initialized with five threads, and inferred their transitions with Timeliner— applying the same spatial-temporal principle from before. Timeliner constructed and topologically sorted a transition graph for the Activities in order to reconstruct the global ordering shown in Figure 7. As the figure illustrates, there was an ambiguity in the order of Activities, with the two possible timelines shown.

While Timeliner is designed to recover user actions (Activities), it is not limited to them. We extended Timeliner to recover and order app actions (*BroadcastReceiver* callbacks on system events) carried out through Intents and BroadcastRecord objects. These interactions are lightweight and do not produce the plethora of data structures created by Activities. As such, we end up with a small number of residual data structures which are also limited to allocations in the smaller-sized thread-local runs. This implies that for two Broadcasts, there can be a spatial ambiguity similar to what was observed in mozjemalloc. Note that as an Activity generates residual data structures spread across various threads, making it very likely that an Activity and a Broadcast share several thread-local runs, it enables Timeliner to infer spatial ordering between an Activity and a Broadcast. The next section demonstrates this in our case study of a spyware attack investigation.

D. Case Study: Spyware Attack Investigation

Being the most widely used smartphone platform, Android has increasingly been targeted by various sophisticated spyware attacks [4], [11]. Spyware has recently been employed by nation states targeting journalists and activists [13] and even by abusive spouses to monitor their families [7]. Modern spyware are extremely stealthy and sometimes do not even require physical access for installation, relying on drive-by downloads and vulnerabilities. These spyware track the victim's calls, texts, app usage, and smartphone features such as keyboard inputs, location, microphone, and camera. In this case study, we examine the capability of Timeliner to recover the actions of TheOneSpy [18], a commercially available spyware application.

Unknown to the victim ("John"), his smartphone has been infected with TheOneSpy. While on his way to a confidential meeting, John receives a text reminder for the meeting and an incoming call for the meeting location. During the meeting, John receives an email for which he initiates a response. However, he notices that the keyboard has been changed from the default (Android Keyboard) to a custom one, which is visually differentiable. A quick investigation in the smartphone's settings reveals that the custom keyboard is the spyware's keylogger, and the spyware has access permissions for the microphone and camera. To confirm the spyware's activity, a memory image is taken and analyzed with Timeliner.

For this case study, we consider only those Activities and Broadcasts that are relevant to the spyware application. Timeliner recovers 9 Activities/Broadcasts with 26 roots and 1638 residual data structures, with one occurring before the last garbage collection. Two local orderings are combined to form a single global ordering, with two possible timelines. Note that the Kendall-Tau distance of the two timelines are 0 and 1 — as the ground truth is one of the possible timelines.

The deduced transition graph is shown in Figure 8. The two possible timelines are shown in Table IV starting with a Broadcast receiver *CommunicationReceiver* on the spyware and the user opening a text with *ConversationActivity*. Following these is another Broadcast receiver *CallRecorderReceiver* and the user answering the call with *InCallActivity*. Next, there are a few spyware Activities/Broadcasts: *FrontCameraActivity*, *VideoTimeReceiver*, and *StopRecordingReceiver*. Finally, there is a *ComposeActivityGmail* Activity as the user replies to the email. While the names are quite verbose, a quick look at the spyware bytecode confirms that *CommunicationReceiver* and *CallRecorderReceiver* are used for spying on incoming texts and calls, respectively. *FrontCameraActivity*, *VideoTimeReceiver*, and *StopRecordingReceiver* are used for remotely recording pictures, videos, and audio, respectively. The confirmation of the spying activity makes it highly likely that the secrecy of the meeting had been compromised.

Finally, note that while there is a spatial and hence a temporal ambiguity between the two Broadcasts, Timeliner still establishes a sufficient evidence of the meeting being compromised — as the remote video recording and audio recording is contained between the phone call and the received email. Both timelines are shown in Table IV.

Next, we show the application of Timeliner in a few crime scenarios where the culprit is a human (instead of spyware).

