An OS-agnostic Approach to Memory Forensics

Andrea Oliveri
EURECOM
andrea.oliveri@eurecom.fr

Matteo Dell'Amico
University of Genoa
matteo.dellamico@unige.it

Davide Balzarotti
EURECOM
davide.balzarotti@eurecom.fr

***Abstract*—The analysis of memory dumps presents unique challenges, as operating systems use a variety of (often undocumented) ways to represent data in memory. To solve this problem, forensics tools maintain collections of models that precisely describe the kernel data structures used by a handful of operating systems. However, these models cannot be generalized and developing new models may require a very long and tedious reverse engineering effort for closed source systems. In the last years, the tremendous increase in the number of IoT devices, smart-home appliances and cloud-hosted VMs resulted in a growing number of OSs which are not supported by current forensics tools. The way we have been doing memory forensics until today, based on handwritten models and rules, cannot simply keep pace with this variety of systems.**

To overcome this problem, in this paper we introduce the new concept of *OS-agnostic memory forensics*, which is based on techniques that can recover certain forensics information without *any* knowledge of the internals of the underlying OS. Our approach allows to automatically identify different types of data structures by using only their topological constraints and then supports two modes of investigation. In the first, it allows to traverse the recovered structures by starting from predetermined *seeds*, i.e., pieces of forensics-relevant information (such as a process name or an IP address) that an analyst knows *a priori* or that can be easily identified in the dump. Our experiments show that even a single seed can be sufficient to recover the entire list of processes and other important forensics data structures in dumps obtained from 14 different OSs, without any knowledge of the underlying kernels. In the second mode of operation, our system requires no seed but instead uses a set of heuristics to rank all memory data structures and present to the analysts only the most 'promising' ones. Even in this case, our experiments show that an analyst can use our approach to easily identify forensics-relevant structured information in a truly OS-agnostic scenario.

I. Introduction

The analysis of volatile memory is gaining more and more importance as part of digital forensics and incident response investigations. This is because the system memory contains artifacts, related both to the current and the past state of the system, that cannot be extracted from any other component. However, the analysis of volatile memory presents unique challenges, as operating systems (OSs) have diverse internal organization ways they store and represent data, with only a few constraints imposed by the underlying hardware and memory management unit (MMU). Unfortunately, this great flexibility becomes a great challenge for forensic analysis tools. For example, the kernel needs to keep track of all processes running inside the system, including their names, memory layout, loaded libraries, and open file descriptors—just to name a few among the many pieces of information required to track processes during their execution. OSs organize this set of information by using data structures, such as linked lists, trees, and arrays. To extract this data from memory dumps and reconstruct the aforementioned structures, analysts must know the (often undocumented) kernel internals to locate and interpret the raw bytes in the dump. This, which is often called the *semantic gap*, constitutes the main problem of memory forensics.

To solve this problem, forensics tools like Volatility [23] and Rekall [4] maintain collections of precise descriptions of the kernel data structures used by popular OSs (e.g., Windows, Linux, and macOS). Thanks to these OS-specific models, called *profiles*, these tools can extract relevant forensics information from memory dumps. However, profiles need to be constantly updated because the internal data structures used by OSs often change with version updates. Furthermore, for highly configurable OSs like Linux, it is often necessary to create specific profiles for the individual machines that analysts want to investigate. While recent works [16], [18] have partially overcome this problem by reconstructing Linux profiles directly from the memory dumps, these solutions are tailored to the data structure of the Linux kernel–which are supposed to be known in advance.

Memory forensics is a very active research area and has largely improved over the last decade. However, it still relies on manual rules that exist only for a handful of OSs. While this was sufficient in the past, today investigations often involve a broad range of devices, ranging from network routers to smart home appliances and smartwatches, all of which operate on a variety of OSs. For instance, recently Cekerevac et al [2] studied 56 different OSs adopted in the IoT space alone (a list that does not even include OSs used in network appliances, such as CISCO IOS). Sadly, *most* of these systems are currently unsupported by memory analysis tools. Even worse, the current approach is based on a long and tedious human effort to reverse engineer the required OS internal details; this makes extending memory forensics support to a large number of OSs completely impractical.

A. Introducing OS-agnostic Memory Forensics

Currently, available forensics tools can extract information from a memory dump by relying on two techniques: (i) the OS is manually analyzed by the tool writers [4], [23], who

Network and Distributed System Security (NDSS) Symposium 2023
27 February - 3 March 2023, San Diego, CA, USA
ISBN 1-891562-83-5
https://dx.doi.org/10.14722/ndss.2023.23398
www.ndss-symposium.org

manually compiled a set of rules to extract the forensics-relevant information for that OS (ii) the OS is instrumented (e.g., by analyzing its source code [1], [6], [9], [10] running it on instrumented VMs [5], [11], or by taking snapshots in different states [19], [22]). The first rule-based approach, which is the standard among memory forensic tools, requires months of work to add support to a new operating system. The second solution, proposed by researchers but not commonly used in practice, requires instead complete and privileges access to the target systems. None allows an analyst to recover any information from a memory dump extracted from a device running an OS that is either unknown (no source code, no installation disks or the possibility to run it on a VM) or unsupported by ordinary forensics tools. This situation can occur when, for example, memory is acquired from IoT or industrial devices, network appliances, or VMs that are running uncommon OSs. This is why memory forensics today does not exist for such cases.

To solve this problem, we propose a paradigm shift in the way we perform memory forensics. While profiles, custom rules, and dynamic introspection remain valuable solutions that are likely to provide better results when they can be applied, we believe a new approach is needed to quickly extend memory analysis to a broader class of target systems. If today the field is driven by a *complete knowledge* of the internal data structures, we propose instead the first step towards what we call *OS-agnostic memory forensics*, which would allow analysts to perform memory forensics without *any* information about the underlying OS.

Our approach relies only on OS-agnostic properties of the data structures used by kernels to organize data. In fact, like all software, OS kernels store information in a collection of data structures that can be detected and reconstructed due to their topological properties. As shown by Oliveri and Balzarotti [15], for most CPU architecture it is possible to reconstruct the kernel address space in an OS-agnostic way by using only information derived from the hardware configuration of the machine. It is by using this information that we reconstruct in-memory kernel data structures like linked lists and trees.

In this paper, we first propose new algorithms to identify forensics-relevant data structures in a fully-automated way. As forensics-relevant we consider all those data structures which contain, or point to, the information needed during a forensics analysis, thus excluding data structures that are used by the OS only for its basic functioning (e.g., data structures for hardware management, communication among its internal parts, synchronization, etc.). Our technique takes as input a single memory dump and a function to extract pointers from raw data. It then, without human intervention, extract forensics-relevant data structures based on their topological properties. We investigate two complementary approaches to perform this task. In the first, we start by observing that the vast majority of data structures are irrelevant from the point of view of a forensics analysis. Forensic-relevant data structures can be identified because they reference or embed specific *seeds*, which are pieces of information that an analyst knows *a priori* or can be easily carved from the memory. Examples of seeds are the name of a process, an IP address, or a file name. Seeds are important because the analyst can use them as "anchors" to filter data structures and then automatically extract information of the same type. For instance, knowing just one process name can be sufficient to (i) identify a data structure (such as a doubly-linked list) that points to it, and (ii) explore all the elements of that structure and print the strings that are referenced by pointers located at the same offset in the data structure. Even with no previous knowledge of the OS, this automated procedure allows a forensic analyst to list all process names–once one of them is known.

The second approach we explore is a seed-less analysis. Once again with no prior knowledge of the OS, we show it is possible to retrieve structured forensics information from a memory dump to be used as a bootstrap for advanced analysis. For instance, by filtering and ranking the extracted data structures according to the statistical properties of their fields, an analyst can use our system to look for data structures referring to forensics-relevant strings. We will show that this approach can retrieve most of the data structures retrieved with the first approach, with the advantage of being applicable when analysts do not known valid seeds.

We implemented our techniques in a proof-of-concept tool, *Fossil*, and we performed experiments to retrieve data structures from 14 different operating systems. Our results show that our OS-agnostic approach to memory forensics can provide invaluable information to analysts facing memory dumps taken from less known, or unknown, OSs. We will release Fossil as an open-source project [14], together with the part of the dataset not covered by particular OS license restrictions.

II. Data Structures in Memory Forensics

Kernels organize information in a variety of data structures (DSs) which can, in turn, be implemented in many different ways. DSs like linked lists, doubly-linked lists and n-ary trees are all based on logical relations among components linked through pointers. On the other hand, structures that require fast access like arrays, stacks and queues are based instead on data locality. As shown in the left part of Figure 1 even a very simple DS like a linked list can be implemented in different ways: pointers among items of the list can refer to the top of the adjacent elements or they can point to a fixed offset within them, data can be stored as part of the items themselves or in auxiliary structures referenced by pointers, and the pointer that signals the end of the linked list can be a `NULL` pointer or a pointer that points to itself. The more complex the structure is, the greater the chances that an OS will implement it by using unique features and tricks, to maximize performance.

Furthermore, multiple "basic" DSs can be combined to form more complex ones. For example, hash tables can be implemented as an array of pointers to linked lists and filesystem-related structures often use *tries*, which are a type of search tree whose nodes have a variable number of children. In this case, a node's children could be represented as a linked list, as shown in the right part of Fig. 1.

Since it is impossible to cover all possible DSs and all possible ways these structures *might* be implemented in OSs, we need to establish a hierarchy based on their forensics-relevance. In fact, complex DSs may be extremely rare, they may not contain forensic-relevant information, they may be

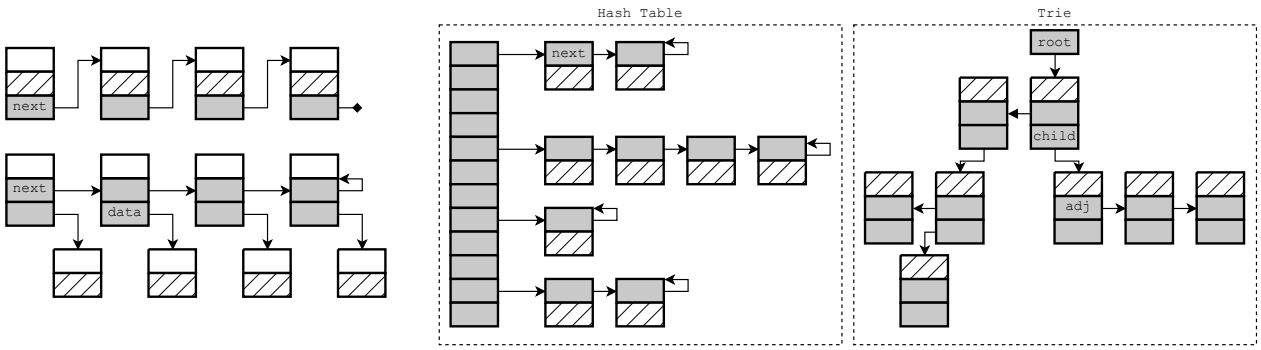

Fig. 1: On the left side, two examples of different implementations of linked lists. On the right side, two examples of complex data structures: a hash table and a trie implemented using arrays and linked lists. Grey cells are pointers while barred ones contain data.

used only as a fast way to access already organized data, or they might be implemented in a very optimized way which requires deep knowledge of the OS code to be recovered from the dump. At the same time, simple and easy-to-manage DSs could be dominant in fundamental tasks of the OS which do not require leading performance but great stability and easy usage. To guide our selection of DSs that are more commonly traversed during memory forensics, we perform a survey on which of them are required by the Volatility framework [23] to extract information for Linux and Windows memory dumps.

