Back to the Whiteboard: a Principled Approach for the Assessment and Design of Memory Forensic Techniques

Fabio Pagani
EURECOM

Davide Balzarotti
EURECOM

Abstract

Today memory analysis plays a fundamental role in computer forensics and is a very active area of research. However, the field is still largely driven by custom rules and heuristics handpicked by human experts. These rules describe how to overcome the semantic gap to associate high level structures to individual bytes contained in a physical memory dump. Structures are then traversed by following pointers to other objects, and the process is repeated until the required information is located and extracted from the memory image.

A fundamental problem with this approach is that we have no way to *measure* these heuristics to know precisely how well they work, under which circumstances, how prone they are to evasions or to errors, and how stable they are over different versions of the OS kernel. In addition, without a method to measure the quality and effectiveness of a given heuristic, it is impossible to compare one approach against the others. If a tool adopts a certain heuristic to list the sockets associated to a program, how do we know if that is the only possible way to extract this information? Maybe other, even better, solutions exist, just waiting to be "discovered" by human analysts.

For this reason, we believe we need to go back to the drawing board and rethink memory forensics from its foundations. In this paper we propose a framework and a set of metrics we can use as a basis to assess existing methodologies, understand their characteristics and limitations, and propose new techniques in a principled way. The memory of a modern operating system is a very large and very complex network of interconnected objects. Because of this, we argue that automated algorithms, rather than human intuition, should play a fundamental role in evaluating and designing future memory forensics techniques.

1 Introduction

Computer forensics is often considered an art, as the analyst proceeds by formulating hypothesis about the cause of an incident and uses inductive reasoning to reinforce or discard them based on clues and artifacts collected from the target system. The way these artifacts are extracted and analyzed is largely based on the experience and on the set of heuristics encoded into the available tools. Memory forensics, i.e. the field focusing on the analysis of snapshots of the physical memory of a machine, is no exception to this rule. Memory forensic tools needs to recover the high-level semantic associated with sequences of raw bytes – thus reconstructing the internal state of the operating system (OS) and its applications at the time the memory was acquired. Unfortunately, modern OSs are very complex software whose memory often contains millions or tens of millions of individual objects at any moment in time. Even worse, both the fields and the layout of these objects can change when the kernel is updated or recompiled, and the connections among them evolves very rapidly – with a considerable amount of links and pointers that change every few milliseconds.

Currently memory forensics techniques rely on a large number of rules and heuristics that describe how to navigate through this giant graph of kernel data structures to locate and extract information relevant to an investigation. For example an analyst can use rules — commonly known as *plugins* in the field terminology — to retrieve the list of processes running at acquisition time (including their name, starting time, process ID, and other related information) or the list of open sockets. The result of the analysis depends on the number and accuracy of these rules. However, the field today is still in its infancy and each individual technique is manually written by researchers and practitioners. As a result, it is often unclear why a particular exploration strategy has been chosen, except for the fact that some developers found it reasonable based on their experience. Even worse, how accurate a given heuristic is and how we can compare it with other candidates to decide which strategy is more suitable for a given investigation remains an open question.

In fact, we still do not even know how to properly characterize the accuracy of a technique, as its quality depends on the metric we use to evaluate it, which in turn depends on the goal of the analyst. For instance, in an adversarial environment in which the analyst is investigating a sophisticated attack, a good heuristic would be one that is difficult to evade for an attacker. In a different investigation in which there is no risk of tampered

kernel data, a heuristic that only traverses closely-related (in physical memory) data structures may be preferable as pages acquired far apart may otherwise contain inconsistent information if the dump was not acquired atomically.

Contribution: The goal of this paper is to introduce a more principled way to approach the problem of memory analysis and forensics. Our plan is articulated around three main points. The first intuition is that heuristics used to extract information from memory dumps should be automatically generated by computers and not handpicked by humans. As we will show in our experiments, the graph of kernel objects is tightly connected and there are tens of millions of different ways to reach a given structure by starting from a global symbol. The second point is the fact that it is very important to be able to quantitatively measure the properties of each heuristic, so that different options can be compared against one another and an analyst can decide which technique is more appropriate for her investigation. Finally, the analyst should be able to obtain some form of guarantee about the results, to ensure that once a given quality metric has been chosen, a certain technique is the *optimal* solution to navigate the intricacies of runtime OS data structures.

As a step towards these goals, we constructed a complete graph of the internal data structures used at runtime by the Linux kernel. In our graph, nodes represent kernel objects and edges a pointer from one object to another. We chose Linux as the availability of its source code simplifies the creation of our model. However, a similar graph was also extracted in the past by Microsoft for the Windows kernel [4] and could therefore be reused for our purpose. The resulting map of the memory is a giant network (containing over a million nodes) with a very dynamic topology that is constantly reshaped as new data structures get allocated and deallocated.

Memory forensics tools adopts rules to navigate through the data structures present in a memory dump, and these rules can therefore be represented as paths in our kernel graph. Nodes and edges can then be decorated with additional pieces of information that capture different properties an analyst can find important in an analysis routine. In our study we model this phase by introducing and discussing five different metrics: *Atomicity*, *Stability*, *Generality*, *Reliability*, and *Consistency*. We used these metrics to compute a score associated to each path, and therefore to existing memory analysis techniques, as well as to compute the optimal solution according to a chosen set of criteria. We then discuss the intricacies of identifying such optimal paths by performing experiments with 85 different kernel versions and 25 individual memory snapshots acquired at regular time intervals.

Building a map of the kernel memory is a very tedious and time-consuming process. However, we believe this map can have many interesting uses in computer security beyond memory forensics – including virtual machine introspection (VMI), kernel hardening, and rootkit detection. Since both the code and the results of the previous attempts to build this graph [4,17,19] are not publicly available, we decided to release all our data – hoping it will help other researchers to considerably reduce the time required to investigate and validate techniques that require information about the content of a running kernel.

2 Motivation

Being quite in its infancy, memory forensics has still many open problems, which have been recently summarized by Case and Richard [6]. The authors divided them in two categories, depending on whether they are related to the *acquisition* or the *analysis* of a memory dump. More precisely, the first category contains all the practical issues of acquiring memory from a device under investigation while the second one deals with the capabilities of memory forensics, such as malware detection and evidence extraction.

One of the main issues belonging the first category is *page smearing*, which is a consequence of the fact that while the acquisition is performed the underlying system is not frozen and thus the dump may contain inconsistent information [12]. While the term was coined in 2004 [5], its actual implications are still unclear to the community. For instance, a recent study from Le Berre [18] pointed out that in real investigations more than the 10% of memory dumps suffer from this problem and thus can not be properly analyzed with existing tools. Our work can help mitigating this issue by assessing how existing techniques are affected by non-atomic acquisitions, and help design new heuristics which are more robust against the presence of inconsistent information.

The second category focuses instead on challenges related to memory analysis. For example, as of today, forensics practitioners lack the necessary tooling for extracting a number of interesting information such as Powershell activity and evidences related to Office applications and private browsing sessions, or to analyze sophisticated userland malware. Finally, a vast range of technologies did not receive any forensics coverage: Apple iOS, Chromebooks, and IoT devices are still out of scope when it comes to memory forensics analysis.

While these issues are very different from one another, most of them share the same underlying assumption: kernel objects must be located, traversed and interpreted by a set of rules. Our approach enables forensics practitioners and researchers to evaluate, under different constraints, the quality of these rules and provide them with a framework to compare and discover new sets of rules.

3 Approach

In this section we describe the four-step approach we propose to precisely measure and improve the quality of existing memory forensics techniques. The first step consists of building a precise representation of all data structures that exists in a running kernel and of the way these structures are connected

① `[init_task].tasks.prev` ↻ → **`task_struct`**`.mm` → `mm_struct.mmap` → `vm_area_struct.vm_next` ↻ → `vm_area_struct`

② `[root_cpuacct].css.cgroup` → `cgroup_root.cgrp.e_csets[2].next` → `css_set.tasks.next` → **`task_struct`**`.mm` → `mm_struct.mmap` → `vm_area_struct.vm_next` ↻ → `vm_area_struct`

Figure 1: Two different paths that reach the same `vm_area_struct` object.

to one another. The challenges and the process we followed to build this kernel graph are described in details in Section 4. In the second phase we map existing forensic analysis techniques into our model, by representing their algorithms as paths through the kernel graph. We then color the graph according to different properties that are relevant for a forensic investigation, and we employ graph-based algorithms to assess the characteristics of the previously-identified paths and find new ones that may exhibit better properties. Finally, in the fourth and final phase of our methodology we translate our findings back to the memory forensic space by generating improved analysis plugins, thus increasing the number and quality of the rules that are used today to analyze memory dumps.

3.1 Memory Forensics as a Graph Exploration Problem

The goal of memory forensics is to bridge the semantic gap between the raw bytes that constitute a physical memory's snapshot and the high-level abstractions provided by modern operating systems. This task requires the forensic tool to be able to correctly translate virtual to physical memory addresses, as well as to identify the data structures that contain the required information (e.g., the name of the files opened by a given process). The latter is typically achieved in two phases. First, the system locates a known object – either because it resides at a fixed or predictable location, by using symbols information generated by the compiler when the kernel was built, or by carving a particular data structure based on a set of known properties and invariants. Starting from this entry point, the analysis then traverses different memory regions, moving from one data structure to the next by following pointers, until it reaches the required piece of information. For example, we assume the analyst found a suspicious process and she wants to extract its executable code for further analysis. On Linux, this analysis starts from extracting the position of the global kernel variable `init_task` of type `task_struct`. This is one of the most important kernel object in terms of Linux memory forensics since every kernel thread and user space process has its own and it serves as a hub to reach several other relevant pieces of information. After locating `init_task`, the processes list is walked until the `task_struct` belonging to the suspicious process is found. From here, the `mm_struct` is reached by dereferencing the `mm` field. Finally, the list of `vm_area_struct`, each of which defines a virtual memory area, is retrieved — first by following the `mmap` pointer, then by using the `vm_next` field. With this information, the analyst can find the executable regions of the process and can proceed to save their content to disk.

