A Study on the Evolution of Kernel Data Types Used in Memory Forensics and Their Dependency on Compilation Options

Andrea Oliveri, Nikola Nemes, Branislav Andjelic, Davide Balzarotti

Abstract

Over the years, memory forensics has emerged as a powerful analysis technique for uncovering security breaches that often evade detection. However, the differences in layouts used by the operating systems to organize data in memory can undermine its effectiveness. To overcome this problem, forensics tools rely on specialized "maps", the profiles, that describe the location and layout of kernel data types in volatile memory for each different OS. To avoid compromising the entire forensics analysis, it is crucial to meticulously select the profile to use, which is also tailored to the specific version of the OS.

In this work, for the first time, we conduct a longitudinal measurement study on kernel data types evolution across multiple kernel releases and its impact on memory forensics profiles. We analyze 2,298 Linux, macOS, and Windows Volatility 3 profiles from 2007 to 2024 to investigate patterns in data type changes across different OS releases, with a particular focus on types relevant to forensic analysis. This allowed the identification of fields commonly affected by modifications and, consequently, the Volatility plugins that are more vulnerable to these changes. In cases where an exact profile is unavailable, we propose guidelines for deciding on the most appropriate alternative profile to modify and use. Additionally, using a tool we developed, we analyze the source code of 77 Linux kernel versions to measure, for the first time, how the evolution of compile-time options influences kernel data types. Our findings show that even options unrelated to memory forensics can significantly alter data structure layouts and derived profiles, offering crucial insights for forensic analysts in navigating kernel configuration changes.

1. Introduction

Memory forensics enables analysts to reconstruct a system's state by analyzing volatile data from a computer's memory (RAM), providing critical insights that other forensic methods cannot achieve, including the list of running processes, active network connections, opened files, encryption keys, and traces of user activity that can reveal the presence of malware, unauthorized access, or other security breaches.

Unlike traditional storage media analysis, like hard drives and SSDs, memory forensics presents unique challenges due to the diverse ways operating systems (OSs) organize and store data in RAM. Each OS maintains system status information using data structures like linked lists, trees, and arrays, built with basic types such as C `struct`s or `union`s. However, these structures differ in how they organize the information, with fields located at different offsets and linked in various ways. Forensic tools such as Volatility [25] and Rekall [4] address this challenge by leveraging detailed models of kernel data types from widely used OSs like Windows, Linux, and macOS. These models, known as "profiles", serve as blueprints that describe the kernel data types and their locations in memory, enabling the post-mortem recovery of critical forensic information.

However, profiles are OS-specific and require regular updates to reflect the structural changes introduced in new OS versions. This challenge is exacerbated in open source operating systems like Linux, where users can extensively customize the kernel with various compile-time options. In such cases, a generic profile is insufficient; analysts must create and use a target-specific profile tailored to the kernel's specific configuration. It is important to note that the correct profile for a specific kernel under analysis is sometimes unavailable. This can occur either because the profile cannot be obtained or, in the case of Linux, because the compile-time configuration options are unknown, making it impossible to generate a specific profile for the machine.

A possible solution strategy that an analyst can use to overcome the problem is to derive information about kernel data types for a specific kernel version from profiles she believes are compatible with the system in question. However, this can lead to significant errors, as the layout of data structures may differ drastically between kernel versions. To better support this process, a quantification of how the data types used in memory forensics have evolved over time across major operating systems is needed. Furthermore, the impact of Linux compile-time options on the profiles used for forensic analysis, despite being well-known by the forensics community, has never been properly characterized or quantified. Finally, there are currently no guidelines that can help analysts in selecting a compatible profile that maximizes the amount of

Listing 1: Example of the different data types used by kernels to represent a process.

task_struct (Linux)

```
struct task_struct {
    #ifdef CONFIG_THREAD_INFO_IN_TASK
      struct thread_info thread_info;
    #endif
    unsigned int __state;
    void *stack;
    struct refcount_t usage;
    unsigned int flags;
    unsigned int ptrace;
    int on_cpu;
    ...
    struct list_head tasks;
    ...
    struct mm_struct *mm;
    ...
    struct list_head sibling;
    ...
}
```

proc and task (macOS)

```
struct proc {
    LIST p_list;
    struct proc *p_pptr;
    struct proc_ro *p_proc_ro;
    ...
}


struct task {
    struct lck_mtx_t lock;
    struct os_refcnt_t ref_count;
    bool active;
    ...
    _vm_map *map;
    ...
    queue_head_t threads;
    ...
}
```

_EPROCESS (Windows)

```
struct _EPROCESS {
    struct _KPROCESS Pcb;
    struct _EX_PUSH_LOCK ProcessLock;
    void *UniqueProcessId;
    LIST_ENTRY ActiveProcessLinks;
    struct _RUNDOWN_REF RundownProtect;
    unsigned long Flags2;
    long AddressSpaceInitialized;
    union _LARGE_INTEGER CreateTime;
    unsigned long ProcessQuotaUsage[2];
    unsigned long ProcessQuotaPeak[2];
    unsigned long PeakVirtualSize;
    ...
    LIST_ENTRY ThreadListHead;
    ...
    struct _RTL_AVL_TREE VadRoot;
    ...
}
```

forensic information retrieved from a target image.

Contribution – In this work, we perform, for the first time, a longitudinal measurement study of how kernel data structures evolved across different kernel releases and how these changes affect the profiles used by memory forensics tools. In a first set of experiments, we analyzed 2298 Volatility 3 profiles for different Linux, macOS, and Windows kernel versions released from 2007 to 2024. We looked at how the data types evolved across different kernel releases, with particular attention to the types used in memory forensics. This measurement allowed us to identify which Volatility plugins are the most vulnerable to changes in the data structures layout and to which extent a profile built for a wrong (but close) release of an OS can be used to recover forensic information, proposing a set of guidelines for selecting the most appropriate profile when the required one is unavailable.

Furthermore, for the first time in literature, we have characterized and quantified the impact of compile-time configuration changes in the Linux kernel. We have developed a tool to analyze the source code of 77 different Linux kernel versions, allowing us to study how the evolution of compile-time options in the source code affects data structures. Our study highlights that the introduction of new options, even those not typically associated with memory forensics, can significantly alter these data structures and has permitted us to identify which kernel options most affect forensic data types.

This theoretical study represents an initial step towards enhancing researchers' understanding of this phenomenon and also aims to support the development of future practical tools to address the issue of missing or unavailable profiles.