Out of all existing plugins, 79% traverse doubly-linked lists, 23% use arrays, 10% explore trees and 12% extract information from simple linked lists. In the majority of the case (73%) the required information is stored inside the DS itself or is contained inside auxiliary C structs pointed by the major DS.

Only 10 of the Volatility plugins require the parsing of more complex data structures, like hash tables, radix trees, and tries. Furthermore, these data structures are used in prevalence to access data that can be also extracted by traversing other DSs, such as the radix tree that Linux uses to connect the process PIDs.

Based on this preliminary study, we decided to focus our techniques on the recovery of single and doubly-linked lists, arrays, and trees. Other DSs could be added in the future, but we believe they can only provide a marginal amount of information during a forensics analysis.

III. Terms and Definitions

Before we describe our technique for DS reconstruction we need to define some terms that we will use over the rest of the paper.

A. Atomic Structures

Atomic structures represent the fundamental blocks to store composite information as in-memory ordered collections of fields of different types and sizes. In languages of the C-family, atomic structs are directly mapped to the `struct` data type. In memory, atomic structs are contiguous chunks of data,[1] which contain fields defined in the structure itself along with possible padding inserted by the compiler to optimize the structure size according to the CPU architecture and optimization requirements. Despite having a clear size and a strict separation among its fields at the source code level, in memory an atomic struct appears as a block of bytes without any field separator nor any delimiter to separate the structure itself from other adjacent data.

While an atomic struct can contain other atomic structs embedded in it, in memory there is no distinction between the fields of the embedded atomic struct and the fields of the embedding one. These ambiguities have important effects on the ability to reconstruct an atomic struct layout starting only from its content in memory.

B. Data Structures (DSs)

The OSs use atomic structs and atomic types (such as `int`, `float`, and `struct X *`) as basic blocks to store information by organizing them in in data structures.

Linked lists are sequences of atomic structs (the nodes of the list) connected by pointers. Starting from an element it is possible to reach the next one by following a pointer (here generically called *next*) contained in the element itself. It is important to note that the *next* pointer can point either to the first address of the next element of the linked list or at a fixed offset inside it. Generally the last *next* pointer is a `NULL` pointer or an autopointer (a pointer pointing to itself, see Section III-C) if the linked list is linear. The linked list may also be *circular*; in that case, the last element of the list points back to the first. Like other structures connected by pointers, the atomic structs of linked lists are not necessarily allocated in contiguous memory locations.

Doubly-linked lists are similar to linked lists. However, elements are linked by two pointers: *next* and *previous* (not necessarily located at adjacent offsets), with the latter pointing to the previous element of the list. A doubly-linked list can be seen also as a combination of two

[1] If the system uses virtual memory the atomic struct is a contiguous chunk of data only in the virtual address space, but can be fragmented in physical memory.

linked lists: a linked list composed of the element joined by *next* pointers and a second list containing the same elements but linked in reverse order through the *previous* pointers. As for linked lists, doubly-linked lists can be linear or circular. It is important to note that the presence of two pointers linking every element of the list introduces additional information (in the form of a constraint) that helps to recognize doubly-linked lists in memory.

Trees are DSs that may exist in different forms (balanced or unbalanced, RB-trees, etc.). For our purposes, they can be generalized by a generic `n`-ary tree model. In this model, a tree is composed of a root atomic struct linked to a predefined maximum of `n` subtrees, each composed, in turn, by a collection of the same type of atomic structs. At the end of each branch is possible to find leaves that could have a different type of atomic struct.

Arrays are DSs in which atomic types or atomic structs are located in adjacent memory locations. This type of DS can be very difficult to detect if the atomic type is not a pointer or if the atomic structs contains only non-pointer types because the sequence of elements becomes indistinguishable from an unstructured binary data blob.

Composite data structures provide a way to combine multiple simpler DSs to form more complex ones, such as hash tables and tries.

C. Pointers

A pointer variable is characterized by two memory addresses: the address of the pointer itself and the address referenced by the pointer. For our goal, we can classify pointers into four categories:

Structural pointers. These pointers represent links between atomic structs that compose a DS or links among different data structures.

Data pointers. These pointers, contained in atomic structs, reference variables, auxiliary atomic structs which are not part of a DS but contain information related to it (e.g. `struct pid *thread_pid` in the Linux `task_struct` which references a structure containing the kernel's internal notion of a process identifier), or relations among atomic structs composing the data structure but which do not define its topology (e.g. the `struct task_struct *parent` in Linux `task_struct`).

Autopointers. These pointers have the peculiarity to point to themselves (`*ptr == &ptr`). This property allows to easily identify them inside a memory dump and to signal the presence of a field in the atomic struct which, with high probability, has been deputed to point to linked list-like DSs.

`NULL` pointers. `NULL` pointers could be used to signal the end of a linear DS (as a linked list) or part of a ramified one (a tree). We assume that the OSs and CPU architectures use all-zeros CPU-native words to represent `NULL` pointers. This fact makes the `NULL` pointer virtually indistinguishable from a zero integer variable.

D. Oracle Function Ω and Γ_x graphs

To identify atomic structs which compose larger DSs it is necessary to retrieve all pointers used by the kernel to maintain relations among single elements. Without any information about the OS and its internals the only way to extract all the pointers from a memory dump is through carving.

In particular, starting from the CPU architecture it is possible to derive the size α of a pointer[2] (e.g. 4 bytes for 32-bit architectures and 8 bytes for 64-bit ones). We suppose the analyst is able to provide a boolean oracle function Ω (see Section V for the implementation we adopted in our evaluation), which takes as input an offset in the dump, the pointer size, and other OS-independent/machine-dependent parameters and returns whether that offset contains or not a possible valid pointer:

$$\Omega(\text{offset}; \alpha, \dots) = \begin{cases} \text{True} & \text{if offset is a valid pointer} \\ \text{False} & \text{otherwise} \end{cases} \tag{1}$$

The Ω function allows us to scan the entire memory dump and identify all valid pointers by iterating through overlapping groups of α bytes: the offset in the memory dump (properly interpreted in relation to the physical-to-virtual address mapping) represents the address of the pointer variable while the content of the α bytes located at that offset represent the address of the pointed data. This technique allows to identify both pointers stored at an aligned address (aligned pointers) as well as unaligned ones. It is important to note that we do not specify the concrete implementation of the Ω function: for our technique it is only a prerequisite necessary to extract the pointers, i.e., the analyst has to provide such OS-agnostic function or a function that best approximates its behavior. Furthermore, once the pointers have been extracted, our technique no longer requires information about the hardware configuration of the machine thus also resulting CPU-agnostic.

All pointers that are identified in a memory dump compose a directed graph Γ. In this graph, a pointer (`pointer = &variable`) is represented as an edge connecting the nodes representing the address of the pointer (`&pointer`) to the address pointed by it (`&variable`).

Finally, we define the set of offset graphs $\{\Gamma_x\}_{x \in [X_{min} \dots X_{max}]}$ as the set of graphs obtained by adding a fixed offset of `x` bytes (positive or negative) to all the destinations of the carved pointers (`pointer + x`). For example, $\Gamma = \Gamma_0$, and Γ_{64} is the graph in which each pointer destination is increased by 64 bytes.

E. Seeds

We define a *seed* as a chunk of data that is either known *a priori* by the analyst or that can be easily recognized as a valid piece of information by inspecting the memory. Seeds can be extracted directly from the memory dump by using OS-agnostic rule-based carving techniques. Examples of seeds are: the name of a process the analyst knows was running when the dump was acquired, a path in the filesystem, the name of a kernel module, an IP address assigned to a network interface, etc. Strings, for instance, can be easily extracted from kernel memory by looking for sequences of ASCII or UTF8/16 characters. The same technique can be used also to extract other information with an immutable and OS-independent structure and encoding such as network packets.

[2] α for modern CPUs is also, in general, the "natural" word size and alignment used in memory access.

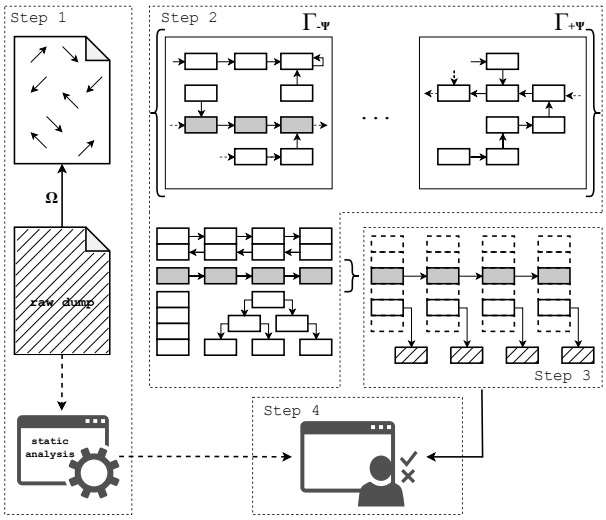

Fig. 2: OS-agnostic DSs reconstruction phases. Grey cells are pointers of a valid DS while barred ones contain data.

IV. Approach

The goal of our work is to extract forensics-relevant information about the state of the system from a memory dump without any prior knowledge about the OS.

An overview of the process is shown in Figure 2. In the first step, we use the provided Ω function to collect the kernel pointers present in the dump. We then locate the seeds by using OS-agnostic data carving and we (optionally) perform a static analysis on the executable pages of the kernel. This analysis extracts cross-references to data structures allocated by the kernel at runtime, which helps our system to prioritize the extracted information and reduce the number of false positives.