This procedure can be naturally represented as a path on a graph in which every node is a kernel object, and every link a pointer. While the final node is dictated by a given forensic task, both the first and the intermediate nodes are often the result of handcrafted routines based on the experience and expert judgment of the developers of the forensic tool.

In our graph, the previously presented analysis would correspond to the path ① in Figure 1. The path contains the names of the structures and fields that need to be traversed (in square brackets when they refer to global symbols in the kernel) as well as the type of transition (→: follow a pointer reference, ↻: visit multiple structures of the same type linked together). For simplicity, we report inner structures in our paths as names in the edge and not explicitly as standalone nodes. Also, note that in the example ①, since the suspicious process was freshly spawned, the shortest path in our graph traverses the process list *backwards* — contrarily to the more common *forward* walking.

On top of the previous solution, our approach shows that a stunning 2.5 million different sequences of vertices exist in the kernel graph to reach the very same target object starting from a global variable, *only* counting the paths with no more than 10 edges. For example, path ② in Figure 1 begins from the little-known global symbol `root_cpuacct`, passes through a number of *cgroup*-related objects, before finding the `task_struct` of the suspicious process.

The previous two "rules" are both capable of locating a given process structure in a memory dump. The first is certainly more intuitive and it may also traverse a lower number of data structures. However, this is purely a *qualitative* assessment, and it is unclear if the first solution actually has any clear advantage or whether it provides any better guarantee then the second.

3.2 Path Comparison

As we saw in the previous example, if we want to assess the quality of a given solution, we first need to define what "quality" means in our context. In other words, when two paths exist to reach the same target data structure, we need to define a metric that can tell us which one is better to follow from a forensic perspective.

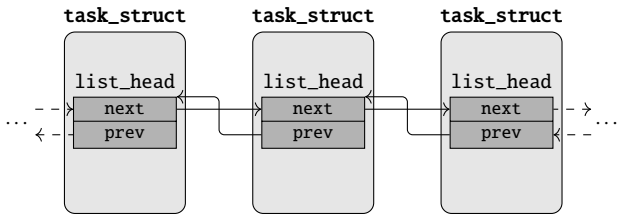


Figure 2: `task_struct`s organized in a doubly linked list.

A developer may favor the shortest path, as it is simpler to implement and may appear to be more robust according to the intuition that the fewer the data structures that need to be parsed, the less likely it is that something can go wrong while doing that. However, this approach raises another important issue about today's approach for memory analysis: its ad hoc nature and lack of a scientific foundation. In fact, it is not clear today how different exploration techniques can be compared and how they can be evaluated against one another in a precise and measurable way.

A first important observation is that there is not a single, absolute metric that defines the quality of a memory exploration rule. It all depends on the goal of the analyst, the conditions under which the memory snapshot was acquired, and the type of threat that is investigated. For example, in the common case in which a memory snapshot is acquired non-atomically, the analyst may prefer to adopt an approach that only traverses structure closely located in memory, thus minimizing the chances of inconsistencies. On the opposite case in which the memory was acquired atomically in a lab from a virtual machine used to investigate a possible rootkit, the analyst would certainly favor a different approach that traverses structures whose values cannot be tampered with by the attacker. In yet another scenario, an investigator may try to analyze a dump for which she was not able to retrieve a correct OS profile, and therefore she might be interested in paths that traverse structures that have changed very rarely across different kernels, to maximize her probability of success.

Therefore, it is the analyst who needs to select the more appropriate fitness function to compare paths according to any combination of desired properties. And once this function has been chosen, it is possible to use it to compute the optimal path (and therefore the optimal exploration strategy) to traverse the kernel graph. In this paper we explore different possible scenarios by proposing several metrics to enrich the graph (more details about this process are presented in Section 5) and then use this information to evaluate existing approaches and discuss other, non-conventional solutions that can provide better guarantees for the analyst.

4 Graph Creation

The first step of our methodology consists in building a model of the operating system kernel, that we can later use to compare different memory forensic approaches. The model we chose for our analysis is a graph of kernel objects, in which nodes represent kernel data structures and edges represent relationships between objects (for example a pointer from one structure to another).

The core idea is simple and relies on two crucial pieces of information extracted from the kernel debugging symbols. The first one is the layout, in terms of the exact type and offset of each field, of all the `struct` defined and used by the kernel code. The second information is instead related to the address, name, and type of global kernel variables that play the role of entry points for our graph exploration. Starting from these global pointers, our algorithm can recursively traverse other structures, each time following a pointer and casting the target memory to the appropriate type. While this process may seem straightforward at first, there are many special cases that make the construction of a kernel graph a complex procedure that requires multiple phases and several dedicated components.

In the rest of the section we discuss in more details some of these problems and the way we handled them in our study: abstract data types (and the issue with non-homogeneous circular lists), opaque pointers, and the presence of uninitialized or invalid data.

4.1 Abstract Data Types

Over the years, to maintain a reasonable quality over its code base, the Linux kernel developers have adopted several design patterns [20]. In particular, the kernel exports a rich set of APIs to manipulate and create complex data structures, such as double-linked lists and trees of various types, thus relieving kernel developers from the burden of reinventing the wheel every time they need to store and organize multiple objects. For this reason, the existing APIs are not tied to a specific type of kernel object but rely instead on predefined data types that can be included in more complex `struct` objects, and in a number of macros to manipulate them.

Figure 2 shows one of the most common example of this pattern, in which several `task_struct` are organized in a doubly linked list using the `list_head` type. While this provides a simple and efficient way to organize data structures, it unfortunately poses a serious challenge to the automated exploration of kernel objects. In fact, if the leftmost `task_struct` in the figure was already identified by other means (for example because it was pointed to from a global variable), simply following the `next` pointer would result in the discovery of the inner `list_head` structure, but *not* of the outer `task_struct`.

In fact, this operation is performed in the source code by using dedicated macros. In the case of the previous example, a developer would invoke:

```
container_of(var, struct task_struct, task)
```

that the compiler pre-processor translates to a snippet of code required to cast the target `list_head` variable `var` to the requested type based on the current offset inside it (as specified

by the field `task`). However, in our analysis we cannot simply mimic the same behavior by subtracting the offset of the list field from `next` pointer and to cast the result to the correct type to obtain a reference to the outer object. In fact, there are many cases in which this approach would lead to wrong results and it is not sufficient to look at the field type or at its value to distinguish these problematic cases. One example is the list rooted in the field `children` of a `task_struct`. While the field points to another `task_struct`, it does so by reaching it at a different offset (in the `sibling` field). Because of this and other similar problems (explained in more details later in the paper) it is not possible to systematically apply the "subtract and cast" strategy.

For each pointer in a data structure we need to know where — in terms of object type and offset in the target structure — it points to. Other works that built a map of the Linux kernel [1, 24, 35] solved the problem by manually annotating the source code. While this was doable for old kernel versions (e.g., 2.4), it would take many weeks of tedious work to annotate a recent kernel – which today uses more than 6000 different data structures and more than a thousand instances of `list_head`s. Moreover, manual annotations are error prone and are tailored to one specific code base, thus requiring to be verified and modified whenever a new kernel version is released. The compiler community has also already extensively studied the points-to problem [9, 14, 15, 22, 30, 34]. Unfortunately, the techniques they proposed are not suitable to our work as they tend to favor speed (an important factor at compile-time) over precision [4] (a more important factor for our analysis). Only four previous studies automatically extracted a type graph of a kernel [4, 17, 19, 29]. However, none of their systems is available: in one case because the authors relied on the internal source code of the Microsoft Windows operating system [4], and in the other because the entire work was lost [17].

For this reason, we decided to implement our own points-to analysis – which consists of a `clang` plugin that reasons on the Abstract Syntax Tree (AST) of each kernel compilation unit. Contrary to standard points-to analysis, our approach focuses only on the *type* information. More precisely, traditional solutions are designed to identify where each pointer points to, while in our case we only need to extract the target structure, and the offset inside that structure. The result is a *type graph* of the kernel under analysis. To extract this information we take advantage of the fact that the information we need can be inferred by analyzing the source code of the kernel that is in charge of manipulating the data structure in question. Our plugin explores the AST until it finds a call to a kernel API related to data structure management. At this point it analyzes the parameters and resolves their structure type and field name. An example of API call and respective AST is given in Figure 3. In the example, a call to the API `list_add` is used to append the `new` task at the beginning of the list rooted at `head->tasks`. This give us the information that the field `tasks` of `task_struct` indeed points to the very same type. Our plugin current supports `list_head`s, `hlist_head`s (used in the implementation of hash tables), and `rb_root` (used in the implementation of red-black trees).

Except for those, the most common type that is still not supported by our prototype is `radix_tree`, which however is only used 8 times in the entire kernel code base.

As we will show in Section 4.6, our approach is very effective and was able to resolve the type pointed by 250 global lists and by more than 1110 unique object fields in the Linux kernel 4.8, compiled with the Ubuntu 16.04 kernel configuration. Moreover, while our approach is tailored to the Linux kernel, it can be adapted to work on any other operating system, given the availability of its source code. Finally, since the parameter resolution routine does not perform complex analyses, our analysis does not introduce any significant overhead at compilation time.

Circular Lists of Non Homogeneous Elements

As we already discussed in the previous section, certain linked list can chain together object of different types. Since the code must have a way to determine to which type the target element belongs to, this pattern is only present in the form of a "root" object which is the first element of a circular list of otherwise homogeneous objects.

As a consequence, these lists can *only* be traversed starting from their root node, as traversing the loop from an intermediary objects can result into unexpectedly reaching the root node (of a different type) when dereferencing one of the `next` pointers. For example, other than the already cited **children** field of `task_struct`, also the `thread_node` field of the same structure points inside a `signal_struct` object.