2. Background

The set of data types that represent the system's state is specific for each OS, and data type properties such as their sizes, fields' positions, and the pointers that connect them often change across the different releases of the same OS. An example of this diversity and variability is represented by how Linux, macOS, and Windows store information about running processes, as shown in Listing 1. All three operating systems use `C struct` types to represent process data, yet they employ different structure' layouts. Linux uses a huge `C struct` type called `task_struct` of approximately 10 KiB, Windows adopts an approach similar to Linux but uses a significantly smaller data type, the `_EPROCESS`. Finally, macOS divides this information into two distinct data types: `proc` and `task`[1]. It is also important to note that in Windows and macOS, a process acts as a container for threads, with each thread being represented by a distinct kernel data structure. In contrast, in Linux, each thread is represented in the kernel in a way that is indistinguishable from a process and using the same data structure.

The `task_struct` contains multiple fields, including, for example, a node of the doubly linked list of all the processes running on the system (highlighted in green), a pointer to a `mm_struct` type containing information about the address space of the process (in red) and another doubly linked list connecting to processes of the same group, as Linux represents threads as processes with the same task group ID (in blue). In contrast, threads in macOS are treated as separate entities, represented by a distinct data type maintained in a separate list. Furthermore, the data types used to maintain the linked list of running processes (in green) and information about the address space (in red) are completely different from the Linux ones. The same observation can be made for Microsoft Windows.

For Linux, another source of kernel data type variability derives from the possibility to recompile the kernel enabling or disabling `CONFIG_*` compile-time options that modify not only the kernel code but also the presence and the layout of the data types used by the kernel. At the source code level, these options are implemented as `#ifdef C` precompiler macros that selectively insert fields in the data structure types. An example of these options is present in Listing 1: if the `CONFIG_THREAD_INFO_IN_TASK` option (highlighted in brown) is enabled at compile time,

[1]This is a direct consequence of the fact that XNU, the macOS kernel, is a merge of two different kernels, the Mach kernel of NeXTSTEP and the FreeBSD one, and thus still maintains part of the data types and internal organization of both.

the structure `thread_info` is embedded in the `task_struct` as a field. This results in the presence of an additional field in the structure but also causes all subsequent fields to be shifted by an offset equal to the size of the embedded substructure. Hence, a profile compiled for a generic kernel without the option enabled fails to provide the correct offsets required to analyze a memory dump where the option was instead activated during compilation. An extreme case is provided by the `CONFIG_RANDSTRUCT` option. When enabled, it causes the compiler to randomly shuffle the memory layout of specific kernel data structures during compilation. This means that the offsets of the fields within these structures will vary from one build to another, forcing the analyst to create a specific profile not only for the version and configuration of the kernel but also for the specific build.

3. Method

In our research, we use two distinct methods to study the evolution of kernel data structures in different OSs and the impact of the reconfigurability of the Linux kernel: one based on available Volatility 3 profiles and one based on the analysis of the Linux kernel codebase.

3.1. Profile Based Analysis

Volatility 3 [25] uses a standardized JSON profile format that contains all the types whose definitions are available in debugging symbols of the kernel binary. This allows the tool to access type information, not only for open source OSs but also for closed ones for which only debugging symbols are available, like macOS[2] and Windows.

Our work takes advantage of this by creating a dataset of different Volatility 3 JSON profiles for the three OSs. This standardized format allows us to analyze the evolution of data structure over time and compare equivalent types across different operating systems. In particular, we collect statistics for each version of the three OSs by comparing the JSON profiles for consecutive versions.

3.2. Source Code Analysis

To quantify how the `CONFIG_*` compile-time options modify the layout of Linux kernel data types, we need to analyze the source code of the Linux kernel directly. However, existing static analysis techniques and tools assume that the source code has already gone through the preprocessor and, therefore, are unsuitable for studying the impact of `#ifdef CONFIG_*` directives. To overcome this limitation, we have developed a custom technique to extract, parse, and compare data structure types across different versions of Linux. The process is composed of three steps:

1) Data Structure Extraction – First, we extract the source code of the data structure types for each Linux kernel version. We use an index file of language objects generated by the *Universal Ctags* tool [10] to locate all the data structure types within the kernel code. The tool generates index files for data types and other language objects, such as variables and functions, allowing editors and other utilities to locate their positions quickly. For each data type, the tool extracts the structure definition, filters out comments and unnecessary `C macro`, and accounts for inner blocks of code. The resulting code snippets are stored in a single JSON file for each kernel version, alongside the name of the type and the paths of the source file from which they were extracted.

2) Data Structure Parsing – After the extraction of the raw data types, our tool parses them by using a syntactic metalanguage, the Backus-Naur Form (BNF) [16, 12] similar to the one used by compilers. A BNF language specification consists of a set of derivation rules (which can be recursive) that describe the structure of the target language. When a BNF specification is applied to a sample of the language it describes, it generates an ordered rooted tree that represents the syntactic structure of the sample known as a parsing or derivation tree.

We use the open source Python parsing toolkit `lark` [22] to generate parsing trees based on BNF specifications of `C structs` and `unions` definitions that we have devoloped. The parsing tree is then used to generate a structured JSON representation of the data structure, which facilitates semantic comparisons by organizing the data as a root tree object with associated fields and metadata. Each field can be a primitive type (e.g., pointers, integers, floats) or an embedded data structure (e.g., `C struct` or `C union`), and includes attributes such as its name, size, type (e.g., `int`, `char`), and the `#ifdef` expressions it depends on.

3) Data Structure Comparison – Comparison is performed between the JSON representations of consecutive kernel versions, matching each data structure type in one version with the corresponding structure in the next. If a data structure is present in only one of the two versions, it is classified as either newly added or deleted. For structures that appear in both versions, individual fields are compared, considering factors such as field type (e.g., `int`, array, pointer), field data type (e.g., `int`), the number of elements in arrays, and the `#ifdef` conditions affecting field presence.

In the rest of the paper, at the end of each section, in which we analyze various aspects of the Volatility 3 profiles dataset and the impact of reconfigurability on the Linux kernel, we include a *Takeaways* box. These boxes summarize the key findings and provide guidelines to assist

[2]The source code of XNU, the macOS kernel, is open source[2]. However, Apple has not released the kernel configurations and the complete buildchain for compilation. This means that builds obtained using third-party compilation scripts produce kernel binaries that may contain different structures' layouts compared to the official releases.

Table 1: Profile-based OSs dataset.