After that, we reconstruct relations among pointers (Step 2 in Figure 2) by using topological properties of the DSs that we are interested to recognize (discussed in detail in Section IV-A). In this phase we reconstruct only the structural skeleton of the DSs: relations among structural pointers do not provide information about the shape (size and limits) of the atomic structs which contain data-pointers and embed forensic-relevant data.

After the skeleton of the DS is recognized, we perform a statistical analysis of the in-memory raw data that surrounds each structural pointer (Step 3) estimating how many bytes the original atomic DS extends around them. This allows us to identify also relations among reconstructed data structures and seeds.

Finally, we present to the analyst the identified DSs with the possibility to filter them by looking for specific known seeds or by using cross-references extracted during the static analysis phase (Step 4).

A. Data Structures Skeleton Recognition

Once pointers are recognized (Step 1 in Figure 2), we tackle the task of recognizing the skeleton of DSs. While our approach can be useful to recognize also more complex data structures, as explained in Section II our current implementation focuses on the most common DSs that are traversed by memory forensics frameworks.

1) Relation between Lists, Trees and Offset Graphs: Consider the case of lists (both single- and doubly-linked) and trees. These DSs are composed of sequences of atomic structs of the same type linked by structural pointers. Unfortunately, it is quite common to see structural pointers whose destination is not the beginning of the next atomic structure, but rather some point in the middle—e.g., the `next` pointers used in doubly-linked lists in the Linux kernel point to the `next` pointer of the successor's struct [12]. In other cases, e.g., the `prev` pointer in Linux's doubly-linked lists, the destination is at a fixed offset with respect to the same pointer in the adjacent struct (i.e., the `prev` pointers point to the address of `next` pointers of the previous struct).

To generalize, atomic structs part of the same DS are reached by dereferencing a structural pointer, adding a (positive or negative) offset β to the destination, and by repeating the process while keeping the offset β constant. With reference to Linux's doubly-linked lists: $\beta = 0$ for the `next` pointers and $\beta = +\alpha$ for `prev` ones. This process is shown in Figure 3 along with the paths it produces in the Γ_β graphs. Hereinafter, we call these paths *chains*.

A chain may correspond to the collection of all the structural pointers of a linked list or parts of other DSs, such as "half" of a doubly-linked list or the branch of a tree obtained by always following a given child link (e.g., all *left* or *right* children). However, this procedure generates also false positives, as shown in Figure 3. In fact, following a pointer and adding an arbitrary offset x to it could, by chance, lead to another pointer and so on. This produces a valid path on a Γ_x graph that does not correspond to any valid DS.

A first step to reduce false positives is to note that in Γ_β the atomic structs that belong to true-positives DSs should be at least β bytes large (because otherwise by adding the offset β we would "fall outside" of the data structure). Moreover, atomic structs should also be larger than the pointer size α. Hence, pointers in a valid chain should all be at least at distance $\max(\beta, \alpha + 1)$ from each other. Chains of atomic structs that do not satisfy this requirement can be discarded, as the path from $\texttt{aux}_A$ to $\texttt{prev}_A$ in Γ_α graph in Figure 3.

In practice, we compute the offset graphs for a suitable symmetric interval of plausible offsets $[-\Psi, \Psi]$. Ψ is an important parameter that affects the maximum size of atomic structs identifiable by our technique. In fact, in the worse case in which a structural pointer is located at the bottom (last field) of an atomic structure and references the top (first element) of the next element, then the maximum size of the atomic structure that can be retrieved by our approach is Ψ.

It is worth noting that each offset graph has a topological peculiarity: since each memory location contains at most one pointer, the maximum out-degree in the graph is one. In other words, there is at most one outgoing edge from each vertex. On the other hand, several pointers may have the same destination. This implies that paths in an offset graph may converge; however, there is at most a single outgoing path from any node of the graph. In light of this, chains may contain tail links back to the head forming a cycle, which corresponds to

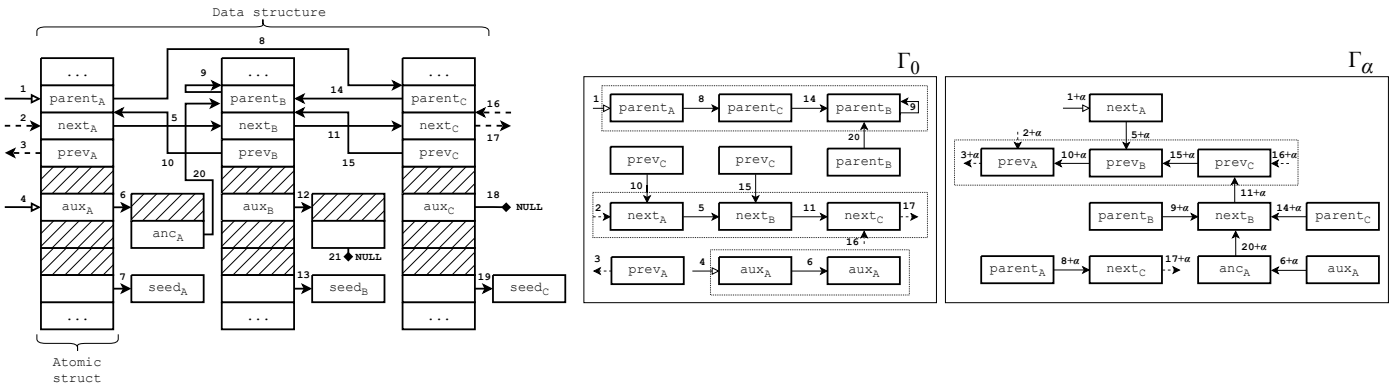

Fig. 3: Example of DS and its partial contribution to two examples Γ graphs. Pointers with empty arrows are false positives due to approximations of the Ω function. Dashed pointers start from structs not represented here. Note pointer 9, an autopointer, and pointer 18 and 21 `NULL` ones. Pointers 6 and 12 are data-pointers (referring to auxiliary atomic structs) as also 7, 13 and 19 (referring to seeds). Cells with diagonal bars contain data. Only contoured chains correspond to meaningful paths and only `next` and `prev` paths involve structural pointers.

circular lists, but also a "P-shape", where the end of the chain links back to a point in the middle of it. We "cut" chains like these into two separate chains–one circular and one linear (up to the beginning of the cycle).

2) Doubly-Linked Lists: A doubly-linked list consists of a set of atomic structs with *previous* and *next* structural pointers and, like singly-linked lists, may be circular. We allow the destinations of the *previous* and *next* links to be at different offsets: for example, *next* pointers can point to the following *next* pointer without the need to apply an offset, and *previous* pointers can point to the preceding *next* pointer.

We identify doubly-linked lists by recognizing chains that are at a fixed distance and in opposite directions: consider the example of a doubly-linked list containing atomic structs A, B, and C; one chain follows the path $A \rightarrow B \rightarrow C$, while the other takes the reverse direction $C \rightarrow B \rightarrow A$. Unfortunately, the generalization of this approach to find doubly-linked lists whose chains point at different offsets in the struct introduces several computational challenges. To solve the issue we proposed a new algorithm, based on bi-directional hashes, explained in detail in the Appendix section.

3) Trees: Our algorithm recognizes trees by following a bottom-up approach (here for simplicity we illustrate the case of binary trees, however, our approach naturally generalizes to n-ary trees with $n > 2$). We first look for complete trees of depth two by checking the intersection of chains that have a length of two: in this case, they would intersect at a point r, the candidate root, that appears in two different offset graphs, Γ_l, and Γ_r, where we say that offset l reaches the *left* child and offset r reaches the *right* one. Then, we perform two checks to verify whether we should discard the candidate tree with root in r: first, we discard all cases in which there is no pointer at offset r from the *left* child or at offset l from the *right* child; second, we verify that structs in the candidate tree are distant enough from each other to create a coherent tree (i.e., where the distance respects a lower bound for the size of the struct representing the tree node). We know that this lower bound t should respect the following conditions: (i) $t > 2\alpha$, because the struct needs to contain the pointers to the two children and some payload data; (ii) $t >= \max(|r-l|, |l|, |r|)$, because the destination of the pointers plus the left and right pointers should all fall within the node struct. Hence, we discard all candidate trees for which any two nodes are at a distance that doesn't satisfy this constraint.

Fully balanced trees of arbitrary depth are then constructed recursively by recognizing height-2 trees whose root's children are trees of height $x - 1$ and whose nodes are still at least as distance t from each other as described above. Non-fully balanced trees can be simply obtained by following the l and r links of the trees that we have discovered.

4) Arrays: As already explained in Section III-B, this type of DS can be undetectable if the atomic type is not a pointer and without any knowledge of the array shape or its content. In light of this, we decided to carve only for arrays of pointers, by looking for sequences of in-memory adjacent not `NULL` pointers.

B. Atomic Structs Size Estimation

The techniques described above allow the reconstruction of relations among structural pointers which maintain links among atomic structs and determine the topology of the DS. However, to extract forensics-relevant information from atomic structs we first have to infer their boundaries, i.e., the offset at which each atomic structure starts in relation to its structural pointers and its total size. Since we have no information on the OS internals, we can rely only on statistical properties derived from the set of all the atomic structs which compose the DS. The idea is that fields located at the same offset in the struct contain data of the same type.[3] This phase is important because an under-estimation of the atomic struct size could exclude fields that refer to seeds easily recognizable by the

[3]We are ignoring C-style `unions` which are very difficult to detect without any knowledge of their definition and represent only a small part of the type of atomic struct in kernels (for example in Linux `unions` represents only 4% of the total number of the types of atomic structures definitions).

analyst, while an over-estimation could return an avalanche of false positives seeds complicating the analyst's work.

V. Implementation

In this section, we explain how we implemented the DSs recognition techniques in our proof-of-concept tool, Fossil, how we reduced the number of false positives by introducing OS-agnostic heuristics, and the dataset used for experiments.

A. Ω Function

The role of the Ω function is to identify values that can be valid kernel pointers in the memory dump. Unfortunately, without any additional information about the OS, this is a very difficult task, as potentially any number could be a valid kernel address. Luckily, the system architecture (which we assume to be known) can provide useful information. For instance, on micro-controllers with a 32-bit memory address range, MMIO, and no virtual memory (e.g. PIC32, or Renesas RX) it is possible to drastically reduce the values of valid memory addresses and build a very precise Ω function. On CPU architectures with a memory protection unit (MPU), such as ARM Cortex-M, it is possible to infer which regions of the RAM are writable and could contain valid pointers.

