LogicMem: Automatic Profile Generation for Binary-Only Memory Forensics via Logic Inference

Zhenxiao Qi, Yu Qu, and Heng Yin
University of California, Riverside
{zqi020, yuq, hengy}@ucr.edu

***Abstract*—Memory forensic tools rely on the knowledge of kernel symbols and kernel object layouts to retrieve digital evidence and artifacts from memory dumps. This knowledge is called profile. Existing solutions for profile generation are either inconvenient or inaccurate. In this paper, we propose a logic inference approach to automatically generating a profile directly from a memory dump. It leverages the invariants existing in kernel data structures across all kernel versions and configurations to precisely locate forensics-required fields in kernel objects. We have implemented a prototype named LogicMem and evaluated it on memory dumps collected from mainstream Linux distributions, customized Linux kernels with random configurations, and operating systems designed for Android smartphones and embedded devices. The evaluation results show that the proposed logic inference approach is well-suited for locating forensics-required fields and achieves 100% precision and recall for mainstream Linux distributions and 100% precision and 95% recall for customized kernels with random configurations. Moreover, we show that false negatives can be eliminated with improved logic rules. We also demonstrate that LogicMem can generate profiles when it is otherwise difficult (if not impossible) for existing approaches, and support memory forensics tasks such as rootkit detection.**

I. Introduction

Advances in memory forensics [13], [14], [37] have shown practical applications in a number of important fields, such as crime investigation, rootkit and malware detection, etc. Memory forensics is capable of revealing a substantial amount of memory-residential information such as encryption keys, network traffics, running processes, and in-memory malware, and analyzing volatile memory from various devices, including desktops, smartphones, embedded devices, etc.

In order to reveal evidence and artifacts from captured physical memory, complete understanding of kernel symbols and kernel data structures is required, which, in the context of memory forensics, is called a profile. To perform memory forensics on a memory dump, a profile of the target system must be generated in advance to the analysis. For Windows operating systems, generating a profile is less challenging. The layout of kernel objects can be retrieved from Microsoft symbol server [10], which contains pre-built kernel debug symbols. Moreover, Windows kernels have limited versions such that users can reuse profiles from symbol servers without needing to know product names, releases or build numbers. However, for Unix-based OSes like Linux, kernel versions upgrade frequently and kernels are highly configurable. Distributors and personal users can customize their own kernels by enabling or disabling various kernel configurations for new features or special hardware. Figure 1 shows some configurations (`#ifdef`) in `task_struct`. Though forensics-required fields are not controlled by configurations in general, their offsets are affected by configurations because enabling or disabling configurations result in adding or removing fields and pushing forensics-required fields to different offsets. While official distributions could potentially provide kernel symbols and debugging information when a new version is released, it is infeasible for user-customized Linux kernels, Android systems, and embedded devices, in which cases vendors do not release debug symbols and the layouts of kernel data structures are extremely diverse due to specific preferences such as tmpfs support, control groups, and security options (e.g. SELinux, AppArmor). In conclusion, it is impossible to have pre-built profiles that work for all Linux kernels.

Researchers have proposed solutions to tackle the Linux profile generation problem. The most popular approach adopted by memory forensics and virtual machine introspection tools (e.g., Volatility [13], LibVMI [6], and DECAF [27]) is extracting the profile from kernel debug symbols. However, this approach is not designed for binary-only memory forensics. First of all, it needs access to the live system and requires that kernel headers and toolchains be installed. Second, a test kernel module must be built or inserted to collect debug symbols, which is often not feasible for production environments. Moreover, human intervention is needed to update the test kernel module along with kernel upgrades, which is rather inconvenient for both developers and users.

Another approach attempts to reconstruct kernel objects for binary-only memory forensics by analyzing the assembly code of kernel functions [16], [22], [35], [36]. The intuition is, when an object member is accessed, a memory-dereference instruction is executed and the offset of this object member is revealed in that instructions. However, this approach can hardly be generalized because kernel functions can be compiled into various instruction formats by different compilers on different architectures. Moreover, it is difficult to precisely locate these offset-revealing memory dereferences from a large number of memory-dereference instructions in the kernel code.

In this paper, we present a logic inference approach that reasons about offsets of forensics-required fields in kernel objects in an accurate and efficient manner, without knowing the exact kernel version and the configuration. Our approach

Network and Distributed Systems Security (NDSS) Symposium 2022
24-28 April 2022, San Diego, CA, USA
ISBN 1-891562-74-6
https://dx.doi.org/10.14722/ndss.2022.24324
www.ndss-symposium.org

is motivated by the observation that ***only a small number of kernel object fields are required to perform memory forensics (forensics-required fields) and those fields exhibit invariants that remain unchanged regardless of kernel versions and build configurations.*** To support this observation, we conducted kernel evolution analysis over 47 major Linux versions. Note that the offsets of forensics-required fields may vary due to different kernel configurations [42] and versions [48]. Our approach considers such uncertainty and codifies version- and configuration-agnostic invariant patterns for forensics-required fields, allowing the inference engine to efficiently locate their offsets in kernel objects.

We have implemented a prototype LOGICMEM to demonstrate the efficacy of our proposed approach. We evaluate LOGICMEM on 16 memory dumps collected from the mainstream Linux distributions, e.g., CentOS, Debian, and Ubuntu, 30 memory dumps from customized Linux kernels with random configurations, six long-term support (LTS) or stable releases of recent Android kernels, and two kernel images designed for embedded devices. The evaluation results show that LOGICMEM can infer forensics-required fields with perfect accuracy for the mainstream Linux kernels and 100% precision and 95% recall for customized Linux kernels with random configurations. The false negatives can be further eliminated after a minor adjustment of the logic rules. In the case studies, we demonstrate that, while existing memory forensics tools are unable to analyze Android and embedded systems due to the absence of kernel symbols, LOGICMEM can still support eight Volatility plugins. Furthermore, we show that LOGICMEM can enable the rootkit detection plugin in Volatility to analyze a rootkit-infected memory dump.

We summarize our contributions as follows:

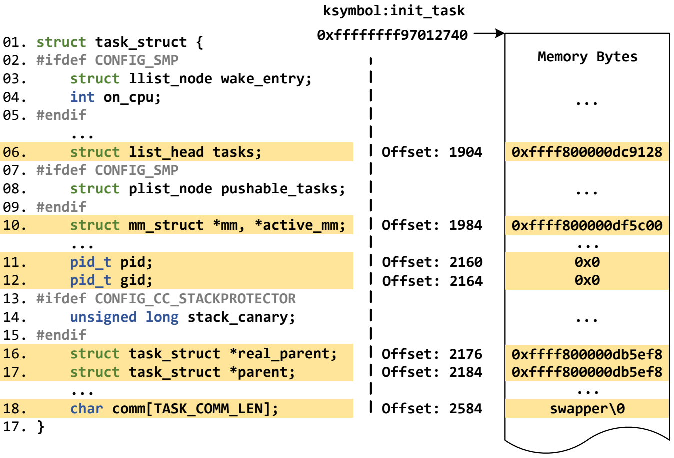
- We propose a logic inference approach to locating offsets of forensics-required fields in kernel data structures without any knowledge about kernel versions and configurations.
- We design and implement LOGICMEM to demonstrate the efficacy of our proposed approach, and evaluate LOGICMEM on 16 memory dumps collected from mainstream Linux distributions and 30 kernels built with random configurations. The evaluation results show that LOGICMEM can achieve 100% precision and at least 95% recall with reasonable efficiency.
- We demonstrate the ability of LOGICMEM to support some memory forensics tasks for memory dumps collected from Android systems and embedded devices, which is difficult (if not impossible) for existing approaches such as Volatility.

To facilitate further research, we make the source code and dataset publicly available[1].

II. BACKGROUND AND MOTIVATION

A. A Running Example

A profile contains the locations of kernel symbols and kernel object layouts that guide memory forensic tools to find a kernel object of interest and retrieve values from desired fields. Figure 1 shows an example of using a profile to retrieve the process information from `task_struct`. Firstly, a memory forensic tool obtains the virtual address of the first process from the `init_task` symbol. Then it translates it into the corresponding physical address and locates the `task_struct` object in the memory dump. Next, the offsets of the desired fields (highlighted in Figure 1) in `task_struct` are obtained from the profile. As presented in Figure 1, the big challenge is that Linux kernel profiles are configuration- and version-specific: Linux project upgrades frequently and contains various compile configurations, a profile generated for specific version and configurations is not likely to be reused by another kernel. Existing approaches [6], [13], [27] build a profile from kernel headers and configurations used to built this kernel. However, it requires access to the live system and installation of kernel headers and compiler toolchains, which is impossible for binary-only forensics such as crime scene investigation, cloud service, and production environment. Other approaches [16], [22], [35] proposed binary-only approaches to solve this problem, but, according to their papers, failed to achieve high precision and recall. In this paper, we propose a logic inference approach that has high generality and accuracy with reasonable efficiency.



```
struct task_struct {
#ifdef CONFIG_SMP
    struct llist_node wake_entry;
    int on_cpu;
#endif
    ...
    struct list_head tasks;
#ifdef CONFIG_SMP
    struct plist_node pushable_tasks;
#endif
    struct mm_struct *mm, *active_mm;
    ...
    pid_t pid;
    pid_t gid;
#ifdef CONFIG_CC_STACKPROTECTOR
    unsigned long stack_canary;
#endif
    struct task_struct *real_parent;
    struct task_struct *parent;
    ...
    char comm[TASK_COMM_LEN];
}
```

Fig. 1: An example of using a profile to retrieve information about running processes in Linux kernel.