To avoid this problem, our analysis classifies every `list_head` field in one of the following three categories: *root* pointer, *intermediate* pointer or *homogeneous* pointer. The first two are used to mark `list_head` fields that belong to lists that contain mixed types, while the latter describes the more common case of homogeneous list. For instance, `task_struct.children` is a root pointer, `task_struct.sibling` an intermediate and `task_struct.tasks` a homogeneous one. This classification can be automatically derived from the type graph: whenever two objects of different types are involved we label the first as *root* and the second as *intermediate*, while all the other objects are labeled as *homogeneous*. During the exploration phase, depending on the type of the pointer, we adopt a different strategy:

- *homogeneous* pointers can be explored by our algorithm in any order.
- *root* pointers require instead our algorithm to immediately walk and retrieve the objects of the entire circular list.
- *intermediate* pointers are ignored since we do not know if they point to another intermediate element or to a

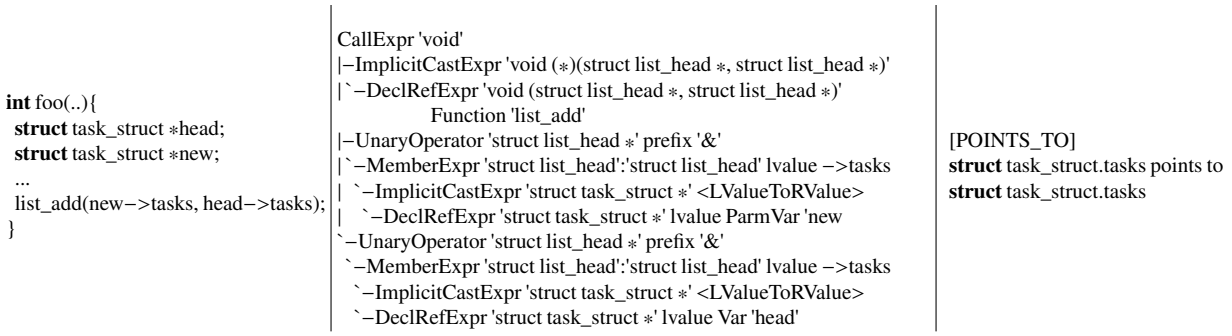


Figure 3: On the left a call to `list_add`, in the center its simplified AST representation, and on the right the plugin output.

root head. This case happens when we enter a circular list from one of its middle elements. This pointer will eventually be explored when the corresponding root node will be visited.

This classification works for every list encountered during the exploration phase, *except* for global `list_head` variables which are always marked as *root* node. In this case, during the very first part of the exploration, these lists are walked entirely and their elements appended to the worklist.

4.2 Uninitialized and Invalid Data

During our data structure exploration, there are cases that could potentially introduce false nodes to our graph. This is due to pointers that contain valid memory addresses but are not yet initialized or that were not valid at the time the snapshot was acquired. One common cause for these errors is the fact that most of the memory management kernel APIs do not initialize to zero the allocated memory. As a result, if an object contains an array of pointers there is not way to tell if one element points to an initialized object (except if the pointer has an invalid value). Another source of false-positives comes from the *non quiescent* state in which the kernel might be when the snapshot is taken [16]. In other words, this means that the kernel could have been in the middle of updating a data structure, leaving dangling pointers in the snapshot. Finally, even if very rare, kernel bugs can contribute to the generation of similar errors.

For these reasons we implemented two sets of heuristics to check if an object is valid or not. The first *soft* rule checks that the number of valid pointers in a kernel object is greater or equal than the number of invalid ones (after removing null pointers and the pointers which normally point to userspace memory, such as the ones contained in `struct sigaction`). The second, more precise, heuristic immediately flags an object as invalid if certain conditions are not verified (such as kernel objects that contain a negative spinlock, or those with function pointers that do not point in the executable sections of the kernel). Finally, we require that, whenever present, a `list_head` has to be valid, i.e. its `next` and `prev` pointer must point to addressable memory. If these rules are not met, we consider the object invalid and discard it from our analysis.

4.3 Opaque Pointers

Opaque pointers, as represented by `void*` fields or by `long long` integers that contain at runtime the address of other objects, are traditionally one of the hardest obstacle to build a complete map of kernel objects. Luckily, this is not the case in our particular scenario. Since we are interested in using our graph to analyze and improve existing memory forensic techniques, opaque pointers play a very marginal role (if any at all) in this space. As they can point to potentially any structure, and the actual target type can change over time, traversing these pointers can be unpredictable during a post-mortem analysis. Even if none of the heuristics we encountered in our experience make use of them, we decided to include them in our graph. After the exploration ends, in case the target of an opaque pointer was discovered by other means, we create the resulting edge, clearly marking it. In this way, we are able to detect if any of these edges are traversed during our experiments. Finally, it is important to understand that these limitations cannot lead to "wrong" results (since they cannot create erroneous paths in the graph), but nonetheless restrict the guarantees of optimality we discuss in the next sections to the constructed graph.

4.4 Limitations and Manual Fixes

Like all previous attempts to build a map of the kernel memory, two particular limitations also affect our solution: unions, and dynamically allocated arrays. Handling the latter case would require more sophisticated code analysis techniques to identify the variable number of elements contained in the arrays, which are beyond the scope of this paper. Nevertheless, we identified few cases of dynamically allocated arrays that contain information that can be relevant for memory forensics and we decided to handle them by hardcoding a custom logic. The first cases are global hash tables where often the size is not inferable from the hash table itself. For example, the `pid_hash` hash table, used by the kernel to quickly locate a process given its

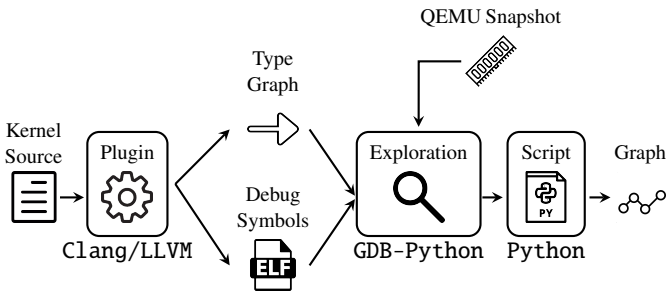


Figure 4: System Overview.

process id, is implemented by using a dynamically allocated array where the size is specified in another global variable (`pidhash_shift`). The second cases are instead dynamically allocated arrays pointed by a kernel object. For example, the file descriptor table associated with each process, which contains the files opened by a process (field `fd` of `struct fdtable`). Once again, this is a dynamically allocated array of `struct file` pointers, and the size can be retrieved from the field `max_fds` of the same structure.

Finally, the handling of `per_cpu` pointers was also hard-coded in our implementation. These are special pointers that, thanks to a double indirection mechanism when dereferenced, give to each processor a different copy of the same variable.

However, we want to stress that this limitation does not invalidate our findings since the graph extracted by our approach is *not* incorrect, but only potentially incomplete.

4.5 Implementation

Our final system is illustrated in Figure 4. It consists of an LLVM compiler plugin to perform the *points-to analysis* on the kernel code at compile-time and a set of python `gdb` extensions that combine the information extracted in the previous step with the information provided by kernel debug symbols to identify all kernel objects contained in a memory snapshot acquired using the QEMU emulator. The kernel exploration routine starts by loading a QEMU snapshot, parsing the type graph, and appending the global object symbols to an internal worklist. At this point the real exploration begins: an object is fetched from the worklist and analyzed using the heuristics we adopted to identify invalid or uninitialized memory. If it is well-formed, each of its field are processed to identify structures, pointers to other structures, or arrays of either type. All them are retrieved and appended to the worklist – paying attention to implement the techniques described above to handle abstract data types. These objects are then processed by a separate component responsible to build the final kernel graph that we will later use to carry out our experiments.

4.6 Final Kernel Graph

We built our kernel graph using *graph-tool* [23], a python library designed to handle large networks. To reduce the size of the graph, we chose to represent with one vertex each *outer* structure identified during the exploration. In other terms we decided to group together, in a single vertex, all the nested structures (but we keep the nesting information as it is needed when we need to move from the graph space back to the memory analysis heuristics). This transformation also makes the graph directed, and result in only one type of edges that represent pointers from a structure to another. As we will thoroughly discuss in Section 5 we assign a number of different weights to each node and edge to allow for several comparisons among different paths.

Figure 5 shows a kernel graph counting 109,000 nodes and 846,000 edges, plotted using Gephi [2]. This graph contains more than 41,000 strongly connected components with the vast majority (95%) containing only one node. On the other hand, the largest one contains 53% of the vertices and has a diameter of 272 nodes. As we will discuss in Section 6, this has important consequences for memory analysis, as it results in a multitude of available paths to move from one node to almost anything else in the kernel memory. The vertex with the highest in-degree is of type `super_block`, pointed by more than 11,000 `inode`s and 11,000 `dentry`s. If we exclude the file system, the node with the highest degree is a `vm_operations_struct`, pointed by more than 4200 `vm_area_struct`s.

In the picture, the size of labels and node is adjusted according to the betwenees centrality of a node. This type of centrality counts how many shortest path between every pair of nodes pass through a node. In other terms, the larger the size the more often a node is present inside every shortest path. The node color depends instead on the kernel subsystem the object belongs to. By using the name of the file where the object is defined we were able to classify them in roughly 7 classes, from file system to object related to memory or process management.