Oliveri and Balzarotti [15] developed an OS-agnostic technique based on the functioning of MMUs, which is able to automatically retrieve the kernel virtual address space from the memory dump for Intel x86/AMD64, ARM/AArch64 and RISC-V 32/64-bits CPU.[4] This allows us to define an approximated oracle function that marks all valid virtual addresses of the kernel address space as potential pointers by scanning the kernel pages contained in the memory dump looking for sequences of α bytes which can be interpreted as valid virtual memory address belonging to the virtual address space of the kernel. This, however, does not guarantee that the discovered sequences of α bytes correspond to real pointers: a group of continuous α bytes that, accidentally, can be interpreted as a valid kernel virtual address, produces a false positive. However, this approach has no false negatives because structural pointers belonging to kernel DSs must reside in the kernel virtual address space to be reachable and accessible by the kernel itself. It is also possible to use other metadata available in the kernel radix-tree's page tables entries to further restrict and refine the Ω function. For instance, we limit our analysis to those memory regions exclusively accessible by the kernel and denied to userspace programs because fundamental DSs maintained by the kernel must be protected from the possible corruption of a malicious user-space process. While the extraction of the kernel address space depends on the architecture [15], our technique does not and it is completely CPU-independent.

B. Dataset

To test the accuracy and performance of our OS-agnostic DS recognition technique we assembled a dataset of memory dumps taken from virtual machines (VMs) running 14 different OSs. The majority of these systems are running on x86/AMD64 (due to the abundance of the available OSs for this architecture) but we also included two OSs running on AArch64: a Linux machine and an iPhone running iOS. We specifically included the Linux Debian machine for both AMD64 and AArch64 architectures to compare the results and check whether our technique introduces some bias, and iOS as an example of a real and complex OS target that is not supported by other forensic tools. The experiments are conducted on virtual machines to easily collect atomic snapshots of the physical memory and avoid inconsistencies [17]. Excepting for the iOS VM, which requires 6 GB to run, each VM was configured with 4GB of RAM because the x86 32-bit architecture cannot address more memory and, at the same time, this is the typical memory size of many IoT devices.

The fourteen OSs, shown in Table I, include general-purpose, embedded, mobile and amateur projects and adopt different designs and architectures (including hybrids, monolithic, real-time, and micro-kernels). For our tests, we decide to focus on OSs written in C or C++ due to their overwhelming prevalence and to the 1-to-1 mapping of atomic DSs to the `struct` construct present in these languages. There have been some attempts to write prototype OSs in other languages, such as Rust or Haskell. However, these languages envelope atomic DS into more complex constructs for either security (Rust) or abstraction (Haskell), requiring a preliminary understanding of the language constructs to perform atomic DS mappings.

C. Static Code Analysis

To extract global references to kernel DSs, we perform a static analysis of the executable pages located in the kernel virtual address space. Using the tool developed by Oliveri and Balzarotti [15] we reconstruct the kernel address space and virtual pages' permissions from the memory dump and save them as an ELF core file. Then we analyze the ELF with a Ghidra [13] plugin, which disassembles the instructions and performs a simple static analysis to identify the address of all the global variables referenced in the code.

D. False Positives Reduction

As discussed in Section IV-A1, chains on Γ graphs are directly related to the skeleton of different DSs. However, as shown in Figure 3, a large number of links are in fact false positives and therefore we employ some heuristics to clean the extracted data.

To begin with, we define a hierarchy among DSs in relation to how much strict their topological constraints are since it is less probable that false positives are generated if topological constraints are strict. Doubly-linked lists have the strictest constraints requiring a fixed distance between the structural pointer which composes the `next` and `prev` chains as already explained in Section IV-A2 and in the Appendix. Binary trees have looser constraints in comparison with doubly-linked lists as they only require that each node has two valid (and not both `NULL`) pointers at a fixed distance from the address pointed by the parent node. Looser constraints are required for arrays, for which we only require more than two consecutive pointers. At the lower end of the scale we have linked lists, for which there is only the requirement to represent paths in Γ subgraphs, and sets of auxiliary atomic structs which are obtained by following

[4] Authors also show that, for some CPU architectures, a full OS-agnostic reconstruction of the kernel virtual address space is not possible (PowerPC) or requires a manual effort from the analyst (MIPS).

data pointers at a fixed offset of already discovered DSs. With this hierarchy in mind, we assume that structural pointers that our technique associates with a DS with tighter constraints cannot be a structural pointer of looser constrained DS (in other words, we assign each structural pointer to only one, the most constrained, DS).

We then proceed to filter each type of DS individually. The heuristics we use are often based on the locality of data and fields (e.g., the fact that right and left pointers in a tree are logically placed closer to each other in the struct definition) and use thresholds that can be configured and fine-tuned by the analysts.

1) Doubly-linked lists: As explained in Section IV-A2 linear and circular doubly-linked lists can be seen as a couple of chains belonging, in general, to two different Γ offsets graphs. Supposing that the implementation of the doubly-linked list used by the kernel is unique we consider as valid linear and circular doubly-linked lists those composed of chains with the most abundant pair of offsets among the doubly-linked lists reconstructed by our tool, filtering out possible false positives DSs based on chains that accidentally match.

2) Trees: We assume that the `left`, `right` and (optionally) the `parent` pointers are close inside the node structure. Thus, our tool looks for binary trees made of chains obtained from structural pointers extracted from Γ graphs with offsets in $[-8\alpha, 8\alpha]$ and composed of at least 2 levels (7 nodes, including the root one).

3) Arrays: We consider an array to be valid only if it contains at least three non-`NULL` consecutive elements, not already used for previous DSs. Moreover, as arrays are abundant as part of other structures, we only report those that are directly referenced by the kernel code (e.g., through global variables) as identified by our static analysis.

4) Linked Lists and Sets of Auxiliary Atomic Structs: These two classes of DSs are the most prone to generating false positives. In fact, every pointer can be a valid starting point of a linked list and every field containing pointers of an already discovered DS can be the root of a set of auxiliary structs. Therefore, for linked lists we apply the same heuristic we used for arrays, limiting the report to those structures that are referenced in kernel code. Furthermore, assuming that the implementation of doubly-linked lists are based on the single linked lists one, we consider only linked lists generated starting from Γ graphs with offsets belonging to the most abundant couple of offsets used to generate doubly-linked lists, and of length greater than 2. For sets of auxiliary atomic structs, we consider only fields of atomic struct of higher level which contains at least 90% of not `NULL`.

E. Size of Atomic Structs

To estimate the boundaries of each atomic DS we analyze the area of memory surrounding the pointers identified in the previous steps. First of all, we assume that an atomic struct can contain a maximum of 1024 fields of size α. This results in DSs with a maximum size Ψ of 4KB for a 32-bit architecture and 8KB for 64-bit. This parameter is necessary because otherwise the entire memory could be considered as a single atomic struct.

We then estimate the size of the atomic struct which belongs to the same DS as:

$$\Psi_{\text{ex}} = min(\Psi, mode(|p-q|))\forall p, q \in \{\text{structural pointers}\}$$

In other words, we consider the minimum between the chosen maximum Ψ value and the typical distance among the structural pointers. This estimation is based on the assumption that allocations of atomic DSs are not isolated events and/or the kernel uses an optimized memory allocator (as SLAB system in Linux kernel). If this is true, at least two atomic structs of the same DS are allocated contiguously in memory, so the distance among their structural pointers is an upper limit for the size of the atomic struct itself. However, instead of considering the minimum distance among all the structural pointers as an estimation of the size, we consider the more conservative limit given by the typical distance among them. This allows for cases in which one of the atomic structs has a smaller size (as is the case for the first element of various doubly-linked lists in the Linux kernel).

After the size of the atomic struct has been estimated, we have to determine its alignment. In fact, we have an estimation for the atomic struct size but we don't know yet what is the offset of structural pointers inside the atomic struct or, in other words, where the atomic struct begins. Before proceeding to the alignment estimation we have to determine which offsets (aligned or not) contain valid pointers in the range of $[-\Psi_{\text{ex}}, \Psi_{\text{ex}}]$ around structural pointers (the intervals contain all possible positions of the Ψ_{ex} window). We consider an offset as containing pointers if at least the 90% of the atomic structs of the DS contain a valid pointer or a `NULL` element at that offset. We use a threshold because, as already said, some atomic structs can have a different shape due to their functions (the head of a list or a root of a tree), and because the Ω function can introduce false positives (data bytes interpreted as pointers).

To estimate the correct alignment we rely on two heuristics. The first one is based on the presence of data pointers that maintain auxiliary links among structs. An example of it is the `struct task_struct *parent` in Linux `task_struct`, which points to the start of the atomic struct of the parent process. Checking if at a fixed offset at least 90% of the pointers refer to addresses which distance the same quantity from one of the structural pointers allows to identify them. If all these pointers reference the first byte of another atomic struct of the same DS we can determine the exact alignment of the structure in the Ψ_{ex} window. The second heuristic is used if the first fails and it considers the minimum and the maximum offset which contain valid not-`NULL` pointers, for at least 90% of the atomic structs compatibly with the atomic struct size already estimated.

VI. Data Structure Recovery

Table I reports, for each OS, statistics about the pointers recovered by the Ω function used in our experiments, highlighting noteworthy values. The number of pointers discovered by the Ω function largely depends on the size of the pointer type (α) and the main language used to write the OS. Columns 6 and 7 of Table I show that for 32-bit OSs, our approach returns the highest number (both absolute and relative to the size of the kernel address space) of pointers. This abundance

TABLE I: OSs dataset and recovered pointers.