B. Existing Techniques

a) Compiler-based Approach: Compiler-based profile generation is the most common approach adopted by memory forensics [13] and virtual machine introspection [6], [27] tools. It produces a profile by collecting kernel symbols from `System.map` file in a live system and compiling a kernel module to emit debugging information. Therefore, it is capable of generating a profile with 100% accuracy. However, this approach is by nature impractical for binary-only forensics. First, it requires access to the live system to collect the `System.map` file and compile a test kernel module. Moreover, it requires compiler toolchains and kernel headers to be installed in the target system, which is hardly the case for servers, IoT devices, and smartphones.

b) Assembly-based Approach: Assembly-based profile generation extracts kernel object layouts directly from the

[1] https://github.com/bitsecurerlab/LogicMem

memory dump. The insight is that when an object member is accessed, an offset-based memory deference is performed, and this offset is observable from assembly instructions, which are defined as Offset-Revealing Instructions (ORIs) [22]. Therefore, an assembly-based approach utilizes ORIs from binary to infer the offsets of object members. RAMPARSER [16] reconstructs kernel objects based on ORIs. RAMPARSER first identifies multiple kernel functions that operate on forensics-required fields. Then it obtains addresses of identified functions from `System.map` file and disassembles kernel functions from a memory dump. To locate ORIs that access forensics-required fields, it compares ORIs from multiple functions that access the same field, and performs an intersection to find ORIs using the same offset. Eventually, offsets of forensics-required fields are extracted from identified ORIs and used for memory forensics analysis. ORIGEN [22] is another work that recovers offsets of object fields in a similar fashion. ORIGEN can generate profiles for cross-version memory forensics. More specifically, it first identifies ORIs from one version of software whose profile is available. Then it performs Control Flow Graph (CFG) based code search to locate equivalent ORIs in an unseen version of the same software. Afterwards, it retrieves offsets of object fields from the located ORIs and generates a new profile. Another work proposed by Pagani [35] identifies ORIs by monitoring runtime memory access patterns. Similarly, it first finds kernel functions that operate on forensics-required fields in kernel objects and extracts them from a memory dump. Then it re-executes kernel functions using the Unicorn emulator [12]. Along the execution, it monitors memory footprints to reveal offsets of forensics-required fields. However, one major drawback of this work is heavy execution overhead: it takes hours to generate a profile for one memory dump.

This assembly-based approach faces several fundamental challenges. First of all, the same source-code-level statement can be compiled into instructions in various formats by different compilers and compiler options. Therefore, to support kernels built by different compilers on different architectures, it needs to exhaust all possibilities of instruction formats. Secondly, it is hard to precisely locate ORIs from kernel function code, especially for large functions where there are many memory access instructions.

C. Design Goals

We would like to support binary-only memory forensics with high accuracy and reasonable efficiency. More specifically, our design goals are as follows:

- **Binary-only Approach.** We assume the only input to our forensic analysis is a memory dump. A good solution should generate a profile directly from a given memory dump and support the existing memory forensic tasks.
- **High Generality.** The proposed technique should be general enough for different kernel versions, configurations, CPU architectures, and compilers.
- **High Accuracy.** Obviously, the generated profile should be accurate to ensure correct analysis results.
- **Reasonable Efficiency.** The proposed technique should generate a profile within a reasonable time frame and can deal with large memory dumps.

III. Kernel Evolution Analysis

The Linux kernel is a large collaborative project and it upgrades frequently. On average, 8.5 changes are accepted into the Linux kernel every hour [7]. Many researchers have studied the evolution of Linux kernel. As discovered by Godfrey et al [26], the kernel project size and lines of code (LOC) grow at a super-linear rate, but the size of header files grows at a sub-linear rate. A more recent study by Lotufo et al. [34] found that around 75% of changes in kernel evolution are at function level, including functionality additions and retiring, bug fixes, etc. These findings imply that only a small fraction of code changes happened to kernel object definitions. Moreover, Israeli et al. [29] found that `arch` and `drivers` directories, which account for about 60% of Linux codebase, grow at a similar rate with the whole kernel codebase. However, most of kernel objects required by memory forensics are defined in the `include` directory. Theses studies show that changes to kernel objects are a very small subset, compared with function-level changes. To further understand changes in kernel objects along Linux upgrades, we conduct our own evolution analysis on kernel objects in 47 major versions.

Previous works have shown that value invariants can be inferred during the execution of kernel and can be used to ensure kernel integrity [20]. In this work, we analyze the kernel object evolution at the source code level, and derive different types of invariants that remain unchanged along kernel upgrades. Based on invariants of forensics-required fields, we design logic rules for these fields, which are used by LogicMem to infer their offsets.

A. Invariant Detector

To automatically identify differences and invariants in kernel objects during kernel evolution, we develop an *invariant detector*. Particularly, it detects four types of changes, i.e., field addition, deletion, modification, and swap. Invariants are then derived from fields that do not involve these types of changes (referred to as unchanged fields). Field addition and deletion capture added fields in new versions and deleted fields from old versions. Field modification refers to changes of field types. Field swap captures the offset changes of the same field in different kernel versions.

The invariant detector relies on a simple text parser to scan and extract information from source code. It scans the object definitions in source code and summarizes likely object invariants over a large variety of kernel versions. For one kernel object, the invariant detector first tokenizes its field definitions and parses data types, compiler attributes, and field names. Afterward, it compares tokenized object definitions between every pair of two adjacent kernel versions and derives differences and invariants of the object in the two versions. There are syntax level changes in the source code that are removed for consideration. For instance, some fields in newer versions are associated with compiler attributes, while the types and field names are unchanged. Therefore, when tokenizing object fields, compiler attributes are removed. Type abstraction is another kind of syntax level changes, e.g., unsigned int is defined as u32. While the type name changed, the variable length is still the same. To ensure consistency, we expand the type tokens to their original types.

TABLE I: Linux kernel object evolution analysis over 47 kernel versions. The # of required fields shows the number of fields in this object that are required to perform memory forensics. The # of changed required fields shows the number of changed (added, deleted, modified or swapped) fields in required fields

Object Name	**task_struct**	**mm_struct**	**vm_area_struct**	**fs_struct**	**mount**	**dentry**	**module**	**fdtable**	**vfsmount**
Tot. # of added fields	146	26	7	0	11	3	24	1	0
Tot. # of deleted fields	66	20	6	0	6	2	21	1	0
Tot. # of modified fields	19	4	0	0	2	1	7	2	0
Tot. # of swapped fields	7	0	2	0	7	1	0	0	0
Avg. # of all fields	213	58	20	7	27	18	47	6	3
# of required fields	15	13	6	2	6	8	8	2	3
# of changed required fields	1	0	0	0	0	0	4	0	0
% of unchanged fields	97.6%	98.2%	98.5%	100%	97.9%	99.2%	97.7%	98.7%	100%

Overall, we analyzed 47 major versions of Linux kernel from version 3.3 to 5.5 and invariants were accumulated incrementally by comparing each pair of adjacent versions. Table I shows the analysis results for eight kernel objects, and the full analysis results can be found in §C. Table I lists the total number of changed fields over 47 kernel versions. Specifically, for each pair of adjacent versions, the invariant detector detects added, deleted, modified and swapped fields, then counts the total number of changed fields for 47 versions. Then we show the average number of all fields for each object across these 47 kernel versions. For each comparison between two adjacent versions, we calculate the percentage of unchanged fields and show the average percentage for 47 versions in the last row. According to the results, over 97% of kernel fields remain unchanged for each kernel upgrade. Based on the kernel object evolution analysis, we summarized three types of object invariants.

a) Type Invariants: Type invariants refer to fields that exist in all observed versions and do not involve in type changes. For example, in `task_struct`, the `pid` field always contains an integer, the `comm` field contains a process name, which is a string and `tasks` is a `list_head` pointer that points to next `task_struct` in the doubly linked list.

b) Order Invariants: Order invariants refer to fields whose relative orders are not changed over the observed versions. According to our analysis, sometimes fields are swapped or moved to another offset for better cache alignment, but order changes happen less frequently compared with other types of field changes. Order invariants of two fields consider configurable fields may exist in between, thus only indicate relative orders instead of exact distance.

c) Range Invariants: Along Linux kernel upgrades, new object fields are added and some old fields are removed, but the total number of object fields is within a reasonable range. For instance, `task_struct` has a relatively large number of field additions and deletions. Along 47 kernel upgrades, there are a total of 146 fields added to `task_struct` and 66 fields are removed. Compared with `task_struct`, changes of field numbers are much fewer in other data structures. For instance, `fs_struct` is not involved in any field addition or deletion over 47 kernel versions.

We focus on collecting invariants for forensics-required fields. In total, there are 220 unique forensics-required fields in 56 kernel objects(more details can be found in §VI). In Table I, the 7th row shows the number of forensics-required fields in each object, and the 8th row shows the number of changed fields in kernel upgrades among forensics-required fields. As presented in Table I, only a small subset of fields is required by Volatility and a few of them are changed during kernel evolution. For instance, there are a total of 238 field changes along 47 kernel upgrades in `task_struct`, but only 15 fields in `task_struct` are demanded by Volatility plugins, and only one (`real_start_time`) of them are affected by kernel upgrades. This field is used to retrieve process start time and was deleted in Linux 5.5. As presented in Table I and Table V, most forensics-required fields remain unchanged during kernel evolution. Therefore, we are able to summarize invariants for those fields. For changed forensics-required fields that do not exhibit invariants across all versions, we discuss the solution in §C. Admittedly, in a newly released kernel, some of these invariants might not hold. In this case, we can run the invariant detector on new kernel versions, and adjust our rules based on the new invariants. More details about handling invariant changes can be found in §VII.

In summary, although offsets of forensics-required fields can be shifted by changed fields due to kernel upgrade and configurable fields that are determined during compilation, numerous invariants existing in the forensics-required fields may help us find the offsets of these fields in a memory dump regardless of kernel versions or configurations.