OS	Kernel versions		
	From/To	**Total**	**Total Changed/ Forensic Fields Offset Shift Only**
Linux	2.6.32-13 (Debian 6) 6.5.10-1 (Debian 12)	509	69.75% – 12.77%
macOS	10.6.3.10D573 (Snow Leopard) 14.3.23D60 (Sonoma)	195	63.64% – 15.90%
Windows	6.0.6000.16386 (Windows Vista) 10.0.22621.3235 (Windows 11)	1594	23.34% – 1.69%

forensic analysts in scenarios where the correct profile is unavailable, helping them select an alternative profile that can be manually adapted.

4. Profile-based Analysis

We begin our study by looking at the kernel data structures of the three most used operating systems, tracking their evolution between 2007 and 2024. In total, we analyzed 1594 releases of Microsoft Windows, 195 of macOS, and 509 Debian precompiled kernels. The dataset of Volatility 3 profiles was obtained by combining different online repositories [25, 11] with profiles we generated starting from debugging kernels (Linux) [3], kernel development kits (macOS) [1] or PDB files (Windows) [15]. We chose Debian kernels as representative compiled Linux kernels, as Debian is a widely used distribution for both desktop and server environments, with easily accessible kernel builds available online dating back to 2007. As shown in Table 1, despite having analyzed all publicly released macOS kernel versions for which it is possible to generate a profile, the dataset contains approximately 8 times fewer macOS kernel versions than Windows. This happens because Apple releases kernel development kits needed for profile generation only for some major and minor versions of the kernel and a few patch releases. It is also important to note that information about data types exported in debug symbols of proprietary kernels can be incomplete or represent only a subset of all the kernel data types due to the security-by-obscurity approach frequently used in proprietary code bases.

4.1. Evolution of Kernel Data Types

Figure 1 shows, for each release of the kernels, the total number of data structure types (in blue) and the percentage of them that were modified from the previous release (in green). These curves illustrate an interesting difference in the use and management of data types across the three operating systems. All three show a continuous increase in the number of types over time, although at different rates and with distinct patterns. For instance, recent releases of the Linux kernels contain over five times more data types than Windows and 12% more than macOS.

It is interesting to observe that macOS maintains a relatively constant number of types throughout each major release's lifespan, experiencing significant spikes only with the introduction of new OS releases and high ratios of data types changing at them. From Table 1 we can also note that approximately two-thirds of the kernel versions exhibit at least a kernel data type subjected to change, a ratio similar to what can be observed in Linux.

In contrast, Linux exhibits a more consistent growth ratio in the number of types. This phenomenon can also be observed in the percentage of modified data structures, which do not appear to follow any specific pattern. We can note, in particular, that Linux follows a similar trajectory to macOS until the 4.x major release. Beyond this point, the introduction and modifications of data types become erratic, no longer tied to major releases, and more akin to a rolling release development model.

On the other hand, Windows demonstrates remarkable stability in its data types, with only 23% of kernel releases affected by modifications to the data types' fields. Additionally, it displays distinct behavior regarding changes in data types: prior to the launch of Windows 10, most modifications to data types occurred alongside the introduction of a new major Windows release, similar to macOS. After the introduction of Windows 10, kernel versions with over 5% of modified data structures correspond instead to different Windows 10 marketing "codenames" releases. Interestingly, the introduction of Windows 11, despite being a major release, shows a similar percentage of new and modified structures as a minor Windows 10 release. This pattern is consistent with a shift to a rolling release development model that started with the introduction of Windows 10. Hence, from a forensic standpoint, Windows 10 and Windows 11 should be viewed as "umbrella names" encompassing a series of distinct operating systems akin to traditional major releases, each corresponding to various minor releases of the same underlying kernel.

Takeaways: Linux, macOS, and Windows all exhibit significant variability in kernel types, with Linux becoming more erratic after version 4.x and macOS primarily changing between major releases. Windows, which showed high variability only during major releases before Windows 10, shifted to a rolling release model with Windows 10 and 11, where each "codename" minor release displays variability comparable to major OS releases.

4.2. Evolution of Memory Forensics Types

We now focus on the evolution of those data structure types used in memory forensics analysis. These kernel data structures include fields that store forensically relevant information or pointers to other structures that

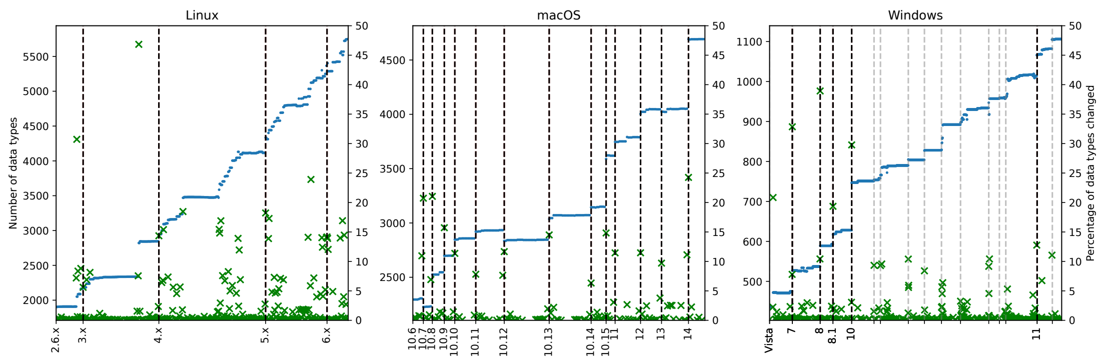


Figure 1: Blue dots: number of data types. Green crosses: percentage of data types with at least a change. Black dashed lines: major revisions. Grey dashed lines: Windows 10 editions. Starting from 10.15, Apple changes major release numbering format.

Table 2: Percentage of the total number of changes in forensics fields divided by kind and change class for the entire lifetime of Linux, macOS and Windows.

	Linux				**macOS**				**Windows**			
Field Kind	**Added**	**Removed**	**Changes**		**Added**	**Removed**	**Changes**		**Added**	**Removed**	**Changes**	
			Prop.	Offset			Prop.	Offset			Prop.	Offset
Data Field	4.27	4.41	-	30.10	1.20	1.50	0.30	35.74	0.72	2.15	-	11.47
Pointer	0.28	1.03	1.17	27.27	0.30	2.10	2.70	26.73	0.36	1.08	0.72	22.94
Embedded Struct	3.10	3.10	6.88	12.81	-	-	3.00	18.02	0.36	0.36	13.98	30.47
Total	7.65	8.54	8.05	70.18	1.50	3.60	6.00	80.49	1.44	3.59	14.70	64.88

do. For instance, the `task_struct` type in Linux contains the `comm` field, which stores the process name, and the `real_cred` pointer, which points to a data structure containing the process's privilege information. This information is essential for exploring relevant structures or extracting forensically relevant data using analysis tools like Volatility, which specifically utilize these fields.