5 Metrics

In the previous section we described how we extracted a global map of a running kernel that can serve as basis for our analysis. However, without any further information, the only way we can compare two paths on the graph is by looking at their *length*, computed by counting either the total number of nodes or the total number of unique structures that need to be traversed. In fact, this simple approach may resemble the one adopted today by most of the memory forensic tools, where the most straightforward path is often chosen by the developers. However, this solution does not tell anything about the *quality* of a given path, nor about the presence of better options to solve the same problem. To get a solid foundation on which we can compare different techniques we need therefore to define a metric. And since the idea of having an absolute metric is

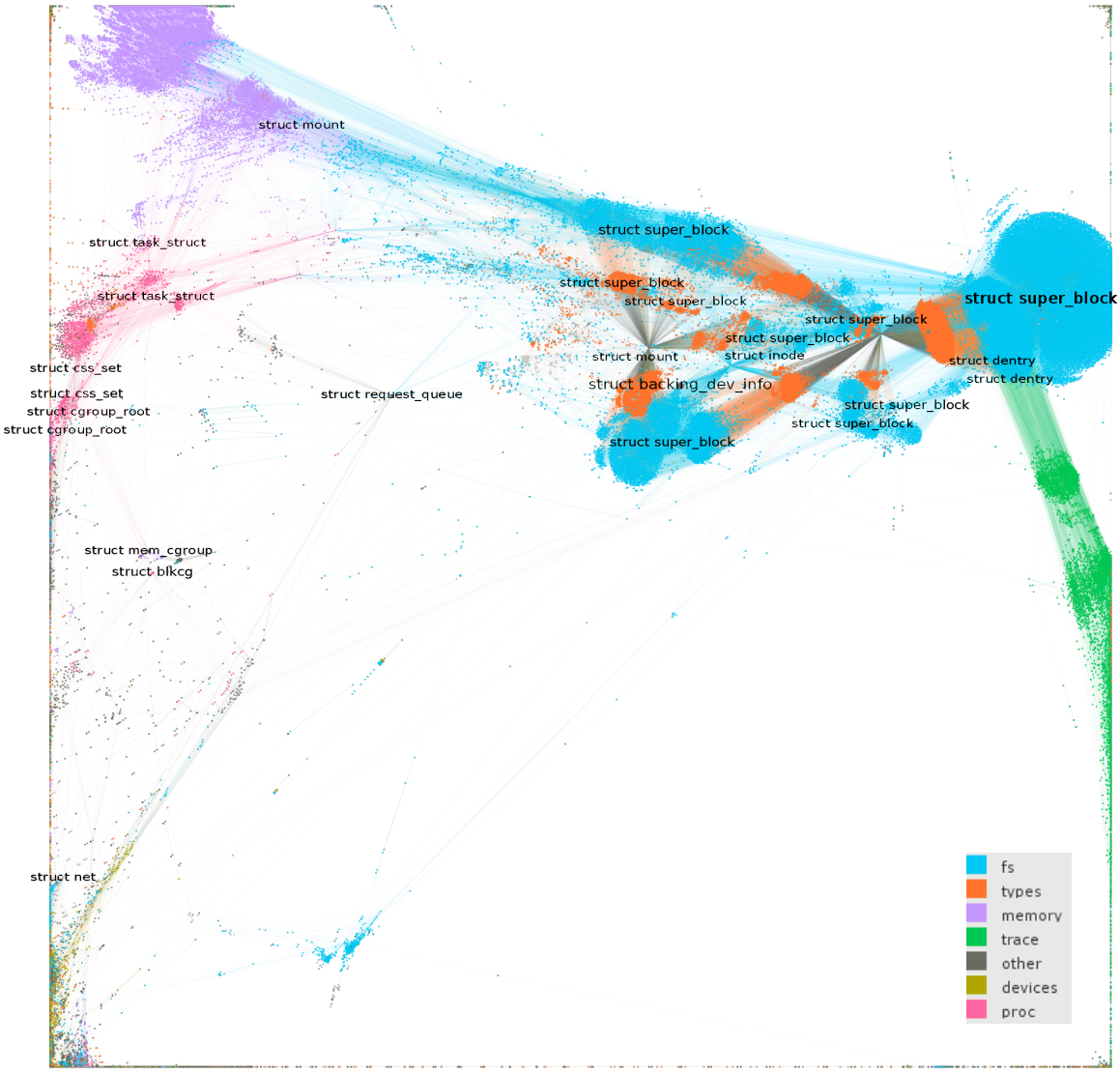


Figure 5: Kernel Graph

unrealistic, multiple different metrics can be plugged on our graph to study the characteristics of each path.

For our experiments we decided to investigate and add to our graph three numerical and two boolean weights, related to the atomicity, stability, generality, reliability, and consistency of a path. As described below, all of them capture different but important aspects of what an analyst may expect from a memory analysis routine.

Atomicity (numerical)

This weight express the distance in physical memory between two interconnected kernel objects. While this metric is expressed in terms of distance among physical pages, for an easier interpretation we often express it in seconds (as distance in time between the acquisition of the two pages). The atomicity is a very important aspect in most of today's investigation that rely on *non-atomic* dumps. In fact, moving across objects located far apart in memory - and thus acquired far apart on the time scale - can introduce inconsistencies. Intuitively, by using this metric the best path between a pair of nodes is the one which minimize the time-delta among all visited structures, thus passing only thorough objects acquired very close in time. More precisely, we can adopt three distinct ways to measure Atomicity:

- `Acquisition Window` (AW) – this is the total window

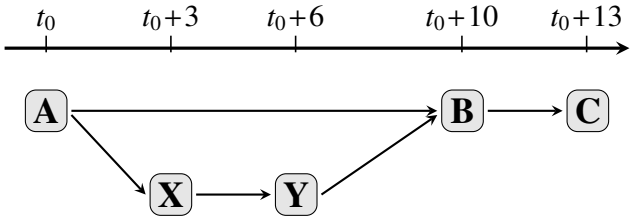


Figure 6: Time acquisition of nodes belonging to two paths.

that covers all data structures traversed in the path. E.g., one path may walk fifteen objects, all of which were acquired in a period of 23 seconds.

- `Cumulative Time Gap` (CTG) – this is the sum of the time difference of each edge traversed in the path. For instance, if a path visits three consecutive nodes (*A*, *B*, and *C*) and the difference between the acquisition time of the pointer in *A* and the content of *B* was 7 seconds and the difference between *B* and *C* was 3, the CTG would be 10 seconds.
- `Maximum Time Gap` (MTG) – this just takes into account the longest "jump" in a path. In the previous example, this would be 7 seconds.

All three measures are related to the Atomicity, but they capture different aspects. If it is important than none of the visited structures have changed during the acquisition, AW is the best metric. CTG gives instead a cumulative probability that things can go wrong by following links. The more edges are traversed, and the more far apart are the objects on the end of those edges, the more likely it is than a link can be corrupted due to the non-atomicity of the dump. Finally, MTG provides an estimation of the single most fragile edge in a path. This can be an important information, as traversing 10 edges each one a second apart can be a better option than traversing a single link with a nine seconds delay in the acquisition.

This can lead to some counter-intuitive results. For example, let suppose our graph analysis identifies two paths to reach a certain target structure *C* namely `{A` → `B` → `C}` and `{A` → `X` → `Y` → `B` → `C}` (for simplicity we ignore the name of the pointers). Both paths start from a structure `A` but the first traverses a single node `B` before reaching the destination while the second takes a detour through two other intermediate data structures `Y` and `Z` before re-joining the first path. Figure 6 shows the two paths on a time scale, that represent at which time the memory containing each data structure was collected.

While the second path is obviously a longer variation of the first, and therefore seems logical to believe that has nothing better to offer, it is very well possible that the detour reduces the probability of incurring in broken links. The pointer `A` → `B` was in fact collected 10 seconds before the object `B`, while the longest path decreases these time gaps to a maximum of four seconds. Whether this is an advantage or not depends on how often those pointers are modified in a running kernel, which we capture with our next metric.

Stability (numerical)

This weight expresses the stability over time of a given node or edge on the graph. Some structures are allocated at boot time and are never modified afterwards, while other parts of the graph are very ephemeral and contain structures that get allocated and de-allocated multiple times per second. By computing a heat-map of the stability of each edge (extracted by processing a number of consecutive snapshots), this weight can provide a valuable information on how the kernel map evolves over time, on which paths are more stable, and on which are instead more ephemeral and may only exists for short periods of time.

We measure Stability by computing the `Minimum Constant Time` (MCT) of all links in a path. The MCT can tell, for instance, that over a certain number of memory images all edges traversed by a certain heuristic remain constant for a minimum time of 30 seconds. In our experiments, we computed this metric by taking a snapshot of the same system at seconds 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 20, 30, 40, 50, 60, 100, 200, 350, 700, 1000, 3000, 5000, 8000 and 12000.

Surprisingly we found that the 81% of edge are stable, i.e., they never change across our experiments. The majority of them are objects related to the file system (`inode` and `dentry`), which the kernel caches for performance reasons. But this does not mean the graph does not evolve, actually quite the opposite. For example, we saw an increase of more than 60% of both nodes and edges between the graph built at $t=0$ and the one built at $t=700$.

Moreover, if we exclude the filesystem subsystem and the paths that remained stable over all our experiments, 11% of the edges changed in less than 10 seconds, 12.5% in less than a minute, and 97% in the first hour.

Generality (numerical)

This weight captures another important problem of memory forensics: the constant change in the layout of kernel objects. This is due to several factors. First of all the kernel is always under active development which means that fields are continuously added to and removed from kernel objects definitions. Moreover, the layout is also influenced by the configuration options chosen at compile time. Existing tools mitigate this problem by requiring additional compile-time information (part of what it is normally called an *OS Profile*). Unfortunately, there are cases in which this information is not available, which today greatly complicate (if not completely preclude) the ability of analyzing a particular memory dump. Therefore, it would be interesting to compute analysis paths that traverse structures which change very rarely across different distributions, kernel versions, and enabled kernel options.

For this reason we downloaded 85 kernels from the Ubuntu repository, spanning from version `4.4.0-21` to `4.15.0-20`. For every object defined in each of these kernels we extracted the offset of the fields required for navigation – such as structure pointers or array of structures. We aggregated this information in a single `Kernels Counter` (KC) weight computed by counting over how many of the 85 kernels an entire path would remain constant (i.e., all its traversed link were present at the same offsets in their corresponding structures).