IV. System Overview

Figure 2 illustrates the overview of LogicMem. It consists of three key components: a *kernel symbol detector*, a lightweight *fact collector*, and a *logic inference engine*. To infer forensics-required fields in a kernel object from memory bytes, LogicMem first recovers kernel symbols from the memory dump and looks up symbols to locate the target object. Forensics-required objects can be found in recovered kernel symbols since memory forensics tools also rely on those symbols to locate desired kernel objects. Then LogicMem translates the virtual address of a kernel object and locates its physical address in the memory dump. The fact collector is then invoked to scan memory bytes in the target object and lift them to low-level facts. These facts serve as a search space for the logic inference engine. Guided by the predefined logic rules, the logic inference engine infers forensics-required fields from the facts collected in the target memory region. Eventually, the recovered kernel symbols and offsets of forensics-required fields are integrated as a profile for downstream memory forensics tools.

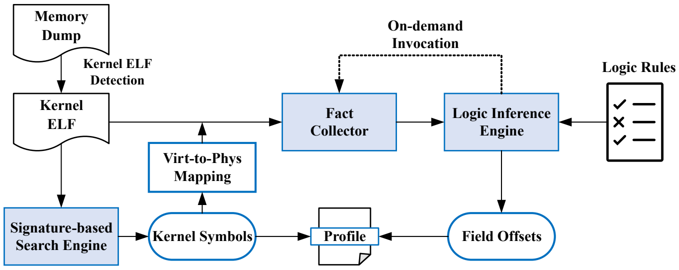


Fig. 2: The overview of LogicMem.

A. Kernel Symbol Table Recovery

Kernel symbols map the global functions and objects to their addresses in the kernel memory. Such information is essential for memory forensics to locate desired kernel objects. Normally, the kernel symbols are available from the `System.map` file in the live system. However, for binary-only memory forensics, where the live system is not available, recovering the kernel symbol table directly from the memory dump is the only feasible approach. To this end, we develop a signature-based search method to locate and recover kernel symbols from a given memory dump.

a) Kernel ELF detection: Kernel symbols reside in the kernel data section. To avoid the overhead of searching in a large memory space and eliminate false candidates outside kernel memory, LogicMem locates and extracts the kernel ELF. This is done by searching the unique string "swapper\0", which is the process name in `task_struct` for the very first process in Linux kernel. Then LogicMem searches for the ELF header around the unique string and extracts the whole kernel memory.

b) Kernel Symbol Recovery: Kallsyms is a mechanism in Linux kernel for resolving all symbols for debugging. Specifically, kallsyms extracts symbols from all sections in the kernel and stores them in relevant data structures. The layout of the related data structures are presented in Figure 3. As defined in </kernel/kallsyms.c>, `kallsyms_addresses` is an array that stores virtual addresses of kernel symbols. `kallsyms_num_syms` is an unsigned long number that represents the number of kernel symbols. To reduce memory usage, kernel symbol names are compressed and stored in three relevant data structures, namely, `kallsyms_names`, `kallsyms_token_index` and `kallsyms_token_table`. After kernel version 4.6, a new configuration was introduced to save the memory usage of symbol addresses by storing a relative base address and an array of offsets, which are named `kallsyms_relative_base` and `kallsyms_offsets` respectively. We closely inspect values stored in the aforementioned data structures and identify unique patterns for each of them. Details about data structure patterns can be found in §B in Appendix.

B. Virtual-to-Physical Address Translation

Virtual-to-physical address translation is an indispensable component for memory forensics. When locating a global object using its virtual address from the kernel symbol table, or following a pointer in one object to retrieve information from another object, a virtual-to-physical address translation is needed. To perform kernel address translation, the physical address of the kernel page table is required, which is exported as the symbol `init_level4_pgt` (on x86 arch with 4-level page tables) or `init_top_pgt`, and can be recovered using the above step. Normally, we can leverage the fact that kernel memory is contiguously mapped, and obtain the physical address of the kernel page table by subtracting a fixed offset from its virtual address. However, when KASLR [4] is enabled, that fixed offset is randomized by an unknown KASLR shift. Volatility [13] requires a valid profile to recognize the shift between kernel virtual addresses and physical addresses. More specifically, it retrieves the virtual address of `init_task` from the profile, and obtains the offset of `comm` fields, which is the process name. Then it adds them up to get the virtual address of the process name for `init_task`, which is known as "swapper\0". Next, it searches for the string "swapper\0" in the memory to locate its physical address and calculates the shift between its virtual address and physical address. However, a valid profile is not always available in advance to analyze a memory dump.

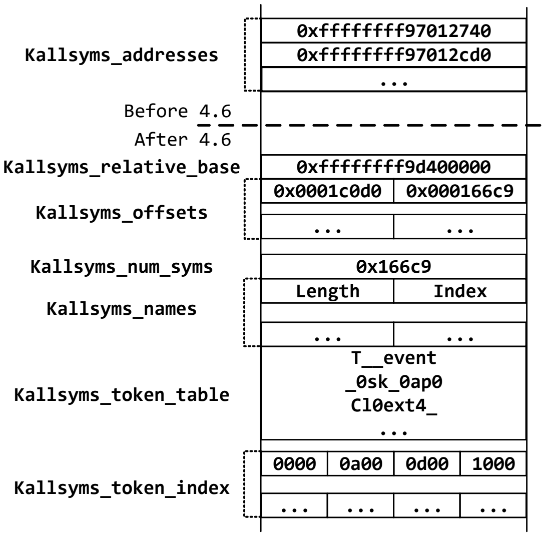


Fig. 3: Kallsyms data structures layout in memory.

Inspired by Volatility [13] and approaches proposed by Pagani [35] and Zhang et al. [47], we leverage the kernel symbol table recovered in the previous step to identify the KASLR shift. Specifically, we utilize the fact that some strings are exported as global symbols with the prefix of `__kstrtab_` in `__kstrtab` section. We choose a string in `__kstrtab` section, e.g., `kallsyms_on_each_symbol`, and search it in the memory dump to locate its physical address. Then we calculate the shift by subtracting the physical address from the virtual address obtained in recovered symbols (`__kstrtab_kallsyms_on_each_symbol`). The kernel ELF detection step also avoids the overhead of searching for symbols from the whole memory in this step.

C. Fact Collector

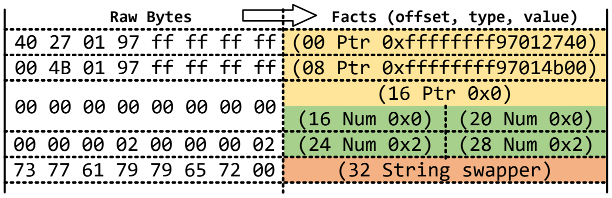
A memory dump is captured from the RAM of a running system, which is a collection of volatile memory bytes. LogicMem employs the fact collector to lift memory bytes into low-level facts. The fact collector is invoked on demand by LogicMem to scan one page of memory bytes, instead of the whole memory dump, as we found one page is enough to include bytes in the target object. More specifically, the fact collector scans from the base address of an object, decodes the value at each aligned address, and makes an educated prediction about its type. Compilers always place fields at even-byte alignment unless the data structure is annotated with *__attribute__((packed))*. According to our kernel evolution analysis, this attribute does not affect forensics-required objects, which is also substantiated by our evaluation results II. Therefore, it is safe to collect facts at aligned addresses. Handling unaligned fields is straightforward because we only need to modify the fact collector to also label unaligned values. The fact collector conservatively predicts three kinds of primitive data types: pointers, strings, and numbers (shorts, integers or longs). To facilitate the following discussions, we assume a 64-bit architecture when introducing the heuristics used by the fact collector to predict data types. The heuristics can be easily adapted to support 32-bit architectures.

a) Pointer: To collect pointer type fields, the factor collector scans the given memory region at an 8-byte granularity. Then it performs a virtual-to-physical address translation with the obtained 8-byte value. If it can be translated to a valid physical address, then this field is considered a pointer.

b) Strings: Strings are stored in `char` arrays with different sizes. In Linux kernel data structures, strings usually represent names, e.g., process name and module name. Therefore, strings contain printable ASCII symbols and end with a NULL terminator. The fact collector scans the memory bytes following the above heuristics to collect string fields.

c) Numbers: If the value is not a pointer, then the fact collector checks whether it is a possible number. Specifically, the fact collector predicts three types of numbers, short, integer, and long, each of them takes 2 bytes, 4 bytes, and 8 bytes respectively on 64-bit architectures.

A value may be predicted to have several types. For instance, a small number may also be a string; a zero can be a NULL pointer, a NULL string and a number. In such cases, the fact collector predicts all possible types for the value.



Raw Bytes	Facts (offset, type, value)
40 27 01 97 ff ff ff ff	(00 Ptr 0xffffffff97012740)
00 4B 01 97 ff ff ff ff	(08 Ptr 0xffffffff97014b00)
00 00 00 00 00 00 00 00	(16 Ptr 0x0) (16 Num 0x0) (20 Num 0x0)
00 00 00 02 00 00 00 02	(24 Num 0x2) (28 Num 0x2)
73 77 61 79 79 65 72 00	(32 String swapper)

Fig. 4: A synthetic example of low-level facts. Offset is the distance to the initial address of the object scanned by the fact collector.

Based on the aforementioned heuristics, the fact collector collects low-level facts in the form of `(Offset, Value, Type)`. Figure 4 shows a synthetic example of collected low-level facts. The offset is the distance to the beginning of the object scanned by the fact collector. Afterwards, collected facts are fed into the logic inference engine as the search space to infer forensics-required fields. Notably, as shown in Figure 2, the fact collector is invoked on demand to scan target objects, instead of whole kernel memory.

D. Object Layout Inference

LogicMem infers forensics-required fields by matching a set of logic rules over facts collected from the inferred object. Logic rules are built based on generated invariants in kernel evolution analysis. These invariants describe features (e.g., type) and relations (e.g., order and value) that forensics-required fields should hold, thus can be used to guide the logic inference engine to search for desired fields. Note that the generated invariants already consider the fact that configurable fields may or may not present in the memory. For example, in Figure 1, the `pushable_tasks` field at line 8 is configurable, thus only relative order of `tasks` and `mm`, instead of precise distance between them, is collected as an invariant. As discovered in the kernel evolution analysis, forensics-required fields are not configurable and those fields exhibits invariants across 47 observed kernels. Therefore, following the logic rules based on these invariants, the logic inference can locate the forensics-required fields regardless of kernel configurations and kernel versions. More details about the logic rules and how they can help the inference engine precisely locate the desired fields are discussed in §V.