E. Case Study: Military Espionage

Timeliner is particularly useful in investigating misuse of mobile devices in secured environments such as a Sensitive Compartmented Information Facility (SCIF) where personal mobile devices are not allowed and commonly, a locker is provided outside the SCIF where mobile phones can be secured, or they are left in the employee's car. Our case study is motivated by real espionage cases, such as the prosecution of Air Force Intelligence Officer Brian Regan [32] or that of Gillette employee Steven Davis [8] who were prosecuted for stealing classified national documents and corporate secrets respectively.

Our perpetrator Skip was hired as a defense contractor, working on a classified project for a federal agency, with routine access to sensitive documents. One day, after attempting to access classified information unrelated to his job, he checked

TABLE IV. RESULTS FOR CASE STUDIES ON MOTOROLA G3.

Case Study	Activity Count	Duration (Minutes)	Activities Recovered	Activities Ordered	Root Structures	Residual Structures	Local Orderings	Number of Timelines	Kendall-Tau Distance
Spyware Attack	8	27	9	8	26	1638	3	2	{0,1}
Military Espionage	18	36	19	18	112	7151	5	1	0
Distracted Driving	17	16	18	17	92	6259	3	1	0
Kidnapping Investigation	18	41	19	18	101	6879	5	1	0

Case Study	Recovered Timelines
Spyware Attack	Spyware{ CommunicationReceiver } → Messaging{ Conversation } → Spyware{ CallRecorderReceiver } → Dialer{ InCallActivity } → Spyware{ FrontCameraActivity → VideoTimeReceiver → StopRecordingReceiver } → Gmail{ ComposeActivityGmail }
	Spyware{ CommunicationReceiver } → Messaging{ Conversation } → Spyware{ CallRecorderReceiver } → Dialer{ InCallActivity } → Spyware{ FrontCameraActivity → StopRecordingReceiver → VideoTimeReceiver } → Gmail{ ComposeActivityGmail }
Military Espionage	Signal{ RedPhone → ConversationActivity } → Launcher → CameraActivity → ChooserActivity(share) → WhatsApp{ ContactPicker → Conversation → ImagePreview → Conversation } → Launcher → FileBrowser{ FileBrowserActivity → TaskProgressActivity(delete) → FileBrowserActivity } → Launcher → Messaging{ Conversation } → Launcher → Chase{ HomeActivity → AccountsActivity }
Distracted Driving	Maps{ MapsActivity } → RecentsActivity → Netflix{ HomeActivity → SearchActivity → MovieDetailsActivity → PlayerActivity } → Launcher → Dialer{ DialContactsActivity → InCallActivity → DialContactsActivity } → Launcher → Netflix{ PlayerActivity → MovieDetailsActivity → SearchActivity → HomeActivity } → Launcher → RecentsActivity
Kidnapping Investigation	CameraActivity → Launcher → Skype{ HubActivity → PreCallActivity → HubActivity } → Launcher → Messaging{ ConversationListActivity → ConversationActivity } → Launcher → Skype{ HubActivity → PreCallActivity → HubActivity } → Launcher → Maps{ MapsActivity } → Launcher → Facebook{ PickerLauncherActivity → ComposerActivity → FbMainTabActivity }

out of the SCIF and walked around the parking lot of the facility. Alerted to the recent unauthorized attempts to access classified information, security personnel followed Skip into the parking lot, where they determined that he was carrying a mobile phone.

The security personnel use Timeliner to determine the timeline of Skip's recent actions. As it turns out, Skip received a secure call (*RedPhone*) just before he checked into the SCIF. After entering the SCIF, he used the Camera app (*CameraLauncher*) in his phone to take some photographs. These photographs were then sent over WhatsApp (*Chooser-Activity(share)*) after he exited the SCIF and then summarily deleted (*TaskProgressActivity(delete)*). For selling the classified information, he received a deposit in his bank account, resulting in a text message (*ConversationActivity*), which he verified by opening the Chase Banking app (*AccountsActivity*). This timeline was deemed incriminating and Skip was then arrested and charged. As Table IV shows, Timeliner recovers 19 Activities, 112 roots, and 7151 residual data structures for this timeline.