OS					Pointers		
	Kernel type[1]	Open-source	Main language	Pointers size (α)	Total	Pointers over total locations ratio%	Autopointers over pointers ratio %
Darwin	H	●	C	8	9.7×10^5	< 0.1	3.3
Embox	R	●	C	4	3.9×10^5	0.15	< 0.1
FreeBSD	M	●	C	8	2.2×10^6	< 0.1	< 0.1
HaikuOS	H	●	C++	4	1.76×10^7	8.99	< 0.1
HelenOS	m	●	C	8	4.4×10^5	< 0.1	6.2
iOS (AArch64)	H	●	C	8	1.5×10^6	< 0.1	0.8
Linux	M	●	C	8	5.6×10^6	0.12	20.99
Linux (AArch64)	M	●	C	8	5.5×10^6	0.25	21.39
NetBSD	M	●	C	4	5.1×10^6	5.33	< 0.1
ReactOS	m	●	C	4	3.5×10^6	1.47	0.3
ToaruOS	H	●	C	8	6.7×10^5	< 0.1	0
vxWorks	R	○	C	8	2.1×10^5	< 0.1	0.8
Windows XP	H	○	C	8	9.9×10^5	0.27	16.89
Windows 10	H	○	C	8	1.9×10^6	0.15	6.3

[1] H: hybrid-, m: micro-, M: monolithic-, R: real-time kernel

is due to the high number of data bytes whose value falls into the kernel address space and thus can be confused with real pointers. On the other hand, the Intel 64-bit architecture (as well as AArch64) requires kernel addresses to have the higher bits set to 1, which introduces a strong constraint on which 8-ple of bytes could represent valid addresses.

Furthermore, the programming language also affects the number of pointers present in the kernel address space. As shown in Table I and highlighted in grey, the OS with the highest number of recovered pointers is HaikuOS 32-bit – which is written mainly in C++. The C++ memory layout is known [20] for introducing overhead due to virtual dispatching, virtual inheritance, and dynamic typing, resulting in a large number of pointers.

We note also that some OSs make extensive use of autopointers (which are almost false positive free due to their definition) while others do not use them at all. A mere glance at the percentage of autopointers in the recovered pointers can tell something about the kernel internals: usually, autopointers are used to represent (doubly) linked lists with zero elements or signal their ends. This is confirmed by Table I: OSs like Linux and Windows which are known to use autopointers for zero-length linked lists show an abundance of them. From the point of view of an OS-agnostic memory forensics approach, the presence of a large number of autopointers is an important piece of information. For instance, it allows the analyst to infer that fields at the same offset in atomic structs which contain autopointers are "roots" of auxiliary linked lists and, in general, it allows to recover information on how the linked lists are implemented in the OS without analyzing the kernel code delegated to their management.

Finally, as expected for the same kernel running on different CPU architectures, we note that the chosen Ω function extracts the same number of pointers for AMD64 and AArch64 Linux with, approximately, the same ratio of autopointers.

TABLE II: Data structures (DSs) recovered.

OS	Linear D.-L. L.	Circular D.-L. L.	Trees	Arrays of *structs	Arrays of *strings	Linked Lists
Darwin	11	385	127	1214	1801	35
Embox	0	22	35	1131	795	6
FreeBSD	86	0	993	1008	895	41
HaikuOS	4117	64	0	305	232	1184
HelenOS	25	1173	127	41	45	1
iOS	20	256	192	5234	229	36
Linux	120	3632	1034	693	5947	46
Linux (Aarch64)	110	3362	936	229	4985	43
NetBSD	41	18	1218	1482	406	45
ReactOS	7	200	49	492	325	12
ToaruOS	101	0	14	62	229	15
vxWorks	51	14	199	349	416	13
Windows XP	38	889	228	463	206	20
Windows 10	145	6639	36	0	282	0

Table II summarizes the number of DSs identified by our tool for each OS. In general, arrays of pointers (to strings or other structs) are the more abundant. In a few cases, our system was not able to find any results for some particular type of DSs. This can be due to a real absence of that DS (for example, because all doubly-linked lists for a given OS are circular and the tool was able to reconstruct them without producing false positive linear ones) or can be a consequence of a more complex implementation that is not currently recognized by our tool (for example, trees in HaikuOS). As for pointers, the number of DSs extracted for AMD64 and AArch64 Linux is approximately the same, confirming the CPU architecture independence of our technique which does not introduce biases.

In Figure 4a we have plotted the cumulative distribution function of the length of circular doubly-linked lists for the three different OSs in which they are more abundant. Dashed lines are power-law fits obtained by using the technique developed by Clauset et al. in [3] and tested against an exponential distribution using the loglikelihood ratio ($P<0.05$). While all the three OSs have circular doubly-linked lists of different magnitude of orders in length, it is possible to note some differences among the distributions. For example, 80% of circular doubly-linked lists in Windows 10 contain less than 11 elements. To reach the same percentage in Linux, lists contain instead up to 200 elements. This fact can suggest to an analyst that the OS under analysis, such as Linux in this case, could make extensive use of this type of data structure also to store small pieces of information. A similar analysis can be made by looking at Figure 4b, which shows the cumulative distribution function of the size of arrays of pointers to strings. In this case, instead, all the OSs prefer to use small arrays of pointers to strings instead of a longer one, which often contains debug strings inserted during the kernel compilation, as in the case of the three OSs considered. The power-law trend of cumulative distributions in cycles and arrays suggests also that is impossible to define simple heuristics to filter out false positives based on a cut-off on the length of cycles/arrays derived from their distribution in the dump. We have also analyzed the distribution of the depth of recovered trees in three OSs (NetBSD, Linux and FreeBSD) chosen with the

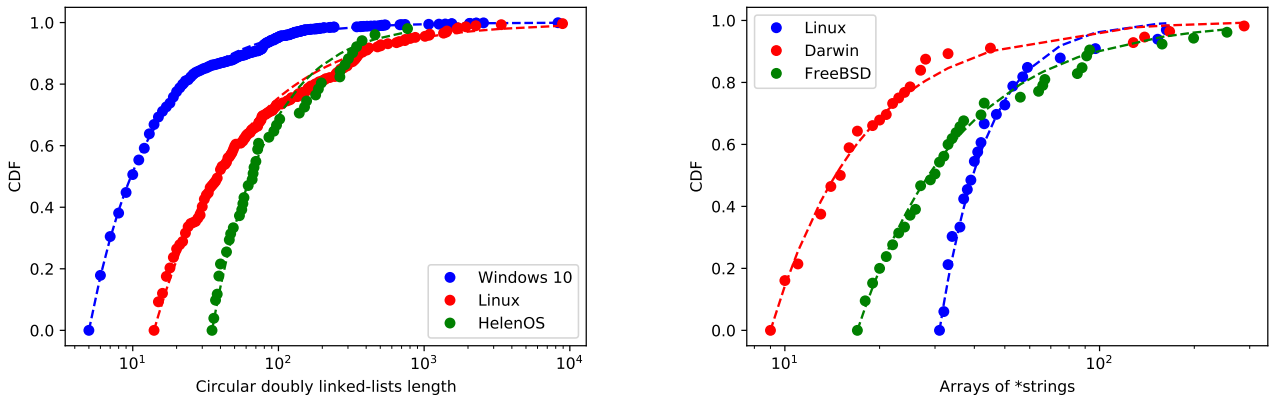

Fig. 4: Circles: cumulative distribution function of the length of circular doubly-linked lists (a) and arrays of pointers to strings (b) for the three OSs in which they are more abundant. Dashed lines: power-law fit of the distributions.

TABLE III: False positives vs seeds.

OS	Process List		
	Structure type[1]	Two seeds all/referenced	One seed all/referenced
Darwin	LDL	6/6	9/9
Embox	AS	5/4	8/5
FreeBSD	L	4/4	34/23
HaikuOS	LDL	3/3	114/114
HelenOS	CDL	2/-	5/-
iOS	LDL	3/3	9/7
Linux	CDL	3/1	76/3
Linux (AArch64)	CDL	1/1	300/2
NetBSD	LDL	2/2	2/2
ReactOS	CDL	5/4	5/4
ToaruOS	A1	1/1	47/47
vxWorks	LDL	2/-	14/-
Windows XP	CDL	3/3	5/3
Windows 10	CDL	4/-	6/-

[1] AS: array of pointers to structs, A1: first level auxiliary structure, CDL: circular doubly-linked list, LDL: linear doubly-linked list, L: linked list

same criteria as above, however, in this case, the distributions do not show the power-law behavior and have a maximum depth for recovered tree equal to 8 (equivalent to a maximum of 256 nodes).

VII. Seed-Based Data Structure Identification

We start our analysis with the most common example of memory forensics: process enumeration. The set of running processes is fundamental to understand the system status at dump time, because it allows the analyst to determine if there were malicious or anomalous processes running on the system, making this the first target, both in terms of priority and relevance, of most memory analyses. To test our technique we simulated the scenario, summarized in Table III, in which the analyst knows either one or two processes' names (the seeds) and wants to retrieve the names of all the other processes running in the system and the data structures that contain or point to them. However, the analyst does not know anything about the OS itself, nor the type of DS the OS uses to maintain processes information. To identify possible seeds we carved the kernel virtual address space for ASCII and UTF-16 strings composed of more than 2 characters.

The second column of Table III reports the DS type of the true process list *as identified by the tool*. It is important to note that, in general, the type identified by the tool can correspond only partially to the exact DS type of the process list: the presence of false positive pointers, due to the use of an approximated Ω function, can affect, for example, the reconstruction of doubly-linked lists which will appear as two separated linked lists as discussed in Section III-B. Most OSs use either single or doubly linked lists to maintain the process list, directly embedding the process name or a pointer to it in the atomic structs. We found two exceptions to this rule. The first is Embox which keeps track of processes through an array of pointers to atomic structs, which in turn contain the process metadata. The second is ToaruOS, which uses a 'light' linked list with atomic structs containing only a pointer to an auxiliary atomic struct that actually contains the process metadata and its name.

Our tool was always able to automatically identify the correct DS and print all the processes' names. However, the tool often identified few candidates structures that pointed to the seeds processes. If the two seeds were uncommon strings, such as `sctp_iterator` for HaikuOS or the well-known `swapper/0` for Linux, the number of DSs identified by our tool as a possible list of processes were always either one or two at the most. If instead, the seeds were more common strings that could appear also in other structures, the tool emitted more possible options. To check this eventuality, in the third column of Table III, we have reported the *worst* case obtained by testing the reconstruction technique with the two process name seeds that are the most referenced by DSs. Each cell of the column contains two values: the first one is the total number of DSs referring to the two seeds, while the second one is the part of them that are referenced by the kernel code as deduced by static analysis. For two OSs (HelenOS and

vxWorks) Ghidra was not able to extract a correct reference to the true process list while for Windows 10 Ghidra was not able to manage the high number of segments that compose the kernel ELF core file and consequently extract references. The table shows that for all the OSs our tool was able to identify the process lists also in the worst-case scenario, outputting at most six possible options for the analyst to check.

In the case in which the analysis knows only one process name and wants to retrieve all the other ones, the fourth column of Table III reports again the *worst* case scenario. For most of the OSs the set of possible structures remained low, with the exceptions of HaikuOS and ToaruOS, which generated respectively 114 and 47 options. In the case of one seed, the references extracted using static analysis can be more useful, significantly reducing the number of options in the cases of FreeBSD and Linux.