V. Logic Inference Engine

Essentially, the logic inference engine performs an optimized search over the finite domain determined by collected facts, and find solutions that satisfy the predefined logic rules.

A. Constraints for Logic Inference

We summarize seven types of constraints based on the three types of invariants observed in the kernel evolution analysis (§III).

a) Type Constraints: Type constraints apply to fields who have unchanged types along kernel upgrades, and can help the logic inference engine reduce the search space from collected facts for certain fields. For example, the `pid` field in `task_struct` contains an integer. To infer `pid` field, the logic inference engine only needs to evaluate integers in collected facts. The `comm` field contains a string representing the process name. Therefore, the type constraint can narrow the search space for `comm` to only strings facts.

b) Object Type Constraints: Object type constraints specify types for inferred pointer fields whose types remain unchanged along kernel upgrades. When inferring a pointer field, the logic inference engine first invokes the fact collector to collect facts from the memory region pointed by the pointer, and determines if the memory region is of the expected object type by checking the logic rules associated with that object type. For example, the `mm` field in `task_struct` contains a pointer to a `mm_struct`. When inferring the `mm` field, the logic inference engine will evaluate each pointer in the collected facts and query whether it points to a valid `mm_struct`. Another property for pointers is whether a pointer can refer back to the current object. Pointers of a doubly linked list (e.g., `list_head`) or of the same type with the current object can point back to the current object. Otherwise, pointers refer to a different memory region. We also utilize this property as constraints for pointers.

c) Order Constraints: An order constraint specifies that two fields follow a strict order: one always appears after (or before) another. For example, according to the kernel evolution analysis, the `parent` field comes after the `tasks` field in `task_struct`. After the logic inference engine pinpoints the `tasks` field, it only needs to search for `parent` from pointer facts after `tasks`. Order constraints consider uncertainty of fields offsets caused by kernel configurations, i.e., constraints do not specify the exact offset of two fields if one or more configurable fields exist in between. As a result, LOGICMEM is able to locate forensics-required fields disregarding kernel configurations. We notice that since Linux 4.3, structure randomization is introduced as a new security feature. When it is enabled, the order of object fields will be randomized during compile time, thereby breaking the order constraints. We discuss about this in details in §VII.

d) Adjacency Constraints: An adjacency constraint is a strict type of order constraint that specifies the spatial offset between two fields. According to the kernel evolution analysis, some fields are always adjacent to each other and there are no configurable fields in between. For instance, `pid` and `tgid` are two integer fields in `task_struct`. `mm` and `active_mm` are two adjacent pointer fields. In such cases, we can specify the offset between fields with adjacency constraints.

e) Structural Constraints: Structural constraints apply to structures in which all fields are not changed along kernel upgrades. For those structures, if there is no kernel configuration, we build strict logic rules to describe fixed offsets of its fields. For instance, according to the analysis results, `vfsmount` remains unchanged among analyzed kernel versions and has no #ifdef directive. Having a set of strict rules for `vfsmount` can help the logic inference engine quickly locate the offset of `vfsmount` type fields in `fs_struct`.

f) Range Constraints: Range constraints narrow the search space of LOGICMEM within a specific boundary for each inferred object. According to our kernel evolution analysis (§III), although fields are added or removed along kernel upgrades, the offsets of object fields are bounded in a reasonable range. The object range is estimated during invariant detection, in which we calculate the maximum size of objects assuming all `#ifdef` directives are enabled and each field takes 8 bytes.

g) Value Constraints: Value constraints refer to the value range for a field and the value relations between fields. Since value constraints cannot be observed by the invariant detector from source code, we only consider fields whose value constraints can be derived from field definitions in source code and domain knowledge. For example, in `mm_struct`, `arg_start` and `arg_end` contain the start and end addresses of program arguments. It is reasonable to constraint `arg_start` to be less than `arg_end`. Some value constraints can be derived from field definitions. For instance, `vm_flags` in `vm_area_struct` contains bit flags that specify the behavior of and provide information about memory pages. According to the flag definition in </include/linux/mm.h>, the `VM_MAYREAD` flag specifies the `VM_READ` flag can be set, and so for the write, execute and share flags. The relations between flags can be utilized as value constraints. According to our evaluation, value constraints are simple and only for selected fields, but effective in eliminating false candidates.

Algorithm 1: Logic Rules Construction Algorithm.

```
Input  : Forensics-required fields: $fr\_fields$;
         Forensics-required objects: $fr\_objects$;
Output: Logic rules: $rules$
1  T ← type constraints; OBJ ← object constraints;
2  O ← order constraints; A ← adjacency constraints;
3  S ← structural constraints; R ← range constraint;
4  Function rule_gen($object$):
5      foreach $a \in fr\_fields$ do
6          if $a$ is a pointer then
7              $rules.add(ispointer(a))$;
8              if $pointer\_type(a) \in fr\_objects$ then
9                  $rules.add(rule\_gen(pointer\_type(a))$;
10             end
11         end
12         if $a$ is a number then
13             $rules.add(isnumber(a))$;
14         end
15         if $a$ is a string then
16             $rules.add(isstring(a))$;
17         end
18     end
19     foreach $a, b \in fr\_fields$ do
20         if $a, b \in O$ then
21             $rules.add(|a| < |b|)$;
22             if $a, b \in A$ or $S$ then
23                 $rules.add(|b| = |a| + |a, b|)$;
24             end
25             $rules.add(|b|) < R)$;
26         end
27     end
28     return $rules$
```

B. Logic Rules Construction

The logic rules generation for an inferred object is presented in algorithm 1. For each forensics-required field, we first apply type constraints in logic rules (line 7, 13, and 16). For a pointer field of forensics-required object type, we recursively generate a set of logic rules following the same algorithm (line 9). Then we add order constraints for forensics-required fields (line 21). Particularly, if fields follow adjacency constraints or structural constraints as detected by the invariant detector, we add rules to specify the spatial offsets between them (line 23). Moreover, we apply range constraints to narrow down the search space (line 25).

As an example, Listing 1 presents the Prolog logic rules for partial fields in `task_struct`. In Prolog, variables start with capital letters and logic rules are defined to constrain variables. The `Base_addr` at line 1 is the initial address of a `task_struct`. A description of logic rules can be found in Table VI in the Appendix.

a) Rules based on type constraints: As shown in algorithm 1, we first apply type constraints using logic rules. At line 3 in Listing 1, `Ptr_fields` declares a list of variables that represent offsets and values of inferred pointer fields. `Str_fields` (line 4) and `Num_fields` (lines 5) are used to constrain string fields and number fields respectively. Collected facts are stored in `Ptr_facts`, `Str_facts`, and

`Num_facts`. `tuples_in` (lines 6–8) is a built-in function that constrains those variables to be in certain types of facts. Therefore, the logic inference engine only searches candidates for a field within its respective type facts. Then we apply object type constraints for pointer fields. If the target object of a pointer field is also forensics-required, we build logic rules for forensics-required fields in it following the same rule generation algorithm 1. Then these logic rules are used by a child inference process to infer forensics-required fields. In Listing 1, `mm_struct` (line 16) is a logic rule that queries whether the address in `MM_addr` points to a `mm_struct`. It creates a new process (line 20) that first collects facts from the target address, then infers forensics-required fields using logic rules defined in `query_mm_struct` (line 22). The parent query receives results from the child query and continues the logic inference process.

b) Rules based on order constraints: Next, we build logic rules to apply order constraints. As shown in Listing 1, we use the Prolog built-in function *chain()* (lines 10) to constrain the order of object fields. The "<" symbol specifies that offsets of these field should be in increasing order. Notably, we only constraint the relative order instead of exact offsets between fields because there may be configurable fields in between, which push forensics-required fields into different offsets when compiled with different configurations. When fields are detected to be adjacent by the invariant detector (i.e., no configurable field in between), we build rules to apply adjacency constraints or structural constraints, which describe the precise offsets between these fields. At line 12, we show an example of adjacency constraints.

c) Rules based on range constraints: Range constraints are estimated by the invariant detector assuming all fields are present and each field takes 8 bytes. The range constraint does not have to be precise since we only need it to narrow down the search space. Line 14 in Listing 1 shows an example of a range rule. In the logic rules, we do not have to build range rules for every field because they are already constrained by order rules.

d) Rules based on value constraints: Value constraints are derived based on source code and human knowledge. We only build simple and intuitive value constraints for selected fields, thus it's not presented in algorithm 1. Listing 1 shows some value constraint rules for `mm_struct` at line 22. For example, `mmap_base` denotes the starting point for memory mappings in the virtual address space. Its value is architecture-specific and is greater than `0x7f0000000000` in Intel x86_64 architecture (line 30). `arg_start` and `arg_end` point to the start and end addresses of arguments. Therefore `arg_start` is less than `arg_end` (line 31).

Listing 1 also presents a logic query at line 33 implementing structural constraints. According to the kernel evolution analysis, the `vfsmount` remains the same over observed kernel versions. Therefore, we use logic rules (line 41–43) to constraint its fields to fixed offsets.