This case study demonstrates how important a timeline of user-actions is to an investigation. A traditional content recovery alone would be extremely limited as the photographs of the classified documents were deleted by Skip, severely limiting an investigation relying on content. Timeliner, on the other hand, provides conclusive proof of photographs being taken, shared, and then deleted.

We acknowledge that federal authorities are currently more likely to have the expertise and resources to react quickly enough to use Timeliner to retrieve actionable evidence before the detrimental effects of garbage collection occur, as in Skip's case above. However, with proper resources and training, Timeliner is also usable in a variety of scenarios by local and state authorities. The next case studies explore two such very important potential uses.

F. Case Study: Distracted Driving

In this case study we consider the problem of distracted driving. Specifically, using a smartphone while driving, which accounts for roughly 18% [5] of all injury-inducing automotive crashes. This situation is becoming so severe, that akin to a breathalyzer test, the state of New York is considering a *Textalyzer* law [17]. This law shall allow a police officer to conduct on-the-spot forensic analysis of a smartphone to determine if a driver was distracted while driving. Traditional techniques focus only on app-specific events, limited mostly to text/call/email/browsing logs [12]. On the other hand, Timeliner's app-agnostic capabilities work *without* temporal logs and provide much stronger proof of a driver's suspected distraction while driving.

We base this case on an accident involving a Tesla vehicle [16] where the driver was determined to be watching a movie after putting the car into the *AutoPilot* mode, which would clearly be classified as distracted driving. In our case study, the driver called roadside assistance after an accident. The police arrive a few minutes later and notice the "recent apps" screen on the driver's smartphone. Suspecting termination of an application they image the smartphone memory.

Table IV shows that Timeliner recovered all 17 Activities that the driver used during the course of this case study via recovering three local orderings with 92 root and 6259 residual data structures. In the reconstructed timeline, investigators can see that the driver was first running the *MapsActivity* in Google Maps. At some point, the driver started Netflix with *Home-Activity*, followed by *SearchActivity* and *MovieDetailsActivity*, and finally playing a video with the *PlayerActivity*. Then the

driver goes to the Dialer app and places a phone call, which was identified as the call to roadside assistance using call logs. After the call, the driver restarts the Netflix app, but backs out of all activities and finally terminates the app from the *RecentsActivity*.

The timeline confirms that the Netflix video was playing before the call to roadside assistance (and hence before the accident) was placed, but was terminated afterwards to hide the incriminating actions. Timeliner's ability to generate a timeline to precisely capture user actions across several applications — even in the face of deliberate app termination to hide evidence — is essential in this case. Timeliner is able to reconstruct the evidence solely from the phone's memory and termination of applications by the user does not hinder the recovery of actionable evidence.

G. Case Study: Kidnapping Investigation

Mobile phone investigations have aided in the apprehension of numerous criminals, and being a memory forensics technique, Timeliner can be used to quickly focus an investigation. We base this case study loosely on the real kidnapping/murder investigation detailed in [37]. Described as a "kidnapping gone wrong", the victim was bound using duct tape, and unfortunately she died from asphyxiation before a fake "rescue" the kidnapper had planned could take place.

In our case study, the kidnapper ("Kyle") and an accomplice ("Fred") force the victim ("Sally") into a pickup truck. A passerby (who identified Sally) quickly informs the police and the police identify Kyle as matching the description of the kidnapper. Kyle, located by the police at his residence, claimed he did not leave his house and showed his recent social media uploads (a photo at home) as proof.

A field-investigation of his phone, with the aid of Timeliner, reveals his actions in the recent past (shown in Table IV). Timeliner recovers 19 Activities with 101 roots and 6879 residual data structures. Joining three of the local orderings, Timeliner is able to precisely and accurately recover the timeline. The timeline shows that Kyle took the "alibi" photo with *CameraLauncher*, but did not post it immediately. Instead, he used the Skype app to call a person (*PreCallActivity*), then message Sally over text (*ConversationListActivity* and *ConversationActivity*), followed by another call via Skype. Then he used Google Maps (*MapsActivity*) to navigate and then finally posts the "alibi" photo to Facebook (*PickerLauncherActivity* and *ComposerActivity*). The police identify the Skype call recipient as Fred via Skype logs and obtain clearance to deploy Stingrays ("IMSI catcher" devices) against Fred's number. This allows them to rapidly find both Sally and Fred in a nearby wooded area.