In our study, we manually reviewed the plugin set shipped with Volatility 2 to identify data types used by the tool to extract artifacts. We chose to focus on Volatility 2 plugins for the study rather than Volatility 3 plugins, as the latter are significantly fewer and lack many features available in the former. This limitation would reduce the number of data structures analyzed, ultimately affecting the study's validity. We have identified 29 data structures for Linux, 36 for macOS, and 44 for Windows, encompassing 455 forensically relevant fields. These fields directly contain data or serve as pointers linking the forensic data types. The complete types and fields for each OS can be found in the `README` file of our tool used for to perform the analysis [18].

4.2.1. Global Statistics

A total of 171,960 unique modifications to fields of kernel types were identified in the profiles in our dataset between 2007 and 2024. Of these modifications, 2064 were to forensic-relevant fields, comprising 1452 for Linux, 333 for macOS, and 279 for Windows. We define a modification as the insertion or deletion of a field, a change in its kind (e.g., a data field becoming a pointer), a change in the field type (e.g., for a data field from `int` to `float` or changing in pointed type for pointers), a change in its total size (e.g., an increase in the size of an embedded array, struct, or union), and a change in the offset of the field within the parent structure. For atomic `C` types like `int`, pointers, `char`, etc., we only consider a change in type if the new (pointed) type has a different size than the previous one. This is because `typedefs` are often used as syntactic sugar to better describe types, but they do not introduce any semantic difference from the original type they are derived from. Furthermore, it is important to note that a change in one field can create a cascade of changes in subsequent fields, each of which is considered a distinct modification in our approach. For example, suppose a new field is added to a type. In that case, we record a modification for the new field's introduction and additional modifications for each subsequent field whose offset changes due to the shift caused by the introduction of the new field. Table 2 breaks down these changes into categories based on what was updated. We report the percentage of changes among forensic fields specifically for three kinds: data fields, pointers, and embedded structures. The other field kinds, such as embedded arrays, unions, and bitfields, are not included due to their marginal contribution. Changes are divided into three categories: adding, removing and changing of a field, the last one divided in changing of an internal property (kind, type, or size), and changing in the field's offset. Finally, we provide a summary line with the totals for each column for the three major field types. Combinations with no occurrences are marked with a "-".

It can be observed that for all three operating systems, most of the recorded changes pertain to the offset changes of fields (in grey in the Total row). This occurs because, as explained before, a change in the size of a field or its addition/removal causes a shift in all subsequent fields.

Interestingly, while the primary contribution to this type of change for Linux and macOS comes from data fields kind, for Windows, it mainly comes from embedded structures. Additionally, in Windows, a significant 14% of the changes, compared to the 7% of Linux, are due to embedded structures changing internal properties, particularly their size. For Linux, notable contributions (around 3%) stem from the changes in the type of embedded structures and the addition and removal of base kind fields.

We have also measured that in most cases in Linux (93%), new fields are added within the structure itself rather than at the end. Lower percentages are observed in macOS (78%) and Windows (62%). This suggests that adding a field to a forensics-related structure–whether or not it is directly related to a forensic-relevant functionality–will likely cause a shift in many other fields.

Takeaways: The majority of changes in structure types stem from alterations in the offsets of fields, with embedded structures (Windows) and pointers fields (Linux and macOS) being the most affected.

4.2.2. Offsets Modification

Now, we focus only on field offset shifts because they represent, as demonstrated in the previous section, the majority of changes in fields. As shown in Table 1, only 16% of the macOS kernel version and less than 2% of the Windows ones introduce changes in the offset of essential fields in forensics data types. While this is undoubtedly positive, it is crucial to examine how these alterations have been implemented across the historical development of the three operating systems. In order to respond to this question, Figure 1 illustrates the proportion of data types with at least one forensic-related field with a modified offset across a range of kernel versions. The release of major kernels is indicated by a blue cross, that of minor releases by a green triangle, and that of patch releases by a red dot. From the graph, it is evident that for macOS the majority of modifications occur during the transition between two major releases (73%). Conversely, for Windows, most changes take place during patch releases (51%) and major releases (40.5%), while in Linux, they predominantly occur during minor releases (69.3%).

Upon closer analysis of the plot, additional patterns emerge that could be crucial in a real forensic analysis when a profile is not available[3]. In that case, the analyst needs two pieces of information that are included in the unavailable profile: the location of kernel global variables and the offsets of forensics-related fields in data types. We will discuss the evolution of kernel global variables and the ways an analyst can extract them if the correct profile is unavailable in Section 4.2.5.Regarding the offsets of data structures, we can observe that for macOS, after several major releases, which consistently alter the layout of data types, other important changes happen only in minor ones. This pattern can be seen with macOS versions 10.8, 10.10, 11, 12, 13, and 14. This suggests that if the correct kernel version profile is unavailable, a profile from the major release or a subsequent minor release can likely be used successfully, completely ignoring the patch ones. For Windows 10 instead, starting from the fourth edition, a potentially effective strategy, when the necessary profile is not accessible, is to use offsets from a profile from the same Windows edition but with the highest possible patch version. Finally, in Linux, changes mainly occur at minor releases only. Therefore, one possible strategy is to use offsets taken from a profile from the nearest lower minor release.

[3]The problem of lack of profile in Linux is known to the forensics community and discussed in literature [20, 9]. However, macOS and Windows might also suffer from similar problems. For instance, Apple does not release the Kernel Development Kit for all kernel versions or does so with a significant delay, and sometimes the PDBs released by Microsoft are corrupted.

Table 3: TOP5 forensics types per number of modifications.

Linux		macOS		Windows	
Type	Count	Type	Count	Type	Count
task_struct	398	proc	85	_EPROCESS	118
mm_struct	254	task	58	_CMHIVE	35
inet_sock	130	thread	58	_ETHREAD	35
vm_area_struct	55	inpcb	24	_HHIVE	11
sock	44	vnode	16	_OBJECT_TYPE	7

Takeaways: The three OSs exhibit distinct patterns in how forensic-related data type changes occur, enabling analysts to mitigate the absence of a profile. For macOS, profiles from the nearest major release could be used, while for Windows, the highest patch version of the same edition can be effective, and for Linux, the nearest lower minor release provides can be a reliable alternative.

4.2.3. Most Affected Fields

To conclude this part, we examine which structures and fields are most affected by offset changes throughout the evolution of the three operating systems. As shown in Table 3, the structure most impacted across all three OSs is the one containing process information. For Linux and macOS, the next most affected structures are related to memory and network information, while for Windows, it is related to the Registry.