C. Logic Inference

a) Direct Predicates: Direct predicates utilize logic rules that directly derive conclusions from facts collected for the inferred object. In Listing 1, `MM_struct` and `Tasks` are constrained to be pointers (line 4–5 and line 13). Thus the logic inference engine will only consider pointers from collected facts as candidates. The order invariant rule specifies inferred fields in increasing order. The range invariant rule restricts the offset of `comm` field within a estimated size of `task_struct`. These logic rules can derive conclusions from collected facts and reduce the search space along the inference.

```
1   query_task_struct(Base_addr) :-
2       %Type constraint
3       Ptr_fields([[MM_offset, MM_val], [Parent_offset,
        ↪  Parent_val]]),
4       Str_fields([[Comm_offset, Comm_val]]),
5       Num_fields([[Pid_offset, Pid_val], [Tgid_offset,
        ↪  Tgid_val]]),
6       tuples_in(Ptr_fields, Ptr_facts),
7       tuples_in(Str_fields, Str_facts),
8       tuples_in(Num_fields, Num_facts),
9       %Order constraint
10      chain([MM_offset, Pid_offset, Parent_offset,
        ↪  Comm_offset], <),
11      %Adjacency constraint
12      Tgid_offset = Pid_offset + 4,
13      %Range constraint
14      range_constraint(Comm_offset, Base_addr,
        ↪  Obj_size),
15      %Recursive rules
16      mm_struct(MM_val),
17      task_struct(Parent_val).
18
19  mm_struct(Base_addr) :-
20      process_create(query_mm_struct, Base_addr).
21
22  query_mm_struct(Base_addr) :-
23      %Type constraint
24      Ptr_fields([[Mmap_offset, Mmap_val]]),
25      Num_fields([[Mmap_base_offset, Mmap_base_val],
26                  [Arg_start_offset, Arg_start_val],
27                  [Arg_end_offset, Arg_end_val]]),
28      ...
29      %Value constraint
30      Mmap_base_val > 0x7f0000000000,
31      Arg_start_val < Arg_end_val,
32
33  query_vfsmount(Base_addr) :-
34      %Type constraint
35      Ptr_fields([[Mnt_root_offset, Mnt_root_val],
36                  [Mnt_sb_offset, Mnt_sb_val]]),
37      Num_fields([[Mnt_flags_offset, Mnt_flags_val]]),
38      tuples_in(Ptr_fields, Ptr_facts),
39      tuples_in(Num_fields, Num_facts),
40      %Structural constraints
41      Mnt_root_offset = 0,
42      Mnt_sb_offset    = Mnt_root_offset + 8,
43      Mnt_flags_offset= Mnt_sb_offset + 8.
```

Listing 1: Part of Prolog logic rules for `task_struct`, `mm_struct`, and `vfsmount`.

b) Recursive Predicates: Recursive predicates are used when direct predicates cannot draw a conclusion from current facts. For instance, reasoning about the type of a pointer over current facts cannot produce any conclusion. Therefore, recursive predicates invoke a sub-process that performs a new logic inference. The fact collector is invoked on demand by the new logic inference process to scan target address and collect facts. Then it performs logic inference following rules defined for the requested object. For instance, as presented in Listing 1, the recursive predicate `mm_struct()` (line 19) invokes a new process which uses rules defined in `query_mm_struct` (line 22) to infer the `mm_struct` layout from the target memory region pointed by `MM_addr`. Eventually, it returns the inference results back, then the logic inference engine precedes.

To avoid endless recursive loops when applying recursive predicates on doubly linked list pointers and pointers pointing to the same type of objects, we restrict the recursive level to one in such cases. For instance, when inferring the `parent` field in a `task_struct`, the logic inference engine will not apply recursive predicates in the child inference process. Therefore, the runtime of the logic inference engine is bounded by the recursive level. In the evaluation §VI-F, we found that setting the recursive level to one is enough to infer these pointer fields.

VI. Evaluation

A. Implementation

We have implemented a prototype of LogicMem in Python and SWI-Prolog [44]. Overall, LogicMem has around 2 KLOC in Python and 2 KLOC in Prolog.

B. Experiment Setup

a) Scope: It is worth noting that only a small subset of kernel objects are required to perform memory forensics analysis. Therefore, LogicMem focuses on objects and fields required by a popular memory forensics tool, Volatility [13], and extracts forensics-required objects and fields accessed by Volatility plugins (2.6.1). To collect forensics-required information, we run all instrumented Volatility Linux plugins and print unique fields and objects that are accessed by each plugin. There are 39 Volatility plugins in total and two of them are removed for consideration. `linux_route_cache` requires Linux routing cache which was deleted after kernel 3.6, and `linux_bash` requires information gathered by gdb from a live system. Therefore, we run the rest 37 Volatility plugins to monitor accessed fields in kernel objects. As a result, we identified 220 unique fields in 56 kernel objects. In the experiments, we evaluate the ability of LogicMem to infer offsets of these forensics-required fields and make Volatility plugins functional in a binary-only situation.

b) Dataset: We first collected 16 memory dumps captured from several mainstream Linux distributions, including CentOS, Debian, and Ubuntu. We also included kernel versions that are unseen by the invariant detector to evaluate the robustness of logic rules on "future" kernel versions. To acquire memory dumps with diverse kernel configurations, we built Linux kernels with randomly determined `#ifdef` directives. More specifically, we utilized the `randconfig` feature in Linux Kbuild to generate random kernel configurations. The `randconfig` randomly chooses to disable or enable `#ifdef` directives in a way that is subject to dependency checks. Note that it is impossible to randomize all directives because the resulting kernel is unlikely to be bootable. Therefore, we used `randconfig` to randomly determine up to 100 directives in 56 forensics-required kernel objects, and left the rest directives as default values. Note that kallsyms is enabled in the default Kbuild configuration. We repeated this process until we obtained 10 bootable kernels for one version. Overall, we obtained 30 memory dumps running Linux 4.15, 4,17, and 4.19 with unbiased kernel configurations for forensice-requried fields. To collect memory dumps, we boot up the system using the x86_64 system emulator, QEMU [15] with 8GB RAM, and dump the RAM content using QEMU built-in functionality. Before the RAM content is dumped, we simulate user actions by randomly starting user programs (e.g., FireFox, PDF Viewer, etc.) until the memory usage is over 50%.

c) Ground Truth: We use Volatility to generate profiles as ground truth for collected memory dumps. Volatility generates a profile by collecting kernel symbols from `System.map` file in the live system and extracting debugging symbols from a test kernel module. Therefore, it requires that the `System.map` file is generated when the kernel in target system is built, and compiler toolchain (e.g., `dwarfdump`) and kernel headers are installed. Then we follow the instructions provided by Volatility to generate profiles for these systems as ground truth.

d) Evaluation Metrics: We use *precision* and *recall* to quantify the correctness of profiles generated by LogicMem. The precision is calculated as $P = \frac{TP}{TP+FP}$, where TP is the number of fields that are correctly extracted, FP is the number of incorrectly extracted results. The recall is calculated as $R = \frac{TP}{TP+FN}$, where FN is the number of fields whose correct offsets are not extracted by LogicMem. To measure the efficiency, we reported the runtime of each component in LogicMem, including kernel ELF extraction, kernel symbol recovery, KASLR calculation, and logic inference.

C. Recovering Kernel Symbols

As mentioned in §IV-A, kernel symbols are stored in kernel data section when kallsyms is enabled. This configuration is enabled by default for the mainstream Linux distributions and the kernels with random configurations, and LogicMem is able to recover kernel symbols for memory dumps in the evaluation dataset.

D. Precision and Recall

In this section, we evaluate the performance of LogicMem's logic inference engine and show precision and recall of the inference results. We run LogicMem on a given memory dump to infer the offsets of 220 forensics-required (i.e., required by Volatility) fields in 56 kernel objects. Then we compare the results with ground truth to calculate the precision and recall.

a) Efficacy against various kernel versions: As presented in Table II, LogicMem produces no false positives and false negatives for kernels with vendor-customized configurations. The results show that invariants hold across kernel versions and logic rules based on these invariants are precise and sound to locate forensics-requried fields. Note that LogicMem has 100% precision and recall for Linux 5.9 and Linux 5.10, which are unseen by the invariant detector. This result suggests that invariants summarized from kernel evolution analysis are robust and not likely to change in new releases.

As presented in Table II, kernels built by official distributions are compiled using different versions of `gcc`. Nevertheless, LogicMem is robust enough to correctly infer forensics-required fields and compilers do not affect our approach.

b) Efficacy against various kernel configurations: Table III show the performance of LogicMem on memory dumps with random configurations. Compared with vendor-customized kernels, kernels with random configurations have diverse object layouts. According to the results, LogicMem can still infer forensics-required fields with 100% precision and 95% recall. We further investigated the results and found that the false negatives are caused by the overly strict range constraints for `neigh_table` and `mount` structures. Our range estimation is overly simplified by assuming each field to be 8 bytes long, ignoring the fact that some fields are embedded structures. Then we fixed the range estimation in the invariant detector by expanding embedded structures according to their definitions in source code. According to the experiment, logic rules with adjusted range constraints can achieve 100% recall for all kernels with random configurations without affecting the precision. Overall, the results show that logic inference with rules based on object invariants is well-suited to locate forensics-required fields for Linux kernels with diverse configurations.

E. Baseline Comparison

We would like to perform a head-to-head comparison with three aforementioned related works, RAMPARSER [16], ORIGEN [22], and the approach proposed by Pagani et al. [35]. However, none of them are publicly available. Therefore, we cited statistics reported in their papers, and compared them with LogicMem.

In RAMPARSER [16], no precision and recall about offsets identified from ORIs was reported. Moreover, it only works on Linux 2.6.27. In order to work beyond Linux 2.6, new logic must be added to accommodate substantial changes in kernels. ORIGEN [22] shows a case study of identifying ORIs in Linux 3.13. According to their results, ORIGEN can achieve a precision of 78% when identifying offsets of seven fields in `task_struct`, while LogicMem is able to achieve a precision of 100%. Pagani et al. evaluated their approach on Debian/Linux 4.19, Openwrt/Linux 4.4, Goldfish/Linux 3.18, and Ubuntu/Linux 2.6, and compared the extracted field offsets with ground truth generated by Volatility. As presented in the paper, they can achieve precisions ranging from 74% to 89%. Compared to the above related works, LogicMem can generate a profile directly from a memory dump with 100% precision and 95% recall on average (before manual adjustment of rules).

In addition, we compared LogicMem's kernel symbol recovery with another approach developed by Pagani [3] that extracts kallsyms symbols. Both approaches can correctly recover kernel symbols from memory snapshot.