This case study demonstrates how Timeliner complements the traditional content recovery forensics. While the accomplice is identified by Skype logs on the smartphone, and the message recipient is identified as Sally with messaging logs, Timeliner provides the incriminating evidence of fake "alibi" photograph being taken before the Skype calls, which raises suspicion and provides enough proof to deploy Stingrays.

V. Discussion

As Timeliner relies on the observation that a temporal ordering in allocations produces a spatial ordering, any attempt by a device owner or an app to hide their actions must attack either the recovery of residual data structures or change the allocator's deterministic behavior.

However, as Timeliner focuses on only the ActivityManagerService process, separate from the apps, erasing evidence by modifying records or running garbage collection is not possible for even the most technically advanced criminals or privacy sensitive app developers. The only way a device owner may affect the ActivityManagerService is to flash a custom "Timeliner-aware" Android runtime system onto their device — which is both technically difficult (modifying Androids internals) and risky (may "brick" the device).

One way an app developer can avoid Timeliner's recovery is by not utilizing Activities, and building their own functionality to emulate Android's Activity stack (e.g., by intercepting back keypresses). However, this is a prohibitively cumbersome process for most app developers and has only been implemented in a few apps including certain web browsers and gaming apps due to strict performance requirements. Further, this also prevents an apps' interaction with other apps, for example – File Browser can directly share a file with the Gmail app (using an Intent for the ComposeActivity) but not with the gmail website opened on the Chrome browser app.

VI. Related Work

Timeline reconstruction is of interest to both cyber and traditional crime investigations. This interest is reflected in the wide variety of work done for creating timelines [28], [35], [36], [41], making better tools for editing and visualization [24], [47], and correlating sources together to infer semantics in a timeline [29], [39], [54]. However, all these methods are dependent on various logs and database files that are formatted independently by applications making their timeline recovery highly application-specific. Further, these logs and database files are limited to a small set of events that are logged. Even widely used commercial tools Oxygen [15] and Cellebrite [3] are application-specific and are limited to these small sets of events. Further, reliance on system level logging is untrustworthy as major phone manufacturers turn off Android features that reveal forensic information [6]. Timeliner, on the other hand, is application-generic and can reliably reconstruct a wider variety of actions into the timeline *from only a single image of volatile memory*.

Timeliner is more related to RetroScope [52], a memory forensics technique capable of reconstructing historical and temporally ordered GUI screens. However, Timeliner differs from RetroScope in two aspects. First, while RetroScope is limited to reconstruction for a single running (at the time of memory snapshot) app, Timeliner works *across* all apps and can construct a device-wide timeline of app activities (including terminated apps). Second, RetroScope reconstructs screens, which are renderings of GUI content, while Timeliner reconstructs *Activities*, which are abstractions of user actions/events. As such Timeliner and RetroScope perfectly complement each other, with Timeliner reconstructing the skeleton of a crime

story involving multiple apps and RetroScope re-rendering the activity details within each app.

Memory forensics has been applied extensively to the Android platform. Mostly these applications have focused on recovering raw data structures: app-specific login credentials, JVM control structures, raw Java objects, text messages, buffered media content, and a variety of application-specific data [22], [23], [38], [45], [51], [59]. Recovery of the raw data structures is performed via value-based [25], [31], [49], [55], [58] (relying on constants and expected values) or structure-based [26], [27], [46], [61] (relying on pointer constraints) scanning. In particular, SigGraph [43] recovers data structure instances using probabilistic analysis on the whole memory image. On the other hand, data structure recovery is only the first, preparatory step in Timeliner's timeline recovery.

Various memory introspection and memory analysis techniques have been used to determine malware and virus activity by observing kernel data structures [34], [48] or by identifying data structure signatures for polymorphic viruses [30]. However, while these techniques either rely on active introspection or recover only live kernel and virus data structures, Timeliner recovers and orders past app activities, including activities with no references from live data structures, using only a single memory image.