Finally, we investigated whether the extracted DSs could also be used to retrieve others non-textual information relative to each process. For example, for all cases that use single/doubly-linked lists it was possible to easily infer the process ID (PID) of each process by querying for integer fields in ascending order (since newer processes are always appended at the end of the list). Also, the same technique can be used to extract timestamps (e.g., related to the starting time of each process).

A. Other Forensics-relevant Data Structures

During a forensics analysis of an unknown OS, there are other DSs that can be useful as a starting point for more advanced analysis of the system. For instance, the set of loaded kernel modules, the set of the kernel memory pools, and information about the mounted filesystems. These DSs can help the analyst to understand the internals of the OS and to extract information that even alone can suggest a compromise of the system. The DS referencing kernel modules, for example, contains information about modules loaded at dump time, making it possible to better understand how the kernel interacts with hardware and identify possible malicious modules loaded by an attacker. The kernel memory pools, such as Windows pool system or SLABs in Linux, are, instead, used by the kernel to allocate memory dedicated to contain other kernel objects divided by their type or size. The identification of pools allows the analyst to deeply explore the memory space of the kernel, understanding how it is managed and what it contains. They are so fundamental that some forensics tools, such as Volatility [23], base their complete analysis of Windows dump on the ability to identify kernel pools. Finally, the kernel manages real or virtual filesystems to organize data on second storage memories or to expose features or statistics, such as the `/proc` or `/sys` virtual filesystems in Linux. Understanding which filesystem is active at dump time can facilitate disk forensics and increase the knowledge about kernel interface (through virtual filesystems) which may have been abused by malicious userspace programs.

For each OS in our dataset, we have collected two kernel module names, two kernel pool names and various strings relative to filesystems (such as filesystem names or the device name associated with hard drives by the kernel) and then used our tool to automatically identify DSs containing them and all the other elements of the same type. Filled circles "●" in Table IV show for which OS our tool was able to successfully extract this information, while those marked with the "†" superscript (e.g., Linux kernel modules) means that the tool found two different DSs that correctly referenced the two seeds. This is very important because sometimes malicious software tampers with kernel data structures to hide their presence but, if we know more data structures are used to link the same atomic structs, it is possible to detect differences among them that might be a sign of infection. For some OSs (Embox, FreeBSD and ToaruOS) it was not possible to detect any DS related to kernel pools because these OSs do not have them. In other cases, our technique fails to detect some of the DSs because kernels use more complex ones which are not detectable by our proof-of-concept tool or due to multiple levels of references to auxiliary structures. An example of the first case is the kernel modules "list" of ToaruOS which is implemented as a hash table, detected by our tool only because, internally, it is implemented using a linked list.

In the fifth column of Table IV we reported other data structures our tool was able to correctly identify by using two known seeds. It is interesting to note that it is possible to obtain information about the interaction between the process and the kernel (communications channels as sockets, pipes, locks and semaphores), information about the hardware (the list of network cards and devices detected), additional information on the kernel itself (kernel parameters, kernel internal tasks) or relative to the userspace programs (associated linked libraries and environment variables). All this information is extremely useful to an analyst which has to face an unknown OS to speed up the forensics process.

These results show the main use of our tool: the analyst extract strings or other known datatypes from memory, then pick a few entries and use our tool to test whether any DS exists that points to the two seeds. If so, the tool also output all the other elements of the DS and prints the corresponding elements that complement the seeds' values.

VIII. Comparison with Other Tools

It would be interesting to measure how many DSs identified by our system are *'correct'*, i.e., they correspond to real structures created by the operating system, and how many DSs we miss. However, this test requires access to ground-truth information regarding the exact data structures present in memory at dump time, also including those unreachable and already de-allocated, since they can still be relevant for forensics investigations. Unfortunately, we are not aware of any tool or technique that can provide this information, even in the case of open-source OSs. Thus, we propose to evaluate the effectiveness of our solution by comparing the forensics-relevant DS recovered by our approach with those extracted by other state-of-the-art tools.

For instance, rule-based systems, such as Volatility, are able to recover forensic-relevant DSs by using custom hand-written rules, specifically tailored for a given version of an operating system. Therefore, for each Volatility Linux plugin, we manually extract the list of DSs used as starting points to extract information from the memory dump. We then check whether Fossil is able to find those same structures in the Linux Debian x86 memory dump used in our experiments.

TABLE IV: Other forensics-relevant data structures.

OS	Kernel modules	Kernel pools	File systems	Other structures
Darwin	●	●	●	• List of network devices • System locks • Kernel/user pipes • Kernel parameters
Embox	●	-[1]	○	• List of commands
FreeBSD	●	-[1]	●	
HaikuOS	●	●	○	• Executable libraries • Kernel/user pipes • Semaphores
HelenOS	●[†]	●	●[†]	
iOS	○	●	●	• List of network devices • System locks • Kernel/user pipes • Kernel parameters
Linux	●[†]	●	●	• Files in `sysfs` • Network protocols
Linux (AArch64)	●[†]	●	●	• Files in `sysfs` • Network protocols
NetBSD	●	●[†]	●	• Kernel tasks
ReactOS	○	●	○	
ToaruOS	●	-[1]	●	• Devices' list • Processes' environment
vxWorks	○	●	○	• Devices' list • Open sockets
Windows XP	●	●	○	
Windows 10	●	●	●	

[1] Not supported by the OS.
[†] Two different types of data structures correctly reference the same seed.

For this comparison, we analyzed 56 Volatility plugins – which correspond to all available commands after removing those that do not involve data structures or that operate on userspace. Out of these, four relied on complex DSs that are not supported by our prototype (i.e., bit-vectors, hashtables and n-ry trees).

Of the final 51, Fossil was able to correctly recover all DSs used by 69% of the Volatility plugins. Starting from those DSs the plugins perform various types of analysis: some of them directly extract the required information, while others access connected auxiliary atomic structs. In all these cases, if the information was in textual form, Fossil reported it by default in its output. Other plugins start instead a graph exploration dereferencing pointers at various offsets, no longer representing coherent, and therefore identifiable DSs, but only paths in the set of Γ graphs. In this case, Fossil correctly identified the DSs used as starting points but was not able to automatically extract the final piece of information. Detailed results are reported in Table VII in the Appendix.

For the remaining 31% of the plugins, the heuristics we use to limit the time complexity of our prototype implementation prevented our tool from recovering the required DSs. For instance, in 10 cases Fossil failed because the DS did not contain a sufficient number of pointers to string (3 plugins) or because it was a list with less than 3 elements (7 plugins).

To our surprise, our OS-agnostic solution was in certain cases able to outperform Volatility. For instance, (highlighted cases in in Table VII in Appendix) Fossil was able to recover the doubly-linked list of the kernel pools required by 7 Volatility plugins, which however Volatility was not able to identify. The reason, as explained in Section I, is that tools that require OS-specific models (i.e., *profiles* in the Volatility jargon) may fail when the kernel is configured with an option not supported by the profile or the plugins system. At a closer inspection, our Debian image used (by default) the `SLUB` kernel pool system, while Volatility's plugins only supported the `SLAB` data structures.

These experiments highlight how tools that use a detailed knowledge of the internals of the OS can extract information precluded to our OS-agnostic approach. But it also shows how small divergence in the shape of atomic structs can completely blind these tools and preclude them from performing *any* analysis. This makes our OS-agnostic approach not only useful for OSs not supported by rule-based systems, but also as a complement to overcoming the limitations of existing solutions.

In a second set of experiments, we compare our approach with Virtuoso, which is an OS-agnostic forensics tool for kernel-space memory proposed so far [5]. Virtuoso collects execution traces of tailor-made test executables running on a virtualized OS and uses those traces to dynamically generate signatures to carve kernel DSs from memory dumps. The authors manually created six test executables to extract forensics-relevant information from the memory dumps of three OSs which are also present in our dataset: Linux, Windows and Haiku. By analyzing the execution traces of these test programs Virtuoso was able to extract: (i) the PID of the process running and the system time at dump time, (ii) the list of all processes including their name and PIDs, and (iii) the list of kernel modules and their names. Excepting the system time and current PID, which are not represented as data structures and therefore are out-of-scope for our technique, Fossil is able to extract the same information from the same OSs.

We remark that our technique has several advantages over Virtuoso. In fact, Virtuoso requires the analyst to run tailor-made executables on the target OS, run the OS in a virtualized environment and trace its entire execution multiple times. Fossil works instead on a single memory dump that can be acquired directly even from hardware devices such as phones, IoT gadgets, printers or network equipment. It also does not require any interaction with the live machine, nor special permissions to execute code or the necessity to virtualize them. Thus, we see Volatility, Virtuoso, and Fossil as complementary solutions for different tasks. Volatility is the best option for those limited targets it supports. Adding a new target is however extremely time-consuming and requires to reverse engineer the internals of the target OS. Virtuoso is the best option to *assist the development* of forensics rules for new OSs, if and only if the analyst has complete control over it and can compile and deploy new applications and trace the entire OS execution. Fossil provides instead a solution to study raw memory acquired from any class of devices and/or OSs, without the need to develop custom rules.

The median execution time to extract all DSs among the 14 OSs we analyzed in our experiments is 17 minutes (the max is 118 minutes for HaikuOS, the OS with the largest number of pointers) on an Intel Xeon 32-core CPU equipped with 128GB of RAM. Table VI in the Appendix reports all individual execution times. Moreover, it is important to note

that this operation only needs to be performed once and does not need to be repeated each time the analyst wants to explore recovered DSs. The DS analysis then takes only a few seconds.

Finally, we do not provide an evaluation of the accuracy of the Ω function, as the function is an input to our approach and not a contribution to this paper. In particular, as discussed in Section V, for our tests we use the function provided by Oliveri and Balzarotti [15] that, according to the authors, has no false negatives, i.e., it finds *all* the existing pointers in kernel memory. Our technique is then built to tolerate pointer false positives, which introduce an additional overhead but no errors in our results.

IX. Seed-Less Data Structure Identification

We now investigate a second, more challenging scenario in which the analyst is unable to identify any seed information. In this case, the analysis is completely blind, but our tool can still automatically identify and reconstruct DSs from the raw dump. However, without seeds, the analyst has to manually check the extracted information to locate interesting data about the running system. To assist in this process, in a seed-less scenario our system provides a set of OS-agnostic heuristics to filter and prioritize the number of DSs that an analyst may need to investigate.