Reliability (boolean)

This is a very important aspect in memory forensics and captures how tamper-resistant is a given path on the graph, assuming an attacker is capable of reading and writing arbitrary kernel memory. Some paths are very easy for an attacker to modify, and therefore cannot be trusted by an analyst whenever she suspects the attacker might have gained admin privileges on the machine. On the other hand, other paths are more robust, as breaking them would make the system unstable. This can potentially result in programs malfunction or termination and, in the worst case, in a crash of the entire operating system. The robustness of individual data structures has already been studied in the past by several works [1, 10, 25]. But here we are instead interested in the reliability of a path, i.e., not in the fact that individual fields (such as a file name) can be modified, but whether an attacker can tamper with the edges that need to be traversed to prevent a certain heuristic to reach its destination (to the best of our knowledge, this problem has never been addressed in the literature). Being able to compute a path on the graph that only traverses tamper-resistant edges may have a great impact on memory forensics. While today we still do not have enough information to color the entire graph according to this metric, we can still compute the reliability on demand. This means that we cannot compute the optimal solution according to its reliability, but once we have a candidate solution we can perform experiments to verify it.

Consistency (boolean)

As a final property in this list we want to show how metrics can also be aggregated to capture more complex properties of a path. For this example we chose to combine the *stability* and *atomicity* of a path in a single measure that captures how likely it is for a given path to traverse consistent information. Intuitively, traversing a path whose nodes were acquired over a period of 20 seconds may be acceptable if those structures change very rarely, but completely unacceptable if its links are modified every few milliseconds. We capture this aspect by consider a path *consistent* if and only if the acquisition gap of each edge is lower than the minimum change time of the edge as computed in all our snapshots.

6 Experiments

We now discuss how our graph-based framework can be used in different scenarios, in which we investigate existing techniques used by Volatility [33], we discuss the intricacies of computing optimal paths, and we discover new solutions to reach all processes running in a system. In any case, these are only examples of what can be achieved by adopting a more systematic approach to memory forensics, and many more applications can benefit from our framework.

All experiments were conducted on a QEMU machine equipped with 2GB of RAM and 4 virtual CPUs, running `wordpress` on top of a `LAMP` stack. Before and between the acquisitions, we generated some activity by visiting the CMS pages and performing basic system administration task, such as logging in via `ssh` and updating the list of packages.

6.1 Scenario 1

In the first scenario we want to apply our methodology to study the quality of current memory forensics techniques. For our example we selected seventeen Volatility plugins that explore different subsystems (process, network, and filesystem) and mapped them as *paths* in our kernel graph. To achieve this we manually analyzed each plugin and extracted which global variables and kernel objects are traversed. With this information we were able to write a python script which automatically extracts these paths from our graph. Note that many plugins traverse similar kernel structures (e.g `linux_pslist` and `linux_pstree`) so, to avoid duplicates, we only report results for a subset that rely on different information. The final list of the plugins we analyzed is reported in Table 1.

Before looking at the individual metrics, we wanted to investigate to which degree the structures traversed by these heuristics are interconnected. The total number of unique objects used by this heuristics depends on the size of the graph. In our experiments they vary from 20 to hundreds of thousands. As we already introduced in Section 4.6, by averaging over the 25 graphs we created, more than 96% of the nodes used by the heuristics belong to a single giant *strongly* connected component that contains on average 53% of all the nodes in the graph. By combining this information with the nodes visited by Volatility, we found that this component contains all the information related to running processes, such as their mapped memory and open files, but also the information related to the `arp` table and the `tty`s. The remaining 4% of the nodes used by the Volatility rules are instead scattered among several other components. The biggest one, which contains only 0.5% of the heuristics nodes, contains the information related to the installed modules and, more in general, to the `kobject`s subsystem. Finally, the rest of the nodes belong to components containing only a single node. These are the nodes representing, for example, the global `pid_hashtable` and its associate `hlist_head`s.

Table 1: Comparison of Volatility plugins implemented as paths in our graph

Name	Description	# Nodes	Atomicity AW	Atomicity CTG	Atomicity MTG	Stability MCT	Generality KC	Consistency *Fast*	Consistency *Slow*
`linux_arp`	Prints the ARP table	13	16.24	53.25	16.24	12,000	50/85	✓	✓
`linux_check_afinfo`	Verifies the function pointers of network protocols	24	16.27	44.55	16.05	700	85/85	✓	✓
`linux_check_creds`	Checks processes that share credential structures	248	16.34	453.92	16.24	2	29/85	✓	✓
`linux_check_fop`	Check file operation structures for rootkit modifications	16099	16.38	142,856.15	16.38	0	29/85	✗	✗
`linux_check_modules`	Compares module list to sysfs info, if available	151	16.27	54.06	16.23	700	85/85	✓	✓
`linux_check_tty`	Checks `tty` devices for hooks	13	16.26	17.52	15.69	30	85/85	✓	✓
`linux_find_file`	Lists and recovers files from memory	14955	16.33	35,627.45	16.32	0	85/85	✗	✗
`linux_ifconfig`	Gathers active interfaces	12	16.25	44.19	16.25	12,000	50/85	✓	✓
`linux_iomem`	Provides output similar to `/proc/iomem`	7	16.70	50.09	16.70	12,000	50/85	✓	✓
`linux_lsmod`	Lists loaded kernel modules	12	16.23	44.27	16.05	700	85/85	✓	✓
`linux_lsof`	Lists file descriptors and their path	821	16.33	19,885.52	16.26	0	29/85	✗	✗
`linux_mount`	Lists mounted fs/devices	495	16.33	8488.13	16.32	10	85/85	✓	✗
`linux_pidhashtable`	Enumerates processes through the PID hash table	469	16.67	451.87	16.67	30	31/85	✓	✗
`linux_proc_maps`	Gathers process memory maps	4722	16.27	2629.19	16.24	0	31/85	✗	✗
`linux_proc_maps_rb`	Gathers process maps through the mappings rb-tree	4722	16.27	3310.69	16.24	0	31/85	✗	✗
`linux_pslist`	Lists active tasks by walking task_struct→task list	124	16.27	189.41	16.24	30	31/85	✓	✓
`linux_threads`	Prints threads of processes	157	16.27	280.68	16.24	30	31/85	✓	✓

This is an important finding, as it means that the vast majority of the information needed for forensic purposes is interconnected and reachable from one another. Translated in practical terms, the presence of this giant connected component means that is enough to locate a single kernel object to reach all the other interesting ones only by dereferencing pointers. This might be beneficial in scenarios where the position of global kernel objects is not available to the analyst. In such cases, one entry point can often be located by memory carving and then used as starting point for every other analysis.

By looking at the atomicity metrics (columns four-to-six in Table 1) the first thing that stands out is that the values for the Acquisition Window (`AW`) and the Maximum Time Gap (`MTG`) are very similar and relatively constant across all commands. After further investigation we discovered that this is due to the fact that, when compiled with normal configurations, Linux kernel global variables are located in the low part of the physical memory while kernel objects are allocated in the higher end. Since all heuristics start from global symbols, the very first edge already accounts for the maximum gap between two consecutive kernel objects. This also influences the acquisition window, since one of the two farthest objects is always the global variable from where the heuristic starts from. On the other hand, the Cumulative Time Gap (`CTG`) shows more variations as it is also influenced by the number of traversed objects.

To better understand this phenomenon, in Figure 7 we plotted the content of the physical memory as a Hilbert curve. In the graph, each pixel represents a physical page and its color shows if the page is traversed by the Volatility heuristics (red in the graph) or if it contains at least one node of our connected component of the kernel graph (green). It is clear that the relevant data structures are not spread equally on the entire physical memory. Instead, they clearly aggregated around three main clusters, which we marked respectively as C1, C2, and C3. In our experiments the kernel global variables are all located in C1 while other information are often stored in C2 and C3. Therefore, most heuristics start from C1 and

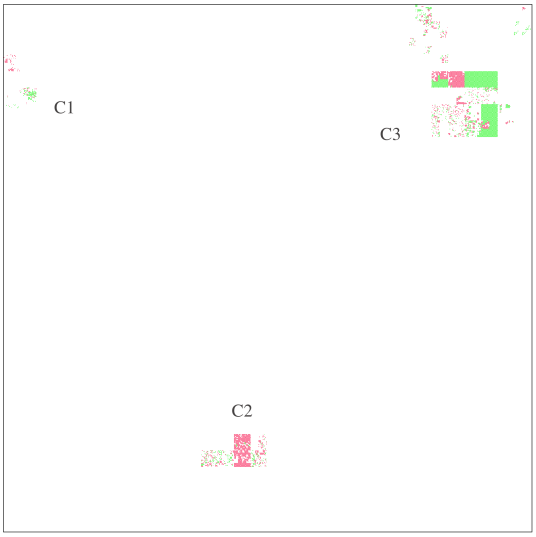


Figure 7: View as Hilbert curve of physical memory.

then eventually traverse an edge towards one of the other regions - which alone is responsible for the entire `AW` and `MTG` metrics. This physical distribution is also very important for the third scenario presented in Section 6.3, where we will encounter heuristics that need to hop back and forth from the three clusters, significantly impacting the atomicity metrics.

The second surprising result of this first scenario is the fact that the Kernel Counters (`KC`) of six plugins *never* changed across all the different kernel versions we used in our analysis. This means that, even when fields were added or removed from these object, the offsets of the fields used by the plugins remained constant. This has important implications for current memory forensics tools where a *profile* of the kernel is needed to analyze a memory dump. Our experiment suggests that, at least for locating certain information, a generic structure layout can be used across almost 100 kernel versions, released

as far as 2 years apart.

Another important propriety we evaluated in this first scenario is the consistency of the selected techniques. This is especially useful to better understand how the continuous modifications of kernel objects might impact memory dumps taken in a non-atomic fashion. This was recently listed by Case et al. [6] as "*one of the most pressing issues*" of memory forensics. While Case focused on page smearing (an inconsistency between the page tables and the referred physical memory), with this experiment we show that this problem does not affect only page tables but also references among kernel objects. The most important variable that influence the consistency of the memory is the duration of the acquisition process. To align with real world scenarios we run two different tests, by setting the acquisition ratio respectively to the fastest and to the slowest tool as reported by McDown et al. [21]. In that study, the authors compared seven different memory acquisition tools, chosen from a survey conducted over 41 companies specialized in memory forensics.