Table 4 highlights the specific fields with the most offset changes, once again revealing a clear predominance of fields related to process information structures. Specifically, in Linux, the field that changed the most is the one containing the process name, followed by the pointer to the process's credentials and several other fields like `children`, `sibling`, and `parent`, which are crucial for reconstructing the process hierarchy. A similar pattern is observed in macOS, where the most affected fields are within the two structures containing process information. In Windows, the top fields are those related to important process flags (`CrossThreadFlags`), the process's address space (`VadRoot`), and thread names. Additionally, also fields related to Registry keys are often impacted.

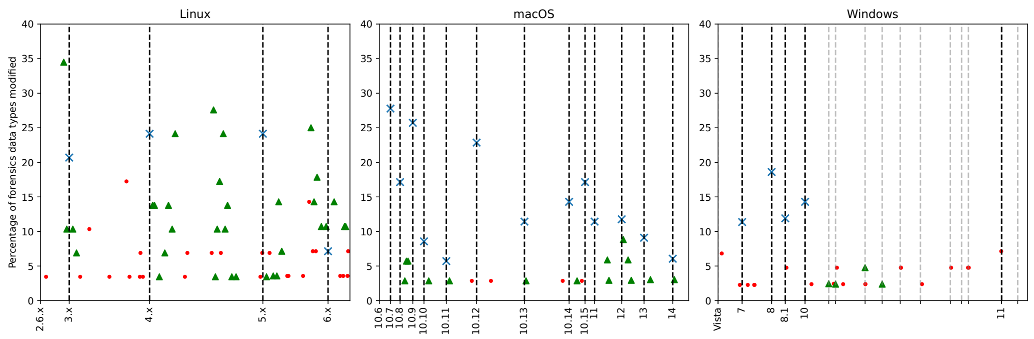


Figure 2: Percentage of forensics data types with at least a change in field's offset per kernel version. Blue crosses: major releases. Green triangle: minor releases. Red dots: patch releases.

Table 4: TOP10 forensics fields per number of modifications.

Linux			macOS			Windows		
Type	**Field**	**Count**	**Type**	**Field**	**Count**	**Type**	**Field**	**Count**
`task_struct`	`comm`	33	`thread`	`thread_id`	18	`_ETHREAD`	`CrossThreadFlags`	14
	`real_cred`	33		`threads`	17	`_EPROCESS`	`VadRoot`	14
`mm_struct`	`exe_file`	28	`task`	`all_image_info_addr`	15		`ExitTime`	13
`task_struct`	`children`	27		`all_image_info_size`	15	`_ETHREAD`	`StartAddress`	13
	`group_leader`	27		`bsd_info`	14	`_CMHIVE`	`FileFullPath`	9
	`parent`	27	`proc`	`p_comm`	12		`FileUserName`	9
	`pid`	27		`p_name`	12		`HiveRootPath`	9
	`sibling`	27		`p_argc`	11	`_EPROCESS`	`ActiveThreads`	8
	`stack_canary`	27		`p_argslen`	11	`_ETHREAD`	`ThreadName`	8
	`tgid`	27		`p_fd`	10	`_HHIVE`	`Storage`	8

Takeaways: Structures that contain information about processes are the most affected by offset changes in their field in particular process relations, virtual address space and flags. Unfortunately, these are also the most commonly used data structures in forensic analysis.

4.2.4. Volatility Plugins Affected

With the information now available, we can identify which Volatility plugins are most likely to encounter problems in extracting forensic information due to incorrect offsets in data types. As shown in Table 4 and discussed in the previous section, the most affected fields across all three operating systems are those containing data or pointers related to types used to represent processes. Plugins such as `pslist`, `pstree`, `psaux`, `ps_env`, and `threads`, which list the processes and threads present in the system by traversing pointers that link these types—like `children` and `sibling` in Linux and `threads` in macOS—become entirely nonfunctional. Moreover, other plugins rely on these to locate processes for further analysis. Consequently, if one of these plugins breaks, the dependent plugins will also fail.

Examples of plugins broken due to the dependency on process listing plugins include those that dump the memory of a process, such as `dump_map`, `elf`, and `procdump`, those that list and dump linked libraries, like `librarylist` and `librarydump`, or those that locate a process by name, such as the macOS plugins `bash` and `calendar`, which analyze and extract information from the `bash` process and the system calendar by using their names contained in the `p_comm` field of the `proc` structure to identify them among the other processes.

On Windows, problems can also arise with plugins that list and dump the address space of processes, like `vadinfo`, due to the usage of `VadRoot` field, which contains the root of the tree representing a process's address space and is commonly subjected to offset changing [6, 26]. Another series of plugins that can be significantly affected by incorrect offsets are those related to extracting the Windows Registry keys from memory like `hivelist` and `dumpregister` that rely on the `FileFullPath` field in `_CMHIVE`. Furthermore, missing offset for the data field `ExitTime` drastically reduces the possibility of the `timeliner` plugins creating a correct timeline from the various artifacts in memory.

Additionally, plugins that locate files and sockets opened by a process, such as `lsof`, `listraw` and `p_fs`, may fail entirely, for example, in macOS due to issues with the `p_fd` field that points to the list of file descriptors opened.

The lack of correct offsets for specific pointers can sometimes be circumvented using plugins that carve types based on specific memory signatures, like pool tags in Windows. These plugins, such as `psscan`, allow at least the identification of those data types in memory. However, if the offsets of other important fields, such as `VadRoot`, are not known, the analyst is forced to locate them manually.

Takeaways: Incorrect offsets in data types can affect essential Volatility plugins used for extracting process information, cached files, and Registry data. However, in some cases, especially with Windows, memory carving techniques can serve as a workaround to identify data types without relying on pointer offsets.

4.2.5. Kernel Global Variables

Profiles also contain the locations of kernel global variables that point to or store important data structures used by Volatility to setup and start the analysis. These variables can change their location in different kernel versions due to introducing or removing code and data parts in the kernel binary. Generally, these variables are exported as global symbols in ELF, Mach-O, or PE kernel files, allowing them to be retrieved through static analysis tools if a valid profile is unavailable. If direct extraction is not feasible, an analyst might use symbol location information from a profile that closely matches the one being analyzed.

To set up an analysis, the minimum set of required symbols includes those used by Volatility to initialize the virtual-to-physical address translation system and pointers to the list of active processes and loaded kernel modules. To determine if offsets from a partially matching profile can be adapted, the graph in Appendix displays the maximum absolute shifts in the offsets of these three key kernel variables between adjacent kernel versions for the three OSs. The variables analyzed are detailed in the graph's caption.