First of all, we continue to hierarchically organize DSs as discussed in Section V-D, which allows the analyst to start the exploration from structures that are less likely to be false positives. We also prioritize DSs that embed (or contain references to) printable strings, as it is simpler for human analysts to recognize information in this format and these structures can serve as entry points to explore other atomic structs linked to them. Furthermore, for each string or reference to a string, we require that the number of unique strings is greater than 50% of the number of referenced/pointed strings: this process removes fields that contain too many repeated strings, which are unlikely to be true positives. Then, we sort the remaining data structures by the mean abundance of different strings which they embed/point, with the assumption that very frequent sequences of characters are more prone to be false positives. Finally, when the system is able to extract global pointers from the static analysis phase, it also provides this information to the analyst.

In seed-less mode, the tool provides as output a ranked list of structures and, for each structure, it shows the address in memory of its elements and a set of strings that are either contained or referenced by pointers in the element. Table V shows examples of the forensics-relevant DSs that we are able to identify in the different OSs in a seed-less configuration. Each cell of the table reports the position in the ordered list of DSs provided by our tool. For instance, the process list was the 2nd in the output of Darwin and the 5th for ReactOS. The "○" symbol means the tool was unable to retrieve the information in the seed-less scenario, but it succeeded when seeds were available (as shown in Table III), while "-" indicates that we were not able to retrieve it at all. We have been able to identify all the process lists, the majority of kernel modules and pools lists and some filesystem-relevant data structures.

It is interesting to note that over 50% of the recovered structures appear in the top5 positions in the output of our system, and by checking the top20 candidates an analyst would be able to discover over 81% of them. This, combined with the fact that it only takes a few seconds to verify one of the output DSs, makes our OS-agnostic approach a viable solution to investigate unknown memory dumps. In fact, with the help of our tool, an analyst can quickly identify several classes of relevant information in a completely automated fashion. Of course, more time is then needed to manually inspect the memory of those structures to identify auxiliary (non-string) information, e.g., to discover which other structures are connected to the process list, which metadata are recorded for each process, to dump the memory of each process, etc.

TABLE V: Ranking position of various information in the seed-less analysis results.

OS	Process list	Kernel modules	Kernel pools	Filesystem
Darwin	2	10	11	7
Embox	17	○	-	-
FreeBSD	24	31	-	26
HaikuOS	6	1	11	-
HelenOS	4	2	1	1
iOS	2	-	2	15
Linux	5	28	26	15
Linux (AArch64)	4	22	19	24
NetBSD	2	6	18	○
ReactOS	5	-	12	-
ToaruOS	3	2	3	-
vxWorks	4	-	2	-
Windows XP	5	1	2	-
Windows 10	41	○	○	○

It is important to note that in the case of seed-less analysis any comparison of Fossil with other tools results impossible because, as far as we know, it does not exist any forensics tool able to work in so extreme conditions: any information on the OS, impossibility to instrument/virtualize it and no multiple dumps availability.

X. Related Work

Other than Virtuoso [5] already discussed in Section VIII, OS-agnostic data structure recovery, as a particular application of the more general problem of data reconstruction, was explored by various authors by using different techniques.

An approach based on multiple snapshots of the memory of the same process in order to collect information about the DSs used by the application is explored by Urbina et al. [22]. SigGraph [10] reconstructs instead the graph of data structures inside a memory dump of a generic OS, by using signatures derived from its source code. DIMSUM [9] uses probabilistic inference to identify instances of a specific DS by starting from its definition. Lin et al. have also developed REWARDS [11], which instruments user-space processes only using Intel PIN, captures the timestamps of each memory access and reveals their DSs in system-wide memory dumps. Researchers have also investigated how to create signatures from binary executables: ORIGEN [6] reconstructs offsets of atomic data structures by using static analysis on the code of a previous version of the same kernel, while Case et al [1]

reconstruct the offsets of important Linux structures by directly analyzing code embedded in the dump itself. Finally, Song et al in [19] have developed a tool able to generate abstract representations for kernel objects by using a graph-based deep learning approach, which requires, however, multiple dumps of the same OS in order to be trained, undermining its extensibility to unknown OSs. All previous approaches always assume that the analyst knows something about the internals of the OS (access to the source code, definition of data structure to recover, multiple dumps of the same system in different conditions etc.), use approximations tailored for the OS for which they are designed or require to run the OS inside a hypervisor affecting their applicability in an OS-agnostic forensic analysis on real devices.

The work most closely related to our study is a technique recently described in a parallel paper from Tran-Quoc et al. [21]. The paper focuses mostly on page table reconstruction for Intel X86, but the authors also describe a use case in which they use the recovered pointers to identify the list of running processes in Linux, BSD, and MS Windows given two initial seeds. While the idea to identify and follow pointers is similar, that work does not discuss data structure reconstruction, cannot deal with trees, arrays, or cases in which the process name is not part of the linked list structure itself, and do not consider seed-less introspection.

XI. Conclusions

In this paper, we discussed the problem of extracting data structures from memory dumps without any knowledge of the OS that has generated them, by extracting data structures using only their topological properties. In particular, we have posed the attention to how the presence of topological constraints may facilitate the identification of certain types of structures and, at the same time, allow to order them on the base of the reliability in their reconstruction. We have discussed how to limit the problem of false positives due to spurious chains in the Γ graphs. Furthermore, we have implemented heuristics to estimate the atomic struct sizes, a fundamental step to identify data embedded/pointed by a certain data structure.

We have tested our technique in both a seed-based and a seed-less scenarios, on 14 different OSs. In the first case, we have shown how, even with only one seed, it is possible to extract the process list, the kernel module list, the set of pools, and the information about filesystems used by each OS. In the more challenging seed-less scenarios, an analyst was still able to extract 81% of the main data structure considered in the seed-based approach by exploring only the top 20 data structures returned by our tool.

Acknowledgment

This project was supported by the European Research Council (ERC) under the European Unions Horizon 2020 research and innovation programme under grant agreement N° 771844 (BitCrumbs).

References

[1] A. Case, L. Marziale, and G. G. Richard, "Dynamic recreation of kernel data structures for live forensics," *Digital Investigation*, vol. 7, pp. S32–S40, 2010, the Proceedings of the Tenth Annual DFRWS Conference. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1742287610000320

[2] Z. Cekerevac, Z. Dvorak, and T. Pecnik, "Top seven iot operating systemsin mid-2020," *MEST Journal*, vol. 8, 07 2020.

[3] A. Clauset, C. R. Shalizi, and M. E. Newman, "Power-law distributions in empirical data," *SIAM review*, vol. 51, no. 4, pp. 661–703, 2009.

[4] M. Cohen, "Rekall memory forensic framework," 2014.

[5] B. Dolan-Gavitt, A. Srivastava, P. Traynor, and J. Giffin, "Robust signatures for kernel data structures," in *Proceedings of the 16th ACM Conference on Computer and Communications Security*, ser. CCS '09. New York, NY, USA: Association for Computing Machinery, 2009, p. 566–577. [Online]. Available: https://doi.org/10.1145/1653662.1653730

[6] Q. Feng, A. Prakash, M. Wang, C. Carmony, and H. Yin, "Origen: Automatic extraction of offset-revealing instructions for cross-version memory analysis," in *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, ser. ASIA CCS '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 11–22. [Online]. Available: https://doi.org/10.1145/2897845.2897850

[7] R. M. Karp and M. O. Rabin, "Efficient randomized pattern-matching algorithms," *IBM J. Res. Dev.*, vol. 31, no. 2, pp. 249–260, 1987. [Online]. Available: https://doi.org/10.1147/rd.312.0249

[8] D. E. Knuth, *The art of computer programming, Volume II: Seminumerical Algorithms, 3rd Edition*. Addison-Wesley, 1998. [Online]. Available: https://www.worldcat.org/oclc/312898417

[9] Z. Lin, J. Rhee, C. Wu, X. Zhang, and D. Xu, "Dimsum: Discovering semantic data of interest from un-mappable memory with confidence," in *Proc. NDSS*, 2012.

[10] 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 *Ndss*, 2011.

[11] Z. Lin, X. Zhang, and D. Xu, "Automatic reverse engineering of data structures from binary execution," in *Proceedings of the 11th Annual Information Security Symposium*, 2010, pp. 1–1.

[12] R. Love, *Linux Kernel Development*, 3rd ed. Addison-Wesley Professional, 2010.

[13] National Security Agency, "Ghidra - software reverse engineering framework," 2022.

[14] A. Oliveri. (2022) Fossil. EURECOM. [Online]. Available: https://github.com/eurecom-s3/fossil

[15] A. Oliveri and D. Balzarotti, "In the land of mmus: Multiarchitecture os-agnostic virtual memory forensics," *ACM Trans. Priv. Secur.*, mar 2022, just Accepted. [Online]. Available: https://doi.org/10.1145/3528102

[16] F. Pagani and D. Balzarotti, "Autoprofile: Towards automated profile generation for memory analysis," *ACM Transactions on Privacy and Security*, vol. 25, no. 1, pp. 1–26, 2021.

[17] F. Pagani, O. Fedorov, and D. Balzarotti, "Introducing the temporal dimension to memory forensics," *ACM Transactions on Privacy and Security (TOPS)*, vol. 22, no. 2, pp. 1–21, 2019.

[18] Z. Qi, Y. Qu, and H. Yin, "LogicMEM: Automatic profile generation for binary-only memory forensics via logic inference," in *Proceedings 2022 Network and Distributed System Security Symposium*. Internet Society, 2022. [Online]. Available: https://doi.org/10.14722/ndss.2022.24324

[19] W. Song, H. Yin, C. Liu, and D. Song, "Deepmem: Learning graph neural network models for fast and robust memory forensic analysis," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 606–618. [Online]. Available: https://doi.org/10.1145/3243734.3243813

[20] P. F. Sweeney and M. Burke, "Quantifying and evaluating the space overhead for alternative c++ memory layouts," *Software: Practice and Experience*, vol. 33, no. 7, pp. 595–636, 2003.

[21] T.-A. Tran-Quoc, C. Huynh-Minh, and A.-D. Tran, "Towards os-independent memory images analyzing: Using paging structures in memory forensics," in *2021 14th International Conference on Security of Information and Networks (SIN)*, vol. 1, 2021, pp. 1–8.

[22] D. Urbina, Y. Gu, J. Caballero, and Z. Lin, "Sigpath: A memory graph based approach for program data introspection and modification," in *European Symposium on Research in Computer Security*. Springer, 2014, pp. 237–256.