Interestingly, out of the 17 plugins we tested, three have a stability of 12,000 seconds, which means that none of the links they traversed *ever* changed over a period of more than three hours. At the other end of the spectrum, eleven plugins walked links that remained stable for less than a minute (and in five cases even less than one second). In this case, this may result in wrong pointers depending on how far in the physical memory were the page containing the link and the page containing the linked object. In fact, the last column of Table 1 shows that our analysis found inconsistencies in five (when the fastest tool to acquire the memory was used) or seven (in the case of the slowest solution was used) plugins. The affected plugins interest different parts of the kernel, but they can be divided to three distinct categories: *Memory* (`linux_proc_maps`, `linux_proc_maps_rb`), *File system* (`linux_check_fop`, `linux_find_file`, `linux_lsof`, `linux_mount`) and *Process* (`linux_pidhashtable`)

In the Memory category we found respectively 33 inconsistencies that affected the connections among `vm_area_struct` of a process, which are kept both in a linked list and in a red-black tree. These errors affected five instances of `apache`, one of `systemd-login` and one of `agetty`. The filesystem category included 40 unique inconsistencies in the hierarchy of dentries (fields `d_subdirs` and `d_child`) 53 in the mapping from a `dentry` to an `inode` (field `d_inode`). The latter object was also involved in 43 cases of inconsistency towards its `file_operations` object (field `i_fop`), while 23 `file` object had inconsistent edges pointing to their `dentry` and its `mount` objects. (field `f_path.dentry` and `f_path.mnt`). The most interesting cases of inconsistencies in this category – 10 in total – involved the array containing the pointers to the `file`s opened by a process. This array belonged to three distinct instances of `apache`, one of `systemd` and one of the `mysql` database. In the process category, we only detected one case of inconsistent edge between a `struct pid` and the pointed `task_struct`.

To systematically understand if these inconsistent paths can be avoided, we used once again our kernel graph – this time by filtering out *all* the 5,000 inconsistent edges, and searching for alternative paths to reach the same objects used by the affected plugins. Our graph exploration was able to discover alternative paths for 107 out of 213 inconsistent edges. For example, in the case of inconsistent array of opened files for the `systemd` process the alternative path — which traversed 11 additional nodes — was able to reach the target `file` by first locating the `task_struct` of the same process, then accessing its corresponding `files_struct` and from here reaching the `file` via the `fd_array` field (an array only used when the process opens less than 64 files). While these detours were sufficient in our experiments to retrieve the missing information, more experiments are required to understand if those alternative paths can be generalized to other scenarios. In any case, they show once more that the giant connected component that hosts most of the relevant data structures may allow analyst to find alternatives solution to mitigate the presence of wrong pointers and inconsistent information. Sadly, almost 50% of the affected pointers did not allow for an alternative path, thus emphasizing again the severe consequences that the lack of atomicity can have on memory analysis.

6.2 Scenario 2

In our second case study we want to understand if we can employ the kernel graph to find new heuristics for common forensics tasks. In particular, we focus on the starting point of many forensics investigation: listing the processes running at the acquisition time. Currently Volatility implements three different plugins[1] to list the processes, respectively by walking the process list, by using the `pidhash` hashtable, and by parsing the kernel memory allocator. However, the latter is only applicable if the kernel uses the `SLAB` allocator. Unfortunately, many distributions, such as Ubuntu and Debian, ships by default with the `SLUB` allocator, which is not supported by Volatility and which does not keep track of full *slabs* – thus making this technique not applicable anymore.

The main reason for looking for alternative solutions is that previous research already pointed out that rootkits are already capable of removing a process from the process list, but also to unlink a process from the pid hashtable [19,26,27] thus leaving the forensic analyst without a reliable method to list processes. Moreover, as we already discussed in the previous scenario, the lack of atomicity of a memory dump can introduce inconsistencies and result in broken pointers also in the list of running processes. For these reason, it is important to find new ways to locate processes, so that their output can be compared with other techniques to spot inconsistencies or hidden processes.

[1] Volatility also includes a plugin to carve `task_struct` objects by using a signature, but this is a parallel approach that does not require exploring memory but relies instead on pattern-matching.

Table 2: Comparison between different heuristics used to find processes

Category	Root Node	New	# Nodes	# task_struct	Atomicity AW	Atomicity CTG	Atomicity MTG	Stability MCT	Generality KC	Reliability	Consistency
scheduling	runqueues	✓	9	4	16.71	20.08	16.70	0.00	34/85	—	✗
	root_task_group	✓	10	4	16.65	21.14	16.27	0.00	18/85	—	✗
cgroup	css_set_table	✓	172	156	16.27	433.32	16.24	10.00	29/85	✗	✗
	cgrp_dfl_root	✓	186	156	16.30	369.10	16.30	10.00	29/85	✗	✓
memory/fs	dentry_hashtable	✓	58383	23	16.31	58120.38	16.30	0.00	36/85	✗	✗
	inode_hashtable	✓	14999	23	16.32	31594.48	16.31	1.00	36/85	✗	✗
workers	wq_workqueues	✓	427	69	16.68	1727.89	16.24	200.00	39/85	✗	✓
process	init_task (linux_pslist)	✗	124	124	16.27	189.41	16.24	30.00	31/85	✗	✓
	init_task (linux_threads)	✗	156	156	16.27	280.68	16.24	30.00	31/85	✗	✓
	pid_hash (linux_pidhashtable)	✗	469	156	16.67	451.87	16.67	30.00	30/85	✗	✓

This scenario is also interesting as it is harder to translate into a graph exploration problem. In fact, since we are looking for techniques to list all (or a part of) the running processes, this is equivalent to a *collection* of, possibly not homogeneous, paths. As a result, listing all processes is not simply equivalent to a path, but more to an *algorithm* to explore the graph.

Our approach to find new heuristics is the following. First, we discarded all the global roots that do not have a path to reach all the `task_struct`s in *every* graph we created. As a result, we were left with 621 global roots (out of more than 8000 we started with). Second, we modified the graph to remove the edges already used by known techniques, such as the `tasks` field. This helps removing all those paths that would just find a different way to reach a single process, and then walk the list like the existing plugins already do. While not useless per se, our goal is to find *new* solutions and not variations of the existing ones.

By only considering the *shortest* paths from every root node to every task structure, our system found more than 100 million distinct paths, generated from a set of more than 966,000 sequences of vertices. This is possible because, as we later discovered, the graph contained many parallel edges connecting the same nodes. In fact, by putting things in perspective, on average every sequence of vertices from a root node to a target object generates more than 100 unique paths. The good news is that this makes extremely difficult for attackers to modify all edges required to completely hide a process. On the other hand though, this also makes very hard the task of identifying interesting patterns in this multitude of options. For simplicity, we first decided to filter out all *similar* edges – i.e., parallel edges that shares the same metrics (and that therefore are equivalent for our purpose). This operation removed more than 300,000 edges, some of which played an important role in the path explosion. For example, many entries of the array `e_cset_node` of the `css_set` object pointed multiple times to the same vertex. After this operation the number of different paths decreased to about 7.5 millions paths.

We then merged similar paths into *templates*, constructed by keeping only the type of the objects present in the path, and by also removing adjacent nodes with the same type (which capture the ↻ link discussed in Section 3). Finally, we removed templates that were subset of other templates, resulting in a final set of 4067 path templates.

By manually exploring these options, we soon realized that they belong to only four main families, depending on the kernel subsystem they live in. The first one is related to the *cgroup* subsystem, the second to the *memory* subsystem through the `mm_struct` structure, the third passes through the *work queues* to reach kernel workers, and the last traverses the `struct rq` and follows the `curr` field, a per-cpu runqueue. The results are summarized in Table 2.

Unfortunately, there are no alternative paths that can improve the atomicity. In fact, the bulk of the time gap (16.24 seconds) is due to the difference in the acquisition time of the global entry points (located in C1 in Figure 7) and the first task structure (located in C2 and C3). However, all these edges are very stable and in only one case (for the `css_set_table`) the value of this first connection ever changed during our memory acquisition.

The memory-based heuristics walked a red-black tree (`i_mmap`) that is very ephemeral and, while exploring it, we found more than 30 edges that could be inconsistent if the memory dump is not taken atomically. A similar problem affects the scheduler, whose structures also contain links that change very rapidly. We observed an interesting phenomenon in the cgroup-related heuristics. The first is inconsistent as it traverses a pointer with a very large time gap. However, the second avoid this problem by reaching the same `css_set` structures by taking a detour through several intermediate objects which act as a bridge to lower the time gap. This is an example of the counter-intuitive behavior we introduced in Section 5 (Figure 6), where we predicted that the most direct path might not always be the best in term of consistency. The worker-related approach was the best in terms of stability, consistency, and generality. However, its goal is to list all active kernel workers and therefore this heuristic is unable to capture normal userspace processes. Finally, the two heuristics in the process category, which represent the Volatility plugins `linux_pslist` and `linux_threads`, had both a stability of 30 seconds. This is strange, as several processes should have started during this

time frame. However, new processes were all appended to the tail of the process list without altering the intermediate nodes.

To test the Reliability of the heuristics we wrote a kernel module that tries to hide an userspace process by unlinking it from the path required by each heuristic. As a result, each case required a custom hiding technique. For the cgroup heuristics we deleted the processes from the `cg_list` linked list. For the memory we first found every non-anonymous, i.e. backed by a file, `vm_area_structs`. We then delete all this structures from the red black tree rooted in the `inode`, which keeps track of all the `vm_area_struct` which are currently mapping this file. For the first two process heuristics, we removed the process from the process list (by unlinking `task_struct.tasks`), while for the `pid_hash` we removed the `struct pid` from the hashtable. For the workqueue we instead created a custom workqueue and queued a simple work function that mimicked the behavior of the userspace process we used in our test. We then proceeded by unlinking the worker from the linked list rooted at `worker_pool.workers`.

In all the cases our program continued to run without observable side-effects – showing that each path we listed so far can be tampered with by a properly written rootkit. As we also discussed in Section 5, we believe that more experiments are needed to improve the assessment of a path's reliability. While it is true that our program continued to run, there can be a multitude of events (e.g. the kernel starting to swap memory) that might compromise the stability of the altered system.