Significant differences among the three OSs are evident: these variables change location much more frequently in Windows (44%) compared to Linux and macOS (14% and 21%, respectively), corroborating the findings in [5]. In macOS, these changes usually coincide with new minor releases (indicated by green triangles in the graph), while in Linux and Windows, they are associated with patch releases (red dots). Moreover, in 76% of Windows cases, the maximum shift of these variables is smaller than the physical page size (represented by the grey dashed line in the graph), in particular starting from the last minor release of Windows 10. In contrast, in Linux and macOS, the shift is typically larger—averaging 1.2MiB for 94% of Linux kernels and 748KiB for 87% of macOS versions.

Takeaways: If kernel variable offsets are missing in Linux and macOS, it is advisable to use offsets from the nearest patch version profile for Linux and the closest minor release for macOS, due to the infrequent but significant and unpredictable changes in these OSs. For Windows, however, use the nearest patch release and consider brute-forcing offsets within a ±4KiB range of the original.

5. `CONFIG_*` Influence on Linux Kernel Profiles

In this Section, we examine how the `CONFIG_*` options modify the layout of data structures in the Linux kernel. Using the technique described in Section 3.2, we analyze the source code of 77 different minor releases of the Linux kernel source code, from version 2.6.32 up to 6.7, successfully parsing 96.6% of the extracted structures. The remaining 3.4% types are data structure types that contain very complex `C macro` embedded in their definition that cannot be easily resolved without implementing a complete `C` preprocessor and setting some `CONFIG_*` to a fixed value. However, we emphasize that none of the analyzed forensic types fall into this category.

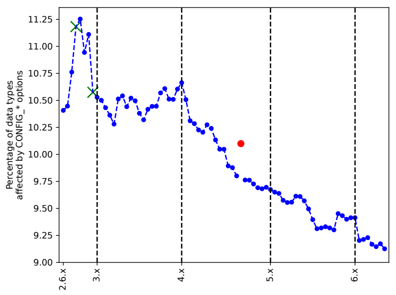


Figure 3: Percentage of Linux kernel data types affected by `CONFIG_*` options. Green crosses and red dot: versions affected by cascade effect. See Subsection 5.1 for more details.

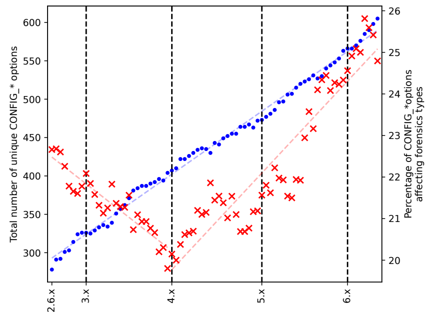


Figure 4: Blue dots: total number of unique `CONFIG_*` options in Linux kernel. Red crosses: percentage of `CONFIG_*` options affecting forensics types. Dashed lines: linear fit to highlight the different trends.

5.1. Cascade Impact of `CONFIG_*` on Data Types

Figure 3 shows the percentage of kernel data types affected by at least one `CONFIG_*` option across the different kernel versions. It is interesting to observe that for certain versions, represented with green crosses and a red dot, there are sudden modifications in the number of affected types that can be directly attributed to the introduction or removal of a *single* configuration option.

In fact, a `CONFIG_*` option can affect a field of a structure that is then embedded in many others through a chain of inclusions. In this case, all of them will be indirectly modified. For instance, in versions 2.6.35 and 2.6.39 (depicted with green crosses), the introduction and subsequent removal of the `CONFIG_OF` option in the `struct device` introduced a chain dependency in 157 other kernel types. A more striking example is the introduction and immediate subsequent removal of the `CONFIG_LOCKDEP_CROSSRELEASE` option [17] in version 4.14 (depicted in red in Figure 3). This option was introduced in `struct lockdep_map`, a structure representing cross-locking mutexes essential for the proper functioning of the OS, but not containing any data relevant to forensic analysis. However, because it is embedded in many other types, it impacted a total of 1451 data types, including 12 forensic data types such as `task_struct`, `module`, and `inode`.

Takeaways: The kernel contains many chains of nested data structures. As a result, introducing even a single new option in a peripheral non-forensics type can, in some cases, trigger cascading changes in many others, including those used in memory forensics.

5.2. Evolution of the Impact of `CONFIG_*` Options

Figure 3 shows a consistent decline in the percentage of types affected by `CONFIG_*` options since the 4.x series. This could suggest that, globally, types are becoming less dependent on compile-time configurations, thus resulting in a more predictable behavior. However, as depicted in Figure 4 (blue dots), the total number of `CONFIG_*` options impacting data types continues to increase linearly, expanding the potential combinations and resulting layouts for various types. Additionally, examining the trend of the percentage of the options influencing forensic types (red crosses), it is possible to distinguish two contrasting behaviors: prior to the 4.0 release, the percentage decreased, whereas from this release onwards, the number steadily increases over time. Consequently, starting from version 4.0, forensic types may experience a growing number of possible combinations for field offsets, resulting in less predictable profile information.

Takeaways: Starting from release 4.0, the influence of `CONFIG_*` options on forensics type increased despite an overall reduction in the number of affected kernel data types.

5.3. Influence of Particular Options

To better understand how options influence the layout of forensic types, we counted how many fields of forensic types each group of options influences (groups are defined by the first branch of the `Kconfig` tree that defines each option). Table 5 reports the top ten groups and the total number of forensic fields they influence across all kernel versions. The "Kernel Hacking" group, which includes options related to kernel development, testing, and tuning, is the primary source of variability among the fields of forensic structures, with an impact that is more than 6.2 times higher than that of the next group. While most of these options are typically disabled in the kernels shipped by major Linux distributions, they can be enabled in customized embedded kernels. However, one of the most influential options in forensic structures, `CONFIG_LOCKDEP`, is part of the "Kernel Hacking" group and is enabled in kernels across major Linux distributions. This indicates that while kernel hacking options are primarily intended for kernel developers, they can also significantly enhance diagnostic capabilities for end users. Other significant contributors to the variability of forensic structures include options related to CGroups, General Setup configurations (such as initramfs support), Security settings, File Systems configurations (like `CONFIG_FS_ENCRYPTION` for disk encryption) and Memory Management ones.

Table 5: TOP10 options affecting fields offset by subsystem.

`CONFIG_*` **Option Group**	**Affected Fields**
Kernel Hacking	4095
CGroups	658
General Setup	647
Security	613
File Systems	575
Memory Management	555
Processor Features	423
Network	357
CPU Configuration	350
General Options	339

Table 6: TOP10 non-debugging options influencing fields offset.