[23] A. Walker, "Volatility framework: Volatile memory artifact extraction utility framework," 2017.

APPENDIX

A. Fast Discovery of Doubly-Linked Lists

As discussed in Section IV-A2, to discover doubly-linked lists we look for "parallel" chains running in inverse directions: e.g., if we have a chain connecting structs A, B, C, D, E, we are looking for another chain connecting the same structs in the opposite direction: E, D, C, B, A. We do not have a way to distinguish *next* from *previous* pointer, but for our purposes the two chains are interchangeable—i.e., we care about finding the skeleton of the doubly-linked list, not distinguishing the two directions. Doubly-linked lists are fundamental in many kernels, so we tackled the task of recognizing them with attention, focusing particularly on minimizing the number of false negative cases.

The first issue that complicates our task is that we will not only want to recognize chains whose structural pointers *previous* and *next* have the same destination, but also those that have a fixed offset between them, suggesting that they point to different offsets in the same struct.

A further problem we want to address is that, for a given chain, it is possible that the first x elements do not belong to the doubly-linked list: they may be non-structural pointers from other atomic/data structures that ultimately lead to the doubly-linked lists, we want to recognize while following pointers at the same offset by coincidence as shown in Figure 3. Hence, for a given chain we also want to test all its suffixes, i.e., all chains of size 3 or more obtained discarding any number of its first elements.

The two issues described above make it challenging to design a fast algorithm to recognize doubly-linked lists. We solved it by focusing on an invariant that still applies in our case: if we take the *difference* between consecutive pointer locations in two chains composing a valid doubly-linked list, they do not change depending on the offset they reach in an atomic struct. In other words, if `&x` denote the address in memory of a given atomic struct, taking the difference between consecutive pointers in an $A \to B \to C$ chain will return the [(`&B-&A`), (`&C-&B`)] values, irrespectively of the offset in the struct at where the pointers are located.

Crucially, if two chains are indeed the two "halves" of a doubly-linked list, taking the same operation of computing the difference of consecutive pointers on the two halves would return two lists that have a clear relationship: by inverting one list and changing the sign of all values, we obtain the other. In the $A \to B \to C$ example, for the corresponding $C \to B \to A$ we would indeed obtain the [(`&B-&C`), (`&A-&B`)], which corresponds to the result of inverting and changing all signs of the [(`&B-&A`), (`&C-&B`)] list mentioned before.

Our algorithm is hence based on the above realization: for each chain c containing pointers $p_0, p_1, \ldots, p_n$, we will compute two hashes:

$$h_{c,0} = h([(p_1 - p_0), (p_2 - p_1), \ldots (p_n - p_{n-1})]),$$

which is the hash of sequence of the difference between pointers in it, and

$$h_{c,1} = h([(p_{n-1} - p_n), (p_{n-2} - p_{n-1}), \ldots (p_0 - p_1)]),$$

the hash of the same sequence after inverting it and changing its sign. Since we are interested in the suffixes of c, we will apply the same procedure to all the suffixes of c, i.e., the chains containing the pointers $p_1, \ldots, p_n$, $p_2, \ldots, p_n$ etc. Each pair of chains c_0 and c_1 where $h_{c_0,0} = h_{c_1,1}$ and $h_{c_1,0} = h_{c_0,1}$ is a candidate doubly-linked list.

Using standard hashing functions to compute these hashes is computationally expensive because computing all the hashes for the suffixes of a chain of length n turns out to have $O(n^2)$ computational complexity. In Appendix B we discuss our design of a specialized strategy to do that with optimal $O(n)$ computational complexity.

Once the hashes are computed, we now process candidate pairs. We group chains by matching couples of hashes and, out of caution, we verify that we did not have hash collisions (i.e., we verify that all chains have the same length and are "parallel": corresponding structs have a constant distance between them).

We want to assign each chain and each pointer to at most one doubly-linked list: any candidate chain that will be processed having a pointer already assigned to another doubly-linked list is immediately discarded. The longest matching chains are those that have both the lowest probability of being false positives and the most informative.

Once we have a candidate chain $c = [p_0, p_1, \ldots p_{n-1}]$, we consider the smallest distance between its elements ordered by their addresses. These pointers should be at the same offset of different atomic structs, so the smallest value of the differences $t = \min_{i \in 1 \ldots n}(c'_i - c'_{i-1})$ is a higher bound for the size of the atomic structs pointed by c. Among the chains that are candidate matches to c, we take the one at the smallest offset o from it. If $o + \alpha \leq t$, then we consider the two chains to be a successful match and output them as a doubly-linked list.

B. Bi-Directional Hashes

In the first iteration of our implementation, we found that the part of the hash computation was a severe bottleneck because for each list of length n we had to compute 2 hashes for each of its suffixes, for a $O(n^2)$ computation complexity for each chain of length n that turned out to be untreatable because we often encountered long chains, leading to expensive computations.

We can state our problem as that of computing, for a list $l = [v_0, \ldots, v_{n-1}]$ having n elements, all the values such $h_{0,i}$ and $h_{1,i}$ such that

$$\begin{cases} h_{0,i} = h([v_i, v_{i+1}, \ldots, v_{n-1}]) \\ h_{1,i} = h([-v_{n-1}, -v_{n-2}, \ldots, -v_i] \end{cases}$$

for all values of $i \in \{0, 1, \ldots, n-1\}$, where h is a suitable (non-cryptographic) hashing function.

We define our hash function to be

$$h([x_0, \ldots, x_{n-1}]) = \sum_{i=0}^{n-1} a^i x_i \mod 2^{64}, \qquad (2)$$

where a is a constant.[5] This function is inspired by rolling hash functions such as those used in the Rabin-Karp algo-

[5] In our implementation, $a = 6364136223846793005$ [8].

rithm [7]. The modulo 2^{64} arithmetic is implemented simply and efficiently by using unsigned 64-bit integers (and ignoring overflows).

The $h_{0,i}$ values are computed simply by noting that

$$\begin{cases} h_{0,n-1} = v_{n-1} & \mod 2^{64} \\ h_{0,i} = v_i + a \cdot h_{0,i+1} & \mod 2^{64} \quad \forall i \in \{n-2, \ldots, 0\}. \end{cases} \tag{3}$$

Hence, we can compute all these values simply by starting with the known value of $h_{0,n-1}$ and looping back to $h_{0,0}$ using Eq. (3).

For the $h_{1,i}$ values, we need to find the multiplicative inverse of a, that is $a^{-1} \mod 2^{64}$. We can do that using the extended Euclidean algorithm [8].[6] Similarly to Eq. (3), we compute auxiliary values t_i such that

$$\begin{cases} t_{n-1} = -v_{n-1} & \mod 2^{64} \\ t_i = -v_i + a^{-1} \cdot t_{i+1} & \mod 2^{64} \quad \forall i \in \{n-2, \ldots, 0\}. \end{cases}$$

We finally compute

$$h_{1,i} = a^{n-1-i} t_i \mod 2^{64} \quad \forall i \in \{0, \ldots, n-1\}.$$

It is not hard to verify that values computed in this way satisfy Eq. (2).

In a further optimization, we recognize that chains can be "confluent", i.e., they have a common suffix. When this happens, we do not recompute the hashes already computed for suffixes already examined, and we reuse the values $h_{0,j}$ and t_j already computed for the common suffix of the chains.

C. Performance Evaluations

TABLE VI: Time required for the extraction of data structures

OS	Minutes	OS	Minutes
Darwin	12	Linux (AArch64)	64
Embox	3	NetBSD	110
FreeBSD	26	ReactOS	11
HaikuOS	118	ToaruOS	3
HelenOS	5	vxWorks	4
iOS	22	Windows XP	12
Linux	51	Windows 10	24

TABLE VII: Fossil's ability to detect data structures used by Volatility plugins

Plugin	Seekable by Fossil	Found by Fossil	Plugin	Seekable by Fossil	Found by Fossil
apihooks	●	●	list_raw[8]	●	○
arp[1]	●	○	lsmod	●	●
aslr_shift[2]	○	○	lsof[7]	○	○
banner[2]	○	○	malfind	●	●
bash[3]	○	○	memmap	●	●
bash_env[3]	○	○	moddump	●	●
bash_hash[3]	○	○	mount	●	●
check_afinfo[1]	●	○	mount_cache[1,6]	●	◐
check_creds	●	●	netfilter[8]	●	○
check_evt_arm[4]	○	○	netscan[2]	○	○
check_fop[6]	●	●	netstat	●	●
check_idt[2]	○	○	pidhashtable[7]	○	○
check_inline_kernel[2]	○	○	pkt_queues	●	●
check_modules[1]	●	○	plthook	●	●
check_syscall	●	●	proc_maps	●	○
check_syscall_arm[4]	○	○	proc_maps_rb	●	●
check_tty	●	●	procdump	●	●
cpuinfo[2]	○	○	process_hollow[3]	○	○
dentry_cache[6]	●	●	psaux	●	●
dmesg[5]	○	○	psenv	●	●
dump_map	●	●	pslist	●	●
dynamic_env[3]	○	○	pslist_cache[6]	●	●
elfs	●	●	psscan[2]	○	○
enumerate_files[6]	●	○	pstree	●	●
find_file[7]	○	○	psxview[6]	●	◐
getcwd	●	●	recover_filesystem[6]	●	○
hidden_modules	●	●	route_cache[6]	●	○
ifconfig[8]	●	○	sk_buff_cache[6]	●	●
info_regs	●	●	slabinfo	●	●
iomem[7]	○	○	strings[8]	●	○
kernel_opened_files[8]	●	○	threads	●	●
keyboard_notifiers[8]	●	○	tmpfs	●	●
ldrmodules	●	●	truecrypt[8]	●	○
library_list	●	●	vma_cache[6]	●	●
librarydump	●	●	yarascan[2]	●	●

- Highlighted rows: Fossil can identify data structures that should be used by the plugin but which are missed by Volatility, see Section VIII for more details.
- ◐: Fossil identifies only part of the needed data structures.

[1] Less than 90% of pointed/embedded strings in the data structure.
[2] Plugin looks for unstructured data.
[3] Data structure in user space.
[4] Not applicable on x86.
[5] Plugin not working/broken on the analyzed kernel version.
[6] Based on another plugin that uses data structures (not) identified by Fossil.
[7] Use a more complex data structure/data representation not supported by Fossil.
[8] Data structure too short to be identified/not present in the dump.

[6] In our implementation, $a^{-1} = 13877824140714322085$.