6.3 Scenario 3

In the third scenario we show how we can compute optimal paths, with respect to the different metrics we proposed in this work. As running example, we picked this time the problem of finding the files opened by a given process (identified by its `task_struct`).

To run our experiments we collected all the `task_struct` and all the associated `file` objects and analyzed the paths Volatility would take to move from the first to the second. However, we immediately run into a strange behavior, as the metrics were returning very different results for different files. To understand the reason we had to look closer at how the physical pages were assigned to the different kernel objects.

Figure 7 explains very well the three classes of behavior we identified in our experiments. Since the clusters (C1, C2 and C3) are located far apart in memory (and therefore they can be acquired far apart in time), whenever a heuristic moves from one structure contained in one cluster to another contained in a different one, it needs to take a "jump" with associated a considerable time gap. If a `task_struct` and all the intermediate objects needed to reach the open files are located inside the same cluster, then time gaps are extremely small and path are always consistent. In this case paths are already optimal and there is no much room for improvement. If they are instead located in two different clusters, then the atomicity increase by almost nine seconds. However, the picture shows that also in this case it is not possible to find better alternatives, as all paths would need to cross the gap between the clusters– incurring in the same penalty. Finally, there are examples in which the `task_struct` and the `file` objects were located in the same cluster, but the intermediate structures traversed by Volatility resided in the other one. In this case the Volatility heuristic needs to jump across clusters twice, incurring twice in the risk of inconsistent links. But in this third case it might be possible to use our graph to find an alternative path that is fully contained in the same cluster.

An example of each of these three cases is shown in Table 3, along with the metrics computed on the Volatility heuristic and those computed on the optimal paths extracted from our graph. Regarding the cumulative time gap (CTG), our insight was correct and only paths belonging to the third category could be considerably improved. In fact, the table shows that from more than 17 seconds in the Volatility case, the optimal path had a CTG of less than 0.01 seconds. Accordingly, also the MTG decreased with the same magnitude. As we discussed in the previous scenario, finding a consistent path for this particular problem is sometimes possible. Indeed, when this is the case, we were able to find a path that remained stable for all our experiments. Interestingly, for the second case, one of the paths with maximum stability has also higher generality than the one used by Volatility but, since it passes through more nodes, it has an higher CTG. On the other hand, maximizing the generality of a path has a serious impact to its consistency and stability. In fact, while we were able to find paths which are constant over 50 kernels, none of them was consistent, independently to the speed of acquisition.

7 Discussion and Future Directions

The goal of our work is to provide a principled way to think about memory forensics as a graph-related optimization task. This way of modeling the problem opens the door to a multitude of different possibilities to evaluate and compare existing techniques, design algorithms to compute new alternative solutions, validate the consistency of kernel structures, or propose heuristics customized to different experiments setup and acquired dump.

We tried to discuss some of these opportunities through our experiments, but we are aware that many questions are still open and new research is needed to shed light to each individual use case. For this reason, we decided to release our code and data to other researchers, hoping that this will facilitate new experiments in this field and accelerate new findings based on our methodology.

In this paper we focused on the analysis of traditional computers. This choice was simply dictated by the fact that this is the area where memory forensics is more mature and for which most of the heuristics have been designed so far. Nevertheless, we believe that our system could be used to help researchers to better design and implement future forensic

Table 3: Optimal paths compared with Volatility paths

Name	# Nodes	Atomicity AW	Atomicity CTG	Atomicity MTG	Stability MCT	Generality KC	Consistency *Fast*	Consistency *Slow*
File A – all structures in one cluster								
`Volatility`	4	0.01	0.01	0.01	700	29/85	✓	✓
`Opt-MTG`	4	0.01	0.01	0.01	700	29/85	✓	✓
`Opt-CTG`	4	0.01	0.01	0.01	700	29/85	✓	✓
`Opt-MCT`	4	0.54	0.54	0.54	**12000**	29/85	✓	✓
`Opt-KC`	4	0.01	0.01	0.01	700	29/85	✓	✓
File B – structures located in two clusters								
`Volatility`	4	8.72	8.72	8.72	12000	29/85	✓	✓
`Opt-MTG`	4	8.72	8.72	8.72	12000	29/85	✓	✓
`Opt-CTG`	4	8.72	8.72	8.72	12000	29/85	✓	✓
`Opt-MCT`	11	16.23	72.84	16.21	12000	36/85	✓	✓
`Opt-KC`	8	9.71	46.15	9.71	0	**50/85**	✗	✗
File C – structures located in one cluster, with intermediate steps in the other								
`Volatility`	4	8.73	17.45	8.73	12000	29/85	✓	✓
`Opt-MTG`	3	0.003	0.003	**0.003**	12000	29/85	✓	✓
`Opt-CTG`	3	0.003	**0.003**	0.003	12000	29/85	✓	✓
`Opt-MCT`	3	16.23	82	16.20	12000	36/85	✓	✓
`Opt-KC`	10	9.71	55.56	9.71	0	**50/85**	✗	✗

frameworks tailored to emerging technologies such as mobile devices and the Internet of Things (IoT).

Main findings: our experiments show that a large part of the kernel graph belongs to a giant connected component. This means there are thousands, or even millions of possible paths that allow an analyst to move from one node to another. It also means that it is very difficult for an attacker to completely hide some piece of information from all possible paths.

Another consequence of the interconnected topology of the graph is that it is hard for an analyst to simply inspect all possible paths, looking for new techniques to implement in memory forensic tools. We tried to do this in our second scenario, and run into a path explosion problem even by considering only all shortest paths. However, this effort allowed us to discover two new promising techniques (one based on `cgroups` and one on `workrqueues`) that can complement those used today by Volatility [2].

Sadly, the problem of finding an optimal path turned out to be very delicate and dependent on multiple factors. In fact, the exact memory layout when the snapshot is acquired may affect the metrics associated to different links (e.g., one path may be optimal for one dump but poor in another). This may suggest that maybe, instead of relaying on a single solution, new techniques should try to explore the graph by following many parallel paths.

Moreover, we are aware that some of the metrics we proposed in this paper turned out to be ineffective in the evaluation. However, we decided to include them anyway in the paper for two reasons. First, because we did not know in advance that (for example the Maximum Time Gap) would be irrelevant in the analysis of common Linux kernels. This has nothing to do with the heuristic itself, but with the fact that the kernel allocates global variables (entry points) very far from other objects. We believe this fact to be an interesting finding which came as a consequence of applying our framework. Second, while this is true in our experiments, it is probably not the case on other operating systems or OS kernels. So, we believe it is still interesting to implement and discuss those ineffective metrics in our framework.

Finally, we want to stress the fact that our main contribution is not the discovery of new technique, but the introduction of a model that can be used to *reason* about memory analysis, explore its complexity, and perform quantitative measurements.

Future Work: In this paper we discuss a number of metrics an analyst can use to compare different solutions. However, the list is certainly not exhaustive and we expect more to be defined in the future. More work is also needed to understand which metric is better at capturing certain aspects of an investigation.

Reliability is certainly one of the most important characteristic of an analysis technique. Unfortunately, it is also the only one we discussed that cannot be extracted with automated experiments. More research is needed to fill this gap and enable to compute the reliability of a large amount of links among kernel objects.

Finally, to be useful in practice, our prototype should be applied to a larger number of memory dumps taken from different systems. This could help generalize the results and customize the analysis to an environment that resemble the one under investigation.

[2] We implemented both as Volatility plugins

8 Related Work

The analysis of kernel objects and their inter-dependencies has attracted the interest of both the security and the forensics community. While the common goal, namely ensuring the integrity of the kernel against malware attacks using the inter-dependencies between kernel object, is shared by the majority of works on this topic, the methods and the tools used to achieve it are often different. Several research papers have also been published on reconstructing and analyzing data structure graphs of user-space applications [3, 7, 28, 31]. However, most of these techniques are not directly applicable to kernel-level data structures, because they do not take in account the intricacies present in the kernel, such as resolution of ambiguous pointers to handle custom data structures. For this reason they will not be discussed in this Section.

To better highlight the different approaches, we decided to divide them in two distinct categories. The first one covers approaches which presented the analysis of a *running* kernel. The second category is focusing instead on *static* approaches, which require only a memory snapshot or the OS binary.

Dynamic Analysis

One of the first example of dynamic kernel memory analysis was presented by Rhee et al. in 2010 [26]. The tool, named `LiveDM`, places hooks at the beginning and at the end of every memory-related kernel functions, to keep track of every allocation and deallocation event. When these hooks are triggered, the hypervisor notes the address and the size of the allocated kernel object and the call site. The latter information is used, along with the result of an offline static analysis of the kernel source code, to determine the type of the allocated object. A work built on top of `LiveDM` is `SigGraph` [19]. In this paper the authors generate a signature for each kernel object, based on the pointers contained in the data structure. These signatures are then further refined during a profiling phase, where problematic pointers - such as null pointers - are pruned. The result can then be used by the final user to search for a kernel object in a memory dump. The major concern about signature-based scheme is their uniqueness, that avoids problems related to isomorphism of signatures. The authors found that nearly the 40% of kernel object contains pointers and - among this objects - nearly 88% have unique `SigGraph` signatures. Unfortunately the uniqueness was reported *prior* the dynamic refinement, so it is unclear the percentage of non-isomorphic signatures. For this reasons a kernel graph built using the approach adopted by `SigGraph` would only retrieve a *partial* view of the entire memory graph.

Another work focused on signatures to match kernel objects in memory was done by Dolan-Gavitt et al in 2009 [10]. The main insight of this work is that while kernel rootkits can modify certain fields of a structure - i.e. to unlink the malicious process from the process list - other fields (called *invariants*) can not be tampered without stopping the malicious behavior or causing a kernel crash. The invariant are determined in a two steps approach. During the first one every access to a data structure is logged, using the stealth breakpoint hypervisor technique [32]. Then, the most accessed field identified in the previous step are fuzzed and the kernel behavior is observed. If the kernel crashes then there are high chances that this field can not be modified by a rootkit. On the other hand, if no crash is observed, than the field is susceptible to malicious alteration. The direct results of these two phases is that highly accessed field which result in a crash when fuzzed are good candidates to be used as *strong* signatures. While this approach looks very promising is not easily adaptable to our context since, as also noted by the authors, creating a signature for small structures can be difficult. Furthermore, generating a signature requires to locate at least one *instance* of a structure in memory, which might not be straightforward.