F. Runtime Performance

TABLE II: The precision and recall of profiles generated by LogicMem. * indicates this kernel version is not observed in kernel evolution analysis.

OS	Compiler Ver.	Release Date	Precision	Recall
CentOS/Linux-3.10	gcc-4.8.5	07/30/2013	100%	100%
CentOS/Linux-4.18	gcc-8.3.1	08/12/2018	100%	100%
Debian/Linux-4.19	gcc-8.3.0	10/22/2018	100%	100%
Debian/Linux-5.9*	gcc-8.3.0	10/22/2018	100%	100%
Debian/Linux-5.10*	gcc-8.3.0	10/22/2018	100%	100%
Ubuntu/Linux-4.11	gcc-6.3.0	04/30/2017	100%	100%
Ubuntu/Linux-4.12	gcc-6.3.0	07/02/2017	100%	100%
Ubuntu/Linux-4.13	gcc-7.2.0	09/03/2017	100%	100%
Ubuntu/Linux-4.14	gcc-7.2.0	12/02/2017	100%	100%
Ubuntu/Linux-4.15	gcc-7.2.0	01/28/2018	100%	100%
Ubuntu/Linux-4.16	gcc-7.2.0	04/01/2018	100%	100%
Ubuntu/Linux-4.18	gcc-8.2.0	08/12/2018	100%	100%
Ubuntu/Linux-4.19	gcc-8.2.0	11/22/2018	100%	100%
Ubuntu/Linux-4.20	gcc-8.2.0	12/23/2018	100%	100%
Ubuntu/Linux-5.3	gcc-9.2.1	09/15/2019	100%	100%
Ubuntu/Linux-5.4	gcc-9.2.1	12/24/2019	100%	100%

TABLE III: The precision and recall LogicMem on memory dumps with randomized kernel configurations. The first column represents configurations with different randomized `#ifdef` directives.

	Linux 4.15		Linux 4.17		Linux 4.19	
Config	Precision	Recall	Precision	Recall	Precision	Recall
Rand_1	100%	100%	100%	93.6%	100%	100%
Rand_2	100%	97.3%	100%	97.3%	100%	93.6%
Rand_3	100%	100%	100%	100%	100%	97.3%
Rand_4	100%	96.4%	100%	96.4%	100%	93.6%
Rand_5	100%	93.6%	100%	93.6%	100%	93.6%
Rand_6	100%	93.6%	100%	93.6%	100%	96.4%
Rand_7	100%	96.4%	100%	93.6%	100%	93.6%
Rand_8	100%	93.6%	100%	93.6%	100%	93.6%
Rand_9	100%	93.6%	100%	93.6%	100%	93.6%
Rand_10	100%	93.6%	100%	93.6%	100%	93.6%

a) End-to-end performance: In this section, we evaluated the runtime performance of LogicMem. To average the impact of KASLR, we booted up target systems three times and collected three memory dumps for each trail. Then we calculated the average runtime. Figure 5 presents the runtime of four components in LogicMem. LogicMem first performs sequential search to locate and extract the kernel ELF. The runtime of this step depends on the memory size and KASLR shift. For these 8GB memory dumps (KASLR enabled) in our dataset, it takes 36 seconds on average to finish the kernel ELF extraction step. Then LogicMem recovers kernel symbols and computes the KASLR shift from the detected kernel ELF. As presented in Figure 5, the average runtime to finish these two steps is 17 seconds and 0.39 second respectively. Afterward, LogicMem performs logic inference to reconstruct layouts of 56 selected kernel objects, which takes 51 seconds on average. Overall, the end-to-end runtime of LogicMem ranges from 89.2 to 119.36 seconds.

b) Runtime on memory dumps of different sizes: As illustrated in Figure 5, it takes similar amount of time to finish the kernel symbol recovery, KASLR computation and logic inference. The main difference lies in the runtime of kernel ELF extraction, which is determined by KASLR that shifts the kernel memory to a random location in the memory space. To further investigate the impact of KASLR and memory size, we collected a set of memory dumps of difference RAM sizes. More specifically, we ran four Linux systems, Ubuntu-16.04/Linux-4.18, Ubuntu-16.04/Linux-4.19, Ubuntu-16.04/Linux-4.20 and Ubuntu-20.04/Linux-5.3, with five RAM sizes ranging from 512MB to 8GB. To average the impact of KASLR, we boot the target systems three times to collect three memory dumps with different KASLR randomization, and calculate the average runtime. Figure 6 shows the breakdowns of runtime in this experiment. From four sub-figures, we can

TABLE IV: Runtime of LOGICMEM with different recursive levels.

Config	Linux 4.18	Linux 4.19	Linux 4.20	Linux 5.3
Recur_1	48.9s	47.9s	48.1s	49.3s
Recur_2	142.9s	147.8s	159.2s	152.3s
Recur_3	3178.4s	3105.3s	3197.8s	3201.9s
Recur_4	>10h	>10h	>10h	>10h
Recur_5	>24h	>24h	>24h	>24h

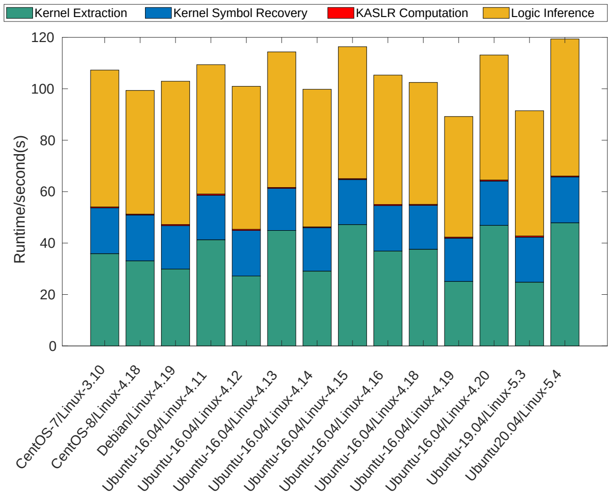


Fig. 5: Runtime performance of LOGICMEM

see that the runtime for kernel symbol recovery, KASLR computation and logic inference is similar for different memory sizes. This is because these three steps are performed on the extracted kernel memory, which is of similar size. As shown in Figure 6, the runtime for kernel ELF extraction varies for memory dumps with different sizes, and it is linearly-related with the size of memory. This is because the KASLR shifts the kernel memory into random locations in the memory. Since LOGICMEM searches sequentially to locate the kernel ELF region, it is expected to spend more time for larger memory dumps. For the Linux 4.19 and Linux 4.20 memory dumps, we can see that the runtime to extract kernel memory for 8GB memory dump is non-linear with others. The reason is KASLR shifts the kernel to lower memory region, thus it takes less time when searching sequentially. In conclusion, only the runtime of kernel extraction is affected by the size of the memory dump. Therefore, the impact of memory dump size on the performance of LOGICMEM is insignificant.

c) Performance of different recursive levels: The evaluations above show that LOGICMEM can already achieve high precision and recall with one level recursive query. To evaluate LOGICMEM with different recursive levels, we set the recursive predicates on the double linked list pointer (`tasks` field in `task_struct`) to different levels. The `tasks` field points to the next `task_struct`, by doing so, LOGICMEM recursively evaluates `task_struct` in the process list. We confirmed that the accuracy of logic inference remains the same with different recursive levels, but, according to the results Table IV, the execution time increases greatly. The reason is that Prolog recursively reasons every valid pointer to see if it satisfies logic rules defined for `task_struct`.

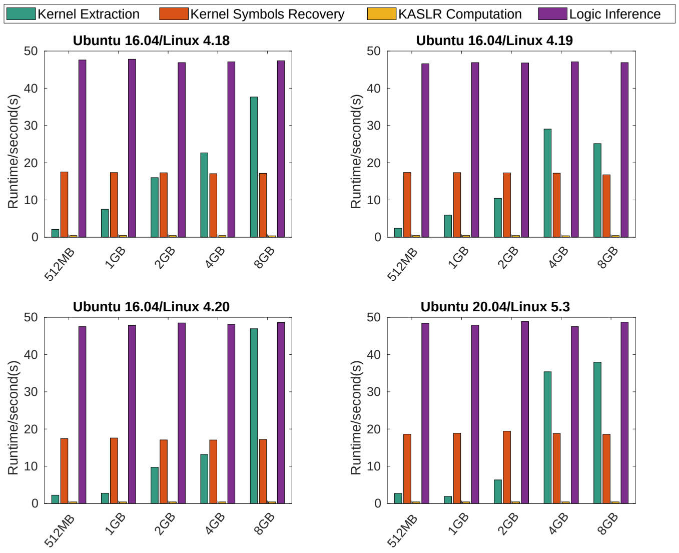


Fig. 6: Runtime of LOGICMEM on memory dumps of different sizes.

Along the recursive reasoning, a large amount of facts are collected and the query process of Prolog can be quite slow with recursive queries on large search spaces. Since the same type objects have exactly the same layout in memory and our logic rules describe what the layout should be, logic rules that hold for one object will also hold for other objects with the same layout. Therefore, it is safe to choose one level recursive query for efficiency purposes.

G. Case Study

In this section, we evaluated LOGICMEM in real-world scenarios where it is difficult (if not impossible) to apply existing memory forensics tools such as Volatility. First, we evaluated LOGICMEM on memory snapshots acquired from Armv8 64-bit Android system with Cortex-A53 CPUs. Second, we tested LOGICMEM on two memory dumps acquired from OSes designed for embedded devices (x86_64 and Arm64 architectures). Finally, we ran LOGICMEM on a memory dump collected from a system infected by a kernel rootkit.