A number of recent works have gone beyond merely recovering raw data structures towards full-utilization of their content. DSCRETE [53] recovers a single data structure instance and utilizes binary analysis and code reuse to transform it into a human-understandable form. DEC0DE [60] also operates on a single data structure at a time, recovering call log entries using a finite state machine. Tools such as HOWARD [56], REWARDS [44], and TIE [40], infer data structure definitions in binary programs. DIMSUM [42] utilizes probabilistic inference to identify data structures without page mapping information. VCR [51] recovers media content using vendor generic signatures, and GUITAR [50] pieces back together various data structures to retrieve an application's GUI. As a new, complementary addition to the above tool set, Timeliner leverages spatial memory layout information to infer temporal ordering of user Activities.

VII. Conclusion

Targeting the problem of re-sequencing an Android device user's past actions, we present Timeliner, a memory forensics technique that reconstructs a timeline of *Activities* across all apps (including those which have terminated) that were performed on the device. Starting from the set of data structures left in a memory image by past Activity launches, Timeliner infers *Activity* transitions based on the relative memory layout of those data structures. Our results show that Timeliner is highly accurate in reconstructing past activities of a user. Moreover, we show through a suite of case studies that Timeliner is applicable to a variety of crime investigation scenarios.

Acknowledgment

We thank the anonymous reviewers for their insightful comments and suggestions. This work was supported, in part, by NSF under Awards 1409668 (including a supplement from NSA) and 1409534. Any opinions, findings, and conclusions in this paper are those of the authors and do not necessarily reflect the views of our sponsors.

References

[1] "Activity API Documentation," https://developer.android.com/reference/android/app/Activity.html.

[2] "Android version market shares," https://fossbytes.com/most-popular-android-versions-always-updated/.

[3] "Cellebrite UFED," https://www.cellebrite.com/en/solutions/pro-series/.

[4] "Chrysaor malware," https://android-developers.googleblog.com/2017/04/an-investigation-of-chrysaor-malware-on.html.

[5] "Dangers of distracted driving," https://www.fcc.gov/consumers/guides/dangers-texting-while-driving.

[6] "Devices without UsageStats API," http://stackoverflow.com/questions/32135903/devices-without-apps-using-usage-data-or-android-settings-usage-access-setting.

[7] "Domestic spying," http://www.independent.co.uk/news/uk/home-news/exclusive-abusers-using-spyware-apps-to-monitor-partners-reaches-epidemic-proportions-9945881.html.

[8] "Gillette corporate espionage," http://www.fraud-magazine.com/article.aspx?id=2147483718.

[9] "Jemalloc," https://github.com/jemalloc/jemalloc/wiki/Background.

[10] "Kendall-tau distance," https://en.wikipedia.org/wiki/Kendall_tau_distance.

[11] "Lipizzan malware," https://android-developers.googleblog.com/2017/07/from-chrysaor-to-lipizzan-blocking-new.html.

[12] "Mccann investigations," http://www.mccanninvestigations.com/media/6310/case_study_texting_and_driving.pdf.

[13] "Mexico spyware," https://www.theguardian.com/world/2017/jun/19/mexico-cellphone-software-spying-journalists-activists.

[14] "mozjemalloc," https://github.com/mozilla/gecko-dev/blob/master/memory/build/mozjemalloc.cpp.

[15] "Oxygen Forensic Analyst Timeline," http://www.oxygen-forensic.com/en/products/oxygen-forensic-detective/analyst/timeline.

[16] "Tesla autopilot accident," https://www.theguardian.com/technology/2016/jul/01/tesla-driver-killed-autopilot-self-driving-car-harry-potter.

[17] "Textalyzer," http://www.nytimes.com/2016/04/28/science/driving-texting-safety-textalyzer.html.

[18] "Theonespy," https://www.theonespy.com.

[19] "Commonwealth v. Phifer," SJC-11242, (2012).