Xuan et al. [35] proposed *Rkprofiler*. This tool combine a trace of read and write operations of malicious kernel code with a pre-processed kernel type graph, to identify the tampered data. The problem of ambiguous pointers is overcome by annotating the type graph with the real target of a list pointer. While it is not clearly stated in the paper, it seems the annotation was manually done. As we discussed in Section 4 the Linux kernel uses a large amount of ambiguous pointers, thus making the manual annotation approach not feasible anymore.

While more focused on kernel integrity checks, `OSck` [16] uses information from the kernel memory allocator (*slab*) to correctly label kernel address with their type. The integrity check are run in a kernel thread, separated by the hypervisor. This two components allow OSck to write custom checkers that are periodically run. While this approach seems promising, only a small subset of frequently-used structures are allocated using *slab* (such as `task_struct` or `vm_area_struct`), and thus is unsuitable for our needs.

Another approach to create a Windows kernel object graph is *MACE* [11]. Using a pointer-constrain model generated from dynamic analysis on the memory allocation functions and unsupervised learning on kernel pointers, MACE is able to correctly label kernel objects found in a memory dump. Once again, while the output of the work is a kernel object graph for Windows, the application only focuses on rootkit detection.

Static Analysis

One of the most prominent work in this field is KOP [4], and its subsequent refinement MAS [8]. Very similarly to our approach, the authors use a combination of static and memory analysis techniques respectively on the kernel code and on a memory snapshot. In the first step they build a precise *field-sensitive* points-to graph, which is then used during the memory analysis phase to explore and build the kernel objects graph. Contrary to this solution, ours does not make *any* assumption about the kernel memory allocator. While the Windows kernel has only one allocator, the Linux kernel has three different ones (`slab`, `slub`, and `slob`). Moreover, only a predefined subset of kernel objects are allocated in custom slabs, while the vast majority is sorted in generic slabs based on their size.

Gu et al. [13] presented `OS-Sommelier+`, a series of tech-

niques to fingerprint an operating system from a memory snapshot. In particular, one of these techniques is based on the notion of *loop-invariants*: a chain of pointers rooted at a given kernel object that, when dereferenced, points back to the initial object. Once a ground truth is generated from a set of known kernels, this loop invariant *signatures* can be used to fingerprint unknown kernels. Unfortunately the paper does not mention the problem of ambiguous pointers, and we believe our graph generation approach could improve `OS-Sommelier`$^+$ detection.

Finally, as we already discussed, none of the tools to automatically build a graph of kernel objects was publicly available.

9 Conclusion

Memory forensics focuses on locating and extracting artifacts from a memory snapshots, using a broad set of custom rules. However, the quality of the existing heuristics is difficult to measure and it is largely based on the experience of the researchers who wrote them. As a result, analysts are left without any clear guidelines on how to compare and evaluate different approaches and how to assess the results they produce.

For these reasons, in this paper we proposed a method to study memory forensics techniques in a principled way. Our solution is based on a graph representation that captures the relationships between all kernel objects, enriched with a set of metrics that covers different aspects of memory forensics. We believe that our framework can help researchers to measure the quality of existing memory forensics techniques, but also to extract qualitatively better heuristics.

References

[1] Baliga, A., Ganapathy, V., and Iftode, L. Automatic inference and enforcement of kernel data structure invariants. In *Computer Security Applications Conference, 2008. ACSAC 2008. Annual* (2008), IEEE, pp. 77–86.

[2] Bastian, M., Heymann, S., Jacomy, M., et al. Gephi: an open source software for exploring and manipulating networks. *Icwsm 8* (2009), 361–362.

[3] Bursztein, E., Hamburg, M., Lagarenne, J., and Boneh, D. Openconflict: Preventing real time map hacks in online games. In *Security and Privacy (SP), 2011 IEEE Symposium on* (2011), IEEE, pp. 506–520.

[4] Carbone, M., Cui, W., Lu, L., Lee, W., Peinado, M., and Jiang, X. Mapping kernel objects to enable systematic integrity checking. In *Proceedings of the 16th ACM conference on Computer and communications security* (2009), ACM, pp. 555–565.

[5] Carvey, H. Digital forensics of the physical memory.

[6] Case, A., and Richard III, G. G. Memory forensics: The path forward. *Digital Investigation 20* (2017), 23–33.

[7] Cozzie, A., Stratton, F., Xue, H., and King, S. T. Digging for data structures. In *OSDI* (2008), vol. 8, pp. 255–266.

[8] Cui, W., Peinado, M., Xu, Z., and Chan, E. Tracking rootkit footprints with a practical memory analysis system. In *USENIX Security Symposium* (2012), pp. 601–615.

[9] Das, M. Unification-based pointer analysis with directional assignments. *Acm Sigplan Notices 35*, 5 (2000), 35–46.

[10] Dolan-Gavitt, B., Srivastava, A., Traynor, P., and Giffin, J. Robust signatures for kernel data structures. In *Proceedings of the 16th ACM conference on Computer and communications security* (2009), ACM, pp. 566–577.

[11] Feng, Q., Prakash, A., Yin, H., and Lin, Z. Mace: High-coverage and robust memory analysis for commodity operating systems. In *Proceedings of the 30th annual computer security applications conference* (2014), ACM, pp. 196–205.

[12] Gruhn, M., and Freiling, F. C. Evaluating atomicity, and integrity of correct memory acquisition methods. *Digital Investigation 16* (2016), S1–S10.

[13] Gu, Y., Fu, Y., Prakash, A., Lin, Z., and Yin, H. Multi-aspect, robust, and memory exclusive guest os fingerprinting. *IEEE Transactions on Cloud Computing 2*, 4 (2014), 380–394.

[14] Hardekopf, B., and Lin, C. The ant and the grasshopper: fast and accurate pointer analysis for millions of lines of code. In *ACM SIGPLAN Notices* (2007), vol. 42, ACM, pp. 290–299.

[15] Heintze, N., and Tardieu, O. Ultra-fast aliasing analysis using cla: A million lines of c code in a second. In *ACM SIGPLAN Notices* (2001), vol. 36, ACM, pp. 254–263.

[16] Hofmann, O. S., Dunn, A. M., Kim, S., Roy, I., and Witchel, E. Ensuring operating system kernel integrity with osck. In *ACM SIGARCH Computer Architecture News* (2011), vol. 39, ACM, pp. 279–290.

[17] Ibrahim, A. S., Hamlyn-Harris, J., Grundy, J., and Almorsy, M. Digger: Identifying os kernel objects for runtime security analysis. *International Journal on Internet and Distributed Computing Systems 3*, 1 (2013), 184–194.

[18] Le Berre, S. From corrupted memory dump to rootkit detection. https://exatrack.com/public/Memdump_NDH_2018.pdf, 2018.

[19] Lin, Z., Rhee, J., Zhang, X., Xu, D., and Jiang, X. Siggraph: Brute force scanning of kernel data structure instances using graph-based signatures. In *NDSS* (2011).

[20] LWN. Linux kernel design patterns - part 2. https://lwn.net/Articles/336255/, 2009.

[21] McDown, R. J., Varol, C., Carvajal, L., and Chen, L. In-depth analysis of computer memory acquisition software for forensic purposes. *Journal of forensic sciences 61* (2016), S110–S116.

[22] Pearce, D. J., Kelly, P. H., and Hankin, C. Efficient field-sensitive pointer analysis of c. *ACM Transactions on Programming Languages and Systems (TOPLAS) 30*, 1 (2007), 4.

[23] Peixoto, T. P. The graph-tool python library. *figshare* (2014).

[24] Petroni Jr, N. L., and Hicks, M. Automated detection of persistent kernel control-flow attacks. In *Proceedings of the 14th ACM conference on Computer and communications security* (2007), ACM, pp. 103–115.

[25] Prakash, A., Venkataramani, E., Yin, H., and Lin, Z. Manipulating semantic values in kernel data structures: Attack assessments and implications. In *Dependable Systems and Networks (DSN), 2013 43rd Annual IEEE/IFIP International Conference on* (2013), IEEE, pp. 1–12.

[26] Rhee, J., Riley, R., Xu, D., and Jiang, X. Kernel malware analysis with un-tampered and temporal views of dynamic kernel memory. In *International Workshop on Recent Advances in Intrusion Detection* (2010), Springer, pp. 178–197.

[27] Riley, R., Jiang, X., and Xu, D. Multi-aspect profiling of kernel rootkit behavior. In *Proceedings of the 4th ACM European conference on Computer systems* (2009), ACM, pp. 47–60.

[28] Saltaformaggio, B., Gu, Z., Zhang, X., and Xu, D. Dscrete: Automatic rendering of forensic information from memory images via application logic reuse. In *USENIX Security Symposium* (2014), pp. 255–269.

[29] Schneider, C., Pfoh, J., and Eckert, C. Bridging the semantic gap through static code analysis. *Proceedings of EuroSec 12* (2012).

[30] Steensgaard, B. Points-to analysis in almost linear time. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (1996), ACM, pp. 32–41.

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

[32] Vasudevan, A., and Yerraballi, R. Stealth breakpoints. In *Computer security applications conference, 21st Annual* (2005), IEEE, pp. 10–pp.

[33] Walters, A. The volatility framework: Volatile memory artifact extraction utility framework, 2007.

[34] Wilson, R. P., and Lam, M. S. *Efficient context-sensitive pointer analysis for C programs*, vol. 30. ACM, 1995.

[35] Xuan, C., Copeland, J. A., and Beyah, R. A. Toward revealing kernel malware behavior in virtual execution environments. In *RAID* (2009), vol. 9, Springer, pp. 304–325.