In the experiments, Volatility failed to generate profiles for these memory dumps because kallsyms is not enabled and installation of required toolchains in target systems is not allowed. As discussed in §VII, when kallsyms is disabled, forensics-required kernel symbols are not stored in kernel memory, thereby making it impossible to recover them from memory. Nevertheless, this problem can be solved by locating required kernel objects using their field patterns. To demonstrate this idea, we develop field patterns for `task_struct` and `module` objects based on the fact that these objects contain unique strings (e.g., process and module names) and pointers that connect them with same type objects in doubly linked lists. LOGICMEM can locate their addresses in the memory dump and perform logic inference on these two objects as well as 14 objects connected to them. These 16 kernel objects are accessed by eight Volatility plugins in Table VII that output information about processes, memory mappings, and loaded kernel modules. Screenshots of sample plugin results can be found in §A.

a) Memory forensics for Android systems: In this experiment, we selected all recent long-term support (LTS) or stable kernel releases from Android kernel tree [1] and cross-compiled six ARMv8 64-bit Android kernels (android-4.4-release and android-4.9-release, android-4.14-stable, android-4.19-stable, android11-5.4-lts, and android12-5.10-lts) from source code with configurations shipped with kernel releases. We then booted each Android kernel using the whole-system emulator QEMU (qemu-system-aarch64) with Cortex-A53 virtual CPUs and dumped the RAM content using QEMU. Volatility failed to generate a profile because the installation of required toolchains was not allowed. In contrast, LOGICMEM can find addresses of two global objects, `task_struct` and `module`, and the physical address of the kernel page table. Moreover, LOGICMEM was able to infer forensics-required fields in `task_struct` and `module` as well as 14 kernel objects connected to them. Then we used offsets of inferred forensics-required fields in 16 objects to run eight Volatility plugins. To verify the correctness of plugin outputs, we compared plugin outputs with information obtained from the live system. More specifically, the first four plugins in Table VII dump information about running processes. We ran "`ps aux`" command in the live system to obtain process information and verified that the process names, PIDs, etc, were consistent. linux_memmap, linux_proc_maps, and linux_dump_map plugins print process memory information. We compared and verified the page size, addresses of memory pages, etc, with `/proc/vmallocinfo` file in the live system. The `linux_lsmod` plugin outputs a list of loaded kernel modules and we verified that the module names and addresses were consistent with the `lsmod` command output. For information that is not available in the live system (e.g., page flags), we checked if the plugin outputs were valid based on domain knowledge. According to the results, we claim that LOGICMEM can make these plugins work correctly.

b) Memory forensics for embedded devices: In this experiment, we evaluated LOGICMEM on a well-known operating system designed for routers, OpenWRT [11], and a fork of OpenWRT, Lede [5]. We downloaded router images (x86_64 and ARM64 architectures) from vendor's official website [5], [11] and booted the systems up using the whole-system emulator QEMU. Volatility also failed to generate profiles because it was forbidden due to security settings (i.e., the kernel module cannot be inserted). For the OpenWRT/ARM64 system, we found that the underlying address translation mechanism was unknown. Therefore, we utilized the fact that kernel memory is continuously mapped and simply subtracted the kernel base offset from a virtual address to compute the physical address. This offset can be calculated using the same method introduced in §IV-B. For the OpenWRT/ARM64 and Lede/x86_64 system, LOGICMEM could locate two required global objects using the pattern matching method. Then we ran LOGICMEM to infer forensics-required in these two objects and 14 kernel objects connected to them. Similarly, we compared plugin outputs with information collected from the live system and verified that they were consistent. According to the results, we claim that the generated profile can make Volatility plugins work correctly.

c) Rootkit Detection: To demonstrate the usability of LOGICMEM on security-related applications, we ran LOGICMEM to generate a profile for a rootkit-infected memory dump. We chose a popular Linux kernel rootkit named `Average Coder` [2], which is able to escalate the privilege of a user process by sharing its `cred` structure with a root process. As presented in Figure 10, we installed the `Average Coder` rootkit in the target system running Linux 3.0, and escalated the privilege of the current process (PID 1851) to root. Afterwards, we acquired the memory dump and ran LOGICMEM to generate a profile. As shown in Figure 11, the `linux_check_creds` plugin was able to detect the escalated process 1851 using the profile generated by LOGICMEM. This experiment shows that LOGICMEM can support the rootkit detection plugin without running any code in the victim system, while Volatility needs to build a profile in the victim system in advance to analysis.

VII. DISCUSSION

a) Recovering Kernel Symbols: Kallsyms is a mechanism utilized by Linux kernel to resolve all symbols for debugging purposes. As we found in the evaluation, mainstream Linux distributions are shipped with this feature by default for debugging. But it is possible for customized kernels to disable this feature. When disabled, there is no possible method to recover all kernel symbols from memory. In this work, we focus on inferring object layouts and consider recovering symbols a less challenging problem, because kallsyms is enabled by default and if not, kernel objects can be located using their unique patterns [20], [32]. To demonstrate the feasibility, we develop a pattern matching method for two forensics-required objects, `task_struct` and `module`. We use value patterns such as unique strings in these objects representing process and module names, and points-to patterns such as doubly linked list fields that point to other objects with the same type. For example, a valid `task_struct` contains a printable string with a null terminator. It also contains a pointer field that points to the next `task_struct` which also contains a valid process name. We also developed a pattern matching method similar to the one proposed by Saur et al. [38] to locate the kernel page table when it cannot be found in kallsyms symbols. Algorithms in question can be found in this anonymous repository [9]. LOGICMEM is able to locate these two objects when kallsyms is not available and infer offsets of 47 fields within these two objects and 14 objects connected to them. As presented in Table VII, LOGICMEM, generated profiles can support eight Volatility plugins even when kallsyms is disabled. As we found during the study, all existing memory forensics tools [6], [13], [27] and profile generation tools [3], [16], [22], [35], [47] rely on kernel symbols either from the live system or kallsyms in the memory dump. Our approach sheds some lights on memory forensics without kernel symbols available.

b) Data Structure Randomization: Structure randomization [8] is a new feature introduced recently to improve the security of Linux kernel. This feature, if enabled, will obfuscate the structure layouts, making the kernel hard to exploit. While public Linux distributions do not enable this configuration because distributors need to publish the randomization seeds to users for module-building, this is useful for private kernel builds. Therefore, for customized kernels with structure randomization enabled, order invariants in kernel objects are broken, which affects the precision of LOGICMEM. In the future, we can extend LOGICMEM to tolerate uncertainties like randomization using probabilistic logic inference [19], [21],

[23], in which rules and facts are annotated with probabilities. By doing so, LogicMEM may still generate valid results when some ordering rules are violated. Moreover, we can take advantage of deep learning frameworks to automatically learn new logic rules [45] or probabilities of logic rules from kernel dataset with structure randomization. Such rules can be independent of order constraints.

c) Changes of Invariants: Our logic inference approach relies on kernel field invariants that are summarized from a large variety of kernel versions. According to our kernel evolution analysis and experiments, the invariants remain persistent over observed kernel versions and remain so in new versions. For a newly released kernel, we can run our invariant detector and check whether generated invariants still hold. If so, the existing logic rules are still valid. If not, we will adjust the logic rules based on new invariants. More details about handling changes of invariants can be found in Appendix §C.

d) Automatic Logic Rule Generation: In principle, it is feasible to automatically generate these rules from the observed invariants by following the algorithm algorithm 1, and fine tune the rules if necessary. We have not implemented this by the time of writing. Instead, we manually create these rules by following the algorithm. Since the rule generation only needs to be done once for Linux kernel and, as shown in the evaluation, the rules work for a wide range of kernel versions. Thus, there is no need to repeat the rule generation and automating this process brings little benefit at this stage.

VIII. Related Work

a) Memory Forensics: Many research works [13], [14], [25], [33], [37], [40] have been proposed to facilitate memory forensic analysis from different perspectives. They aim to extract object layouts for forensic analysis purposes or propose robust memory forensics methods. Existing techniques rely on static analysis or dynamic analysis of kernel functions to locate offset-revealing instructions and figure out layout of objects. However, static approaches cannot achieve high precision in locating offsets of forensics-required fields and dynamic approaches require high-coverage seed inputs. In this paper, we propose a logic inference approach to tackle the Linux profile generation problem and it is shown to have much higher precision and recall than existing approaches. Tools like DeepMem [41] use deep learning approach to generate abstract representations for kernel objects. However, it requires a large volume of qualified training memory dumps, while our proposed approach only requires a set of logic rules.

b) Virtual Machine Introspection (VMI): VMI monitors system-level states of a running virtual machine and has been widely applied for security applications such as malware detection [30], [46] and whole-system dynamic binary analysis [18], [27]. Similarly, VMI also requires a profile of kernel objects in the guest system. LogicMem can generate profiles to support VMI tools.

c) Kernel Object Identification: Dolan-Gavitt et al. [20] presented an approach to generating robust signatures for kernel objects using their value invariants. In this work, we explore other types of invariants in kernel objects and reconstruct object layouts based on those invariants. SigGraph [32] is another work that derives signatures for kernel objects using points-to relations between objects. SigGraph only utilizes pointer fields to generate object signatures, while LogicMem uses more types of fields such as strings and numbers. DIMSUM [31] and DEC0DE [43] leverage a probabilistic inference-based approach to recovering data structure from unmappable memory regions. However, they only focus on specific data structures such as call logs, browsers, etc, while LogicMem can support memory forensics plugins implemented in Volatility. Laika [17] detects objects from memory and derives object signatures for malware identification. Specifically, it infers layouts of data structures using unsupervised Bayesian learning. However, the accuracy of this approach falls below the expectation of memory forensics.

d) Logic Inference and Its Applications: Logic inference has demonstrated its usefulness in solving security-related problems, including C++ reverse engineering [39], binary disassembly [24] and security policy analysis [28], etc. It is motivated by the fact that some analyses rely on domain knowledge or simple patterns, which can be expressed into logic rules and evaluated by a logic reasoning system. In this paper, the evaluation results suggest that the logic inference approach is well-suited for locating offsets of forensics-required fields in kernel objects.