[20] "Nissen v. Pierce County (Majority)," Washingotn S. Ct. 90875-3, (2015).

[21] "Hamilton v. State," Oklahoma S. Ct. F-2015-529, (2016).

[22] 504ENSICS Labs, "Dalvik Inspector," http://www.504ensics.com/automated-volatility-plugin-generation-with-dalvik-inspector/, 2013.

[23] D. Apostolopoulos, G. Marinakis, C. Ntantogian, and C. Xenakis, "Discovering authentication credentials in volatile memory of android mobile devices," in *Collaborative, Trusted and Privacy-Aware e/m-Services*, 2013.

[24] F. P. Buchholz and C. Falk, "Design and implementation of zeitline: a forensic timeline editor." in *DFRWS*, 2005.

[25] C. Bugcheck, "Grepexec: Grepping executive objects from pool memory," in *Proceedings of Digital Forensic Research Workshop*, 2006.

[26] M. Carbone, W. Cui, L. Lu, W. Lee, M. Peinado, and X. Jiang, "Mapping kernel objects to enable systematic integrity checking," in *Proceedings of ACM Conference on Computer and Communications Security*, 2009.

[27] A. Case, A. Cristina, L. Marziale, G. G. Richard, and V. Roussev, "FACE: Automated digital evidence discovery and correlation," *Digital Investigation*, vol. 5, 2008.

[28] Y. Chabot, A. Bertaux, C. Nicolle, and T. Kechadi, "Automatic timeline construction for computer forensics purposes," in *IEEE Joint Intelligence and Security Informatics Conference (ISI-EISIC 2014), 24-26 September, the Hague, Netherlands*. Institute of Electrical and Electronics Engineers, 2014.

[29] K. Chen, A. Clark, O. De Vel, and G. Mohay, "Ecf-event correlation for forensics," 2003.

[30] A. Cozzie, F. Stratton, H. Xue, and S. T. King, "Digging for data structures." in *Proceedings of Symposium on Operating Systems Design and Implementation*, 2008.

[31] B. Dolan-Gavitt, A. Srivastava, P. Traynor, and J. Giffin, "Robust signatures for kernel data structures," in *Proceedings of ACM Conference on Computer and Communications Security*, 2009.

[32] FBI, "Brian P. Regan espionage," https://www.fbi.gov/history/famous-cases/brian-p-regan-espionage, 2001.

[33] R. M. Gardner and T. Bevel, *Practical crime scene analysis and reconstruction*. CRC Press, 2009.

[34] T. Garfinkel, M. Rosenblum *et al.*, "A virtual machine introspection based architecture for intrusion detection." in *Proceedings of Network and Distributed System Security Symposium*, 2003.

[35] K. Guðjónsson, "Mastering the super timeline with log2timeline," *SANS Institute*, 2010.

[36] C. Hargreaves and J. Patterson, "An automated timeline reconstruction approach for digital forensic investigations," *Digital Investigation*, vol. 9, pp. S69–S79, 2012.

[37] J. Harrison, "Death of 15-year-old nichole cable was kidnapping gone wrong, affidavit says," http://bangordailynews.com/2013/05/29/news/bangor/death-of-15-year-old-nichole-cable-was-kidnapping-gone-wrong-affidavit-says/, 2013.

[38] C. Hilgers, H. Macht, T. Muller, and M. Spreitzenbarth, "Post-mortem memory analysis of cold-booted android devices," in *Proceedings of IT Security Incident Management & IT Forensics (IMF)*, 2014.

[39] M. Khan and I. Wakeman, "Machine learning for post-event timeline reconstruction," in *First Conference on Advances in Computer Security and Forensics Liverpool, UK*. Citeseer, 2006, pp. 112–121.

[40] J. Lee, T. Avgerinos, and D. Brumley, "TIE: Principled reverse engineering of types in binary programs." in *Proceedings of Network and Distributed System Security Symposium*, 2011.

[41] A. Levinson, B. Stackpole, and D. Johnson, "Third party application forensics on apple mobile devices," in *System Sciences (HICSS), 2011 44th Hawaii International Conference on*. IEEE, 2011, pp. 1–9.