`CONFIG_*` **Option**	**Group**	**Affected Fields**
`CONFIG_SECURITY`	Security	344
`CONFIG_NUMA`	CPU Features	170
`CONFIG_COMPAT`	Binary Emulations	154
`CONFIG_MEMCG`	Cgroups	153
`CONFIG_TIMER_STATS`	Kernel Statistics	146
`CONFIG_IPV6`	Networking	133
`CONFIG_NUMA_BALANCING`	Memory Management	126
`CONFIG_FSNOTIFY`	File Systems	112
`CONFIG_KEYS`	Security	102
`CONFIG_LIVEPATCH`	CPU Features	86

To better understand the significance of individual non-kernel hacking options, we have reported in Table 6 the top ten that, individually, have the most significant impact on forensic structure fields. It is possible to note that they belong to a considerable variability of groups that control different aspects of the kernel behavior. Of particular importance are options such as `CONFIG_SECURITY`, which enables support for APIs used by Linux security modules like AppArmor and SELinux, `CONFIG_NUMA` which control the support to NUMA nodes, `CONFIG_COMPAT`, which enables the support for 32-bit binaries on 64-bit compiled kernels, `CONFIG_MEMCG`, which enables Cgroup memory support and `CONFIG_IPV6`, which enables support for the IPv6 network stack. Of these, at least three, `CONFIG_SECURITY`, `CONFIG_MEMCG` and `CONFIG_IPV6`, are generally enabled in precompiled kernels shipped with main distributions for desktop and server environments but are seldom activated in IoT devices while one, `CONFIG_NUMA` is always disabled due to the unsupports of this functionality by embedded CPUs. For instance, this aspect must be considered when contemplating using a profile derived from a kernel of a major distribution to replace an IoT one.

Takeaways: Our analysis highlights the substantial impact of "Kernel Hacking" group options on the layout of forensic types. Additionally, some kernel options that are typically enabled or disabled in precompiled kernels can introduce, if modified, variability in custom-recompiled kernels for specific uses (e.g. IoT), making generic profiles completely unsuitable for these devices.

6. Related Works

The research community has extensively examined the reliance of forensic tools on profiles, highlighting the challenge of creating universal profiles for Linux devices due to the significant variability in kernel configurations. In

2022, three independent studies [20, 9, 21] by *Pagani et al.*, *Franzen et al.* and *Qi et al.* proposed solutions to address the issue of missing profiles for Linux kernels with unknown build configurations, using code analysis directly to the kernel contained in the memory dump or using inference based on data structure invariants.

A notable study addressing the problem of profile reconstruction in Windows was conducted by *M. Cohen* [5]. In this work, the author examines the variability of offsets in four fields within the `_EPROCESS`, `_KPROCESS`, and `_TOKEN` kernel structures. Additionally, he explores the feasibility of determining these offsets through static analysis of the kernel code and highlights the significant variability of kernel global variable offsets across different patch versions.

Other researchers have explored methods that eliminate the need for profiles. For example, *Song et al.* [23] demonstrate how deep learning models can directly extract forensic information from memory dumps. Instead, *Dolan-Gavitt et al.* [7] use virtual machine introspection to observe OS runtime behavior and create signatures for identifying data structures in memory dumps, while *Urbina et al.* [24] gather data structure information by taking multiple memory snapshots of applications. Another approach leverages the kernel's source code, including techniques such as reconstructing data structures from memory dumps using source code signatures [14], determining data type offsets through static analysis of earlier kernel versions [8], or combining boolean constraints with probabilistic inference to match data structures without relying on profiles [13].

Recently, in *Oliveri et al.* [19] demonstrate the possibility of conducting forensic analysis even on completely unknown operating systems, without accessing to its source code or introspecting it but only leveraging the topological properties of data structures contained in a single memory dump completely removing the need for using profiles.

7. Conclusions

In this work, we have presented the first comprehensive study of the evolution of data structures within three of the most widely used operating systems, focusing on those crucial for memory forensics and, in the case of Linux, their dependence on kernel compilation options. Our analysis not only confirms but quantifies, for the first time, problems caused by changes in kernel data types and their impact on profiles that the memory forensics community faces daily.

We have demonstrated that most changes in forensic structure types are caused by shifts in field offsets due to the addition or removal of fields rather than by changes in the type or reorganization of fields—particularly affecting pointers. We have also quantified how even minor shifts in peripheral non-forensic types can create a domino effect, impacting many other structures, including those critical to forensics analysis. Our study has measured how much incorrect offsets in process-related data types can severely compromise the functionality of crucial Volatility plugins, impeding or even preventing the retrieval of forensic artifacts due to the interdependencies between these plugins. Potential solutions to this problem come from using multiple paths between kernel data structures, ensuring different ways to reach critical types, and enabling the extraction of forensic information also if not all the offsets in a profile are correct. Alternatively, especially for Windows, memory carving techniques can be employed to "blindly" identify forensic data types within a memory dump without relying on pointer-based linkages.

Our study also confirms that Linux represents the most significant challenge for analysts attempting to obtain valid profiles for uncommon or custom-configured kernels due to its unique compile-time reconfigurability and non-linear development model. From kernel version 4.0 onwards, we demonstrated that while the overall number of affected kernel data types decreases, the influence of `CONFIG_*` options on forensic types increases. By analyzing the impact of compile-time options on field offsets, we identified a set of options that significantly alter forensic structure layouts. These options are frequently enabled or disabled in custom-configured kernels, making applying a generic profile to such Linux kernels difficult.

Moreover, by revealing and measuring distinct patterns of type modifications across Linux, macOS, and Windows, this work has also enabled us to develop, for the first time, strategies for selecting alternative profiles that are more likely to produce accurate results when the exact profile for a target system is unavailable. In conclusion, this work aims to precisely assess the issue, establishing a theoretical and practical foundation for tackling it. This will help the future development of methods to address the challenges posed by the absence of valid profiles, including modifying existing profiles for compatibility, developing new techniques for forensic evidence extraction without relying on profiles or generating profiles dynamically from memory dumps.