[42] Z. Lin, J. Rhee, C. Wu, X. Zhang, and D. Xu, "DIMSUM: Discovering semantic data of interest from un-mappable memory with confidence," in *Proceedings of Network and Distributed System Security Symposium*, 2012.

[43] Z. Lin, J. Rhee, X. Zhang, D. Xu, and X. Jiang, "SigGraph: Brute force scanning of kernel data structure instances using graph-based signatures," in *Proceedings of Network and Distributed System Security Symposium*, 2011.

[44] Z. Lin, X. Zhang, and D. Xu, "Automatic reverse engineering of data structures from binary execution," in *Proceedings of Network and Distributed System Security Symposium*, 2010.

[45] H. Macht, "Live memory forensics on android with volatility," *Friedrich-Alexander University Erlangen-Nuremberg*, 2013.

[46] P. Movall, W. Nelson, and S. Wetzstein, "Linux physical memory analysis." in *Proceedings of USENIX Annual Technical Conference, FREENIX Track*, 2005.

[47] J. Olsson and M. Boldt, "Computer forensic timeline visualization tool," *digital investigation*, vol. 6, pp. S78–S87, 2009.

[48] N. L. Petroni Jr, T. Fraser, A. Walters, and W. A. Arbaugh, "An architecture for specification-based detection of semantic integrity violations in kernel dynamic data." in *Proceedings of USENIX Security Symposium*, 2006.

[49] N. L. Petroni Jr, A. Walters, T. Fraser, and W. A. Arbaugh, "FATKit: A framework for the extraction and analysis of digital forensic data from volatile system memory," *Digital Investigation*, vol. 3, 2006.

[50] B. Saltaformaggio, R. Bhatia, Z. Gu, X. Zhang, and D. Xu, "GUITAR: Piecing together android app GUIs from memory images," in *Proceedings of ACM Conference on Computer and Communications Security*, 2015.

[51] ——, "VCR: App-agnostic recovery of photographic evidence from android device memory images," in *Proceedings of ACM Conference on Computer and Communications Security*, 2015.

[52] B. Saltaformaggio, R. Bhatia, X. Zhang, D. Xu, and G. G. Richard III, "Screen after previous screens: Spatial-temporal recreation of android app displays from memory images." in *Proceedings of USENIX Security Symposium*, 2016.

[53] B. Saltaformaggio, Z. Gu, X. Zhang, and D. Xu, "DSCRETE: Automatic rendering of forensic information from memory images via application logic reuse," in *Proceedings of USENIX Security Symposium*, 2014.

[54] B. Schatz, G. Mohay, and A. Clark, "Rich event representation for computer forensics," in *Proceedings of the Fifth Asia-Pacific Industrial Engineering and Management Systems Conference (APIEMS 2004)*, vol. 2, no. 12. Citeseer, 2004, pp. 1–16.

[55] A. Schuster, "Searching for processes and threads in microsoft windows memory dumps," *Digital Investigation*, vol. 3, 2006.

[56] A. Slowinska, T. Stancescu, and H. Bos, "Howard: A dynamic excavator for reverse engineering data structures," in *Proceedings of Network and Distributed System Security Symposium*, 2011.

[57] M. Stoer and F. Wagner, "A simple min-cut algorithm," *Journal of the ACM (JACM)*, vol. 44, no. 4, pp. 585–591, 1997.

[58] The Volatility Foundation, http://www.volatilityfoundation.org/.

[59] V. L. Thing, K.-Y. Ng, and E.-C. Chang, "Live memory forensics of mobile phones," *Digital Investigation*, vol. 7, 2010.

[60] R. Walls, B. N. Levine, and E. G. Learned-Miller, "Forensic triage for mobile phones with DEC0DE," in *Proceedings of USENIX Security Symposium*, 2011.

[61] J. Zeng, Y. Fu, K. A. Miller, Z. Lin, X. Zhang, and D. Xu, "Obfuscation resilient binary code reuse through trace-oriented programming," in *Proceedings of ACM Conference on Computer and Communications Security*, 2013.