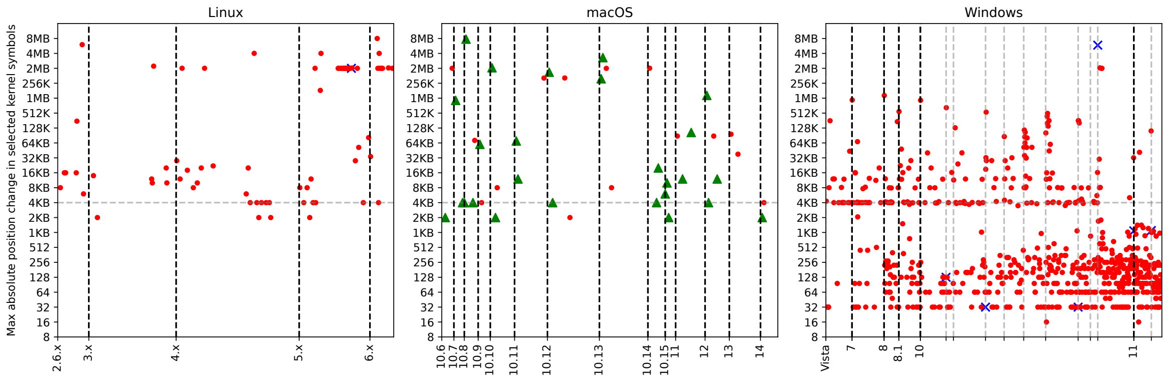
8. Code Availability

The profiles dataset and the tool used for the source code analysis are available as an open source project [18].

9. Acknowledgment

This work benefited from a government grant managed by the French National Research Agency with reference ANR-22-PECY-0007 and by the Ministry of Science, Technological Development and Innovation of Serbia (Contract No° 451-03-65/2024-03/200156) and the Faculty of Technical Sciences, University of Novi Sad, through project “Scientific and Artistic Research Work of Researchers in Teaching and Associate Positions at the Faculty of Technical Sciences, University of Novi Sad” (No° 01-3394/1).

Appendix



Maximum absolute change in the position of selected symbols in the kernel between two subsequent releases. Blue crosses: major releases. Green triangle: minor releases. Red dots: patch releases. Grey horizontal line: page size. Tracked variables: Linux:(`init_level4_pgt`, `init_task`, `modules`), macOS:(`IdlePML4`, `allproc`, `kmod`), Windows:(-,`PsActiveProcessHead`, `PsLoadedModuleList`)

References

[1] Apple, 2024a. macOS Kernel Development Kits. URL: `https://developer.apple.com/download`.

[2] Apple, 2024b. XNU source code. URL: `https://github.com/apple-oss-distributions/xnu`.

[3] Authors, V., 2024. Debian software repository. URL: `https://archive.debian.org/`.

[4] Cohen, M., 2014. Rekall memory forensics framework. DFIR Prague URL: `https://github.com/google/rekall`.

[5] Cohen, M.I., 2015. Characterization of the windows kernel version variability for accurate memory analysis. Digital Investigation 12, S38–S49. URL: `https://www.sciencedirect.com/science/article/pii/S1742287615000109`, doi:`https://doi.org/10.1016/j.diin.2015.01.009`. dFRWS 2015 Europe.

[6] Dolan-Gavitt, B., 2007. The vad tree: A process-eye view of physical memory. Digital Investigation 4, 62–64. URL: `https://www.sciencedirect.com/science/article/pii/S1742287607000503`, doi:`https://doi.org/10.1016/j.diin.2007.06.008`.

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

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

[9] Franzen, F., Holl, T., Andreas, M., Kirsch, J., Grossklags, J., 2022. Katana: Robust, automated, binary-only forensic analysis of linux memory snapshots, in: Proceedings of the 25th International Symposium on Research in Attacks, Intrusions and Defenses, Association for Computing Machinery, New York, NY, USA. p. 214–231. URL: `https://doi.org/10.1145/3545948.3545980`, doi:`10.1145/3545948.3545980`.

[10] Jelveh, R., 2014. Universal Ctags. URL: `https://ctags.io/`.

[11] JPCERTCC, 2024. Japanese CERTs coordination center Windows Profile collection. URL: `https://github.com/JPCERTCC/Windows-Symbol-Tables`.

[12] Knuth, D.E., 1964. backus normal form vs. backus naur form. Commun. ACM 7, 735–736. URL: `https://doi.org/10.1145/355588.365140`, doi:`10.1145/355588.365140`.

[13] Lin, Z., Rhee, J., Wu, C., Zhang, X., Xu, D., 2012. Dimsum: Discovering semantic data of interest from un-mappable memory with confidence, in: Proc. NDSS. URL: `https://www.cs.purdue.edu/homes/xyzhang/Comp/ndss12.pdf`.

[14] Lin, Z., Rhee, J., Zhang, X., Xu, D., Jiang, X., 2011. Siggraph: Brute force scanning of kernel data structure instances using graph-based signatures., in: Ndss. URL: `https://www.ndss-symposium.org/wp-content/uploads/2017/09/lin.pdf`.

[15] Maltsev, M., 2024. Winbindex - the Windows Binaries Index. URL: `https://winbindex.m417z.com/`.

[16] McCracken, D.D., Reilly, E.D., 2003. Backus-naur form (bnf), in: Encyclopedia of Computer Science, pp. 129–131.

[17] Molnar, I., 2017. Patch to remove `CONFIG_LOCKDEP_CROSSRELEASE` from Linux kernel. URL: `https://www.mail-archive.com/linux-kernel@vger.kernel.org/msg1562036.html`.

[18] Oliveri, A., 2025. Code and artifacts. URL: `https://github.com/eurecom-s3/structdiffing`.

[19] Oliveri, A., Dell'Amico, M., Balzarotti, D., 2023. An os-agnostic approach to memory forensics, in: NDSS 2023, Network and Distributed System Security Symposium, 27 February-3 March 2023, San Diego, CA, USA, Internet Society. URL: `https://www.ndss-symposium.org/wp-content/uploads/2023/02/ndss2023_s398_paper.pdf`.

[20] Pagani, F., Balzarotti, D., 2021. Autoprofile: Towards automated profile generation for memory analysis. ACM Transactions on Privacy and Security 25, 1–26. URL: `https://dl.acm.org/doi/pdf/10.1145/3485471`.

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

[22] Shinan, E., 2017. Lark. URL: `https://github.com/lark-parser/lark`.

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

[24] Urbina, D., Gu, Y., Caballero, J., Lin, Z., 2014. Sigpath: A memory graph based approach for program data introspection and modification, in: European Symposium on Research in Computer Security, Springer. pp. 237–256. URL: `https://link.springer.com/content/pdf/10.1007/978-3-319-11212-1.pdf`.

[25] Walker, A., 2017. Volatility framework: Volatile memory artifact extraction utility framework. URL: `https://volatilityfoundation.org/`.

[26] Yosifovich, P., Russinovich, M.E., Solomon, D.A., Ionescu, A., 2023. Windows Internals, Part 1. 7 ed., Microsoft Press, Redmond, WA.