IX. Conclusion

In this paper, we proposed a novel solution to generate profiles for binary-only memory forensics using a logic inference approach. We have implemented a prototype named LogicMem and demonstrated that the proposed logic inference approach is well-suited for locating offsets of forensics-required fields in kernel objects. To the best of our knowledge, LogicMem is the first approach that generates profiles for binary-only memory forensics with 100% precision and 95% recall on average. We also demonstrated the efficacy of LogicMem to enable memory forensics for memory dumps collected from Android and embedded systems.

Acknowledgments

We would like to thank the anonymous reviewers for their valuable suggestions and comments. This work was supported by National Science Foundation under Grant No. 1719175 and Google under Google Security and Privacy Research Award. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the funding agencies.

References

[1] "Android common kernel tree," https://android.googlesource.com/kernel/common.

[2] "Average Coder: Linux rootkit implementation," http://average-coder.blogspot.com/2011/12/linux-rootkit.html.

[3] "kallsyms extractor," https://github.com/pagabuc/kallsyms-extractor.

[4] "Kaslr: Kernel address space layout randomization," https://lwn.net/Articles/569635/.

[5] "Lede: Linux Embedded Development Environment," https://lede.readthedocs.io/en/latest/.

[6] "Libvmi," http://libvmi.com.

[7] "Linux kernel release model," http://kroah.com/log/blog/2018/02/05/linux-kernel-release-model/.

[8] “Linux kernel structure randomization,” https://lwn.net/Articles/722293/.

[9] “LogicMem source code,” https://anonymous.4open.science/r/logicmem/.

[10] “Microsoft debugging symbol server,” https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/accessing-symbols-for-debugging.

[11] “OpenWrt: Wireless freedom,” https://openwrt.org/.

[12] “Unicorn: The Ultimate CPU emulator,” https://www.unicorn-engine.org/.

[13] “Volatility: Memory Forensics System,” https://github.com/volatilityfoundation/volatility.

[14] A. Ali-Gombe, S. Sudhakaran, A. Case, and G. G. R. III, “Droidscraper: A tool for android in-memory object recovery and reconstruction,” in *22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2019)*. Chaoyang District, Beijing: USENIX Association, Sep. 2019, pp. 547–559. [Online]. Available: https://www.usenix.org/conference/raid2019/presentation/ali-gombe

[15] F. Bellard, “Qemu, a fast and portable dynamic translator,” in *In Proceedings of the USENIX Annual Technical Conference (ATC ’05)*, ser. ATC, 2005.

[16] A. Case, L. Marziale, and G. G. Richard III, “Dynamic recreation of kernel data structures for live forensics,” *Digital Investigation*, vol. 7, 08 2010.

[17] A. Cozzie, F. Stratton, H. Xue, and S. T. King, “Digging for data structures,” in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI’08. USA: USENIX Association, 2008, p. 255–266.

[18] A. Davanian, Z. Qi, Y. Qu, and H. Yin, “Decaf++: Elastic whole-system dynamic taint analysis,” in *22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2019)*. Chaoyang District, Beijing: USENIX Association, Sep. 2019, pp. 31–45. [Online]. Available: https://www.usenix.org/conference/raid2019/presentation/davanian

[19] L. De Raedt and A. Kimmig, “Probabilistic (logic) programming concepts,” *Machine Learning*, 2015.

[20] B. Dolan-Gavitt, A. Srivastava, P. Traynor, and J. Giffin, “Robust signatures for kernel data structures,” ser. CCS ’09. New York, NY, USA: Association for Computing Machinery, 2009, p. 566–577. [Online]. Available: https://doi.org/10.1145/1653662.1653730

[21] A. Dries, A. Kimmig, W. Meert, J. Renkens, G. Van den Broeck, J. Vlasselaer, and L. De Raedt, “Problog2: Probabilistic logic programming,” in *Machine Learning and Knowledge Discovery in Databases*, A. Bifet, M. May, B. Zadrozny, R. Gavalda, D. Pedreschi, F. Bonchi, J. Cardoso, and M. Spiliopoulou, Eds. Cham: Springer International Publishing, 2015, pp. 312–315.

[22] Q. Feng, A. Prakash, M. Wang, C. Carmony, and H. Yin, “ORIGEN: Automatic extraction of offset-revealing instructions for cross-version memory analysis,” in *Proceedings of the 11th ACM Asia Conference on Computer and Communications Security (ASIACCS’16)*, May 2016.

[23] D. Fierens, G. V. den Broeck, J. Renkens, D. S. Shterionov, B. Gutmann, I. Thon, G. Janssens, and L. D. Raedt, “Inference and learning in probabilistic logic programs using weighted boolean formulas,” *CoRR*, vol. abs/1304.6810, 2013. [Online]. Available: http://arxiv.org/abs/1304.6810

[24] A. Flores-Montoya and E. Schulte, “Datalog disassembly,” in *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Aug. 2020, pp. 1075–1092. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/flores-montoya

[25] Y. Fu, Z. Lin, and D. Brumley, “Automatically deriving pointer reference expressions from binary code for memory dump analysis,” in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2015. New York, NY, USA: Association for Computing Machinery, 2015, p. 614–624. [Online]. Available: https://doi.org/10.1145/2786805.2786810

[26] Godfrey and Qiang Tu, “Evolution in open source software: a case study,” in *Proceedings 2000 International Conference on Software Maintenance*, 2000, pp. 131–142.

[27] A. Henderson, A. Prakash, L. K. Yan, X. Hu, X. Wang, R. Zhou, and H. Yin, ““make it work, make it right, make it fast”, building a platform-neutral whole-system dynamic binary analysis platform,” in *Proceedings of the 2014 International Symposium on Software Testing and Analysis (ISSTA’14)*, Jul. 2014.

[28] G. Hernandez, D. J. Tian, A. S. Yadav, B. J. Williams, and K. R. Butler, “Bigmac: Fine-grained policy analysis of android firmware,” in *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Aug. 2020, pp. 271–287. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/hernandez

[29] A. Israeli and D. G. Feitelson, “The linux kernel as a case study in software evolution,” *Journal of Systems and Software*, vol. 83, no. 3, pp. 485 – 501, 2010. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0164121209002519

[30] X. Jiang, X. Wang, and D. Xu, “Stealthy malware detection through vmm-based ’out-of-the-box’ semantic view reconstruction,” in *Proceedings of the 2007 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS ’07, 2007.

[31] Z. Lin, J. Rhee, C. Wu, X. Zhang, and D. Xu, “Dimsum: Discovering semantic data of interest from un-mappable with confidence,” in *in: Proceedings of the 19th Annual Network and Distributed System Security Symposium (NDSS’12)*, 2012.

[32] Z. Lin, J. Rhee, X. Zhang, D. Xu, and X. Jiang, “Siggraph: Brute force scanning of kernel data structure instances using graph-based signatures,” in *Proceedings of the Network and Distributed System Security Symposium, NDSS 2011, San Diego, California, USA, 6th February - 9th February 2011*. The Internet Society, 2011.

[33] Z. Lin, X. Zhang, and D. Xu, “Automatic reverse engineering of data structures from binary execution,” in *Proceedings of the Network and Distributed System Security Symposium, NDSS 2010, San Diego, California, USA, 28th February - 3rd March 2010*. The Internet Society, 2010. [Online]. Available: https://www.ndss-symposium.org/ndss2010/automatic-reverse-engineering-data-structures-binary-execution

[34] R. Lotufo, S. She, T. Berger, K. Czarnecki, and A. Wąsowski, “Evolution of the linux kernel variability model,” in *Software Product Lines: Going Beyond*, J. Bosch and J. Lee, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 136–150.

[35] F. Pagani, “ADVANCE IN MEMORY FORENSICS,” *Doctoral dissertation, EURECOM*, 2019.

[36] F. Pagani and D. Balzarotti, “Autoprofile: Towards automated profile generation for memory analysis,” *ACM Trans. Priv. Secur.*, vol. 25, no. 1, nov 2021. [Online]. Available: https://doi.org/10.1145/3485471

[37] N. L. Petroni, A. Walters, T. Fraser, and W. A. Arbaugh, “Fatkit: A framework for the extraction and analysis of digital forensic data from volatile system memory,” *Digital Investigation*, vol. 3, no. 4, pp. 197 – 210, 2006. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1742287606001228

[38] K. Saur and J. B. Grizzard, “Locating ×86 paging structures in memory images,” *Digit. Investig.*, vol. 7, no. 1–2, p. 28–37, oct 2010. [Online]. Available: https://doi.org/10.1016/j.diin.2010.08.002

[39] E. J. Schwartz, C. F. Cohen, M. Duggan, J. Gennari, J. S. Havrilla, and C. Hines, “Using logic programming to recover c++ classes and methods from compiled executables,” in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS ’18, 2018.

[40] A. Slowinska, T. Stancescu, and H. Bos, “Howard: A dynamic excavator for reverse engineering data structures,” in *Proceedings of the Network and Distributed System Security Symposium, NDSS 2011, San Diego, California, USA, 6th February - 9th February 2011*. The Internet Society, 2011. [Online]. Available: https://www.ndss-symposium.org/ndss2011/howard-a-dynamic-excavator-for-reverse-engineering-data-structures

[41] W. Song, H. Yin, C. Liu, and D. Song, “Deepmem: Learning graph neural network models for fast and robust memory forensic analysis,” in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS ’18, 2018.

[42] R. Tartler, C. Dietrich, J. Sincero, W. Schröder-Preikschat, and D. Lohmann, “Static analysis of variability in system software: The 90,000 #ifdefs issue,” in *2014 USENIX Annual Technical Conference (USENIX ATC 14)*. Philadelphia, PA: USENIX Association, Jun. 2014, pp. 421–432. [Online]. Available: https://www.usenix.org/conference/atc14/technical-sessions/presentation/tartler

[43] R. J. Walls, E. Learned-Miller, and B. N. Levine, “Forensic triage for mobile phones with dec0de,” in *Proceedings of the 20th USENIX*

Conference on Security, ser. SEC'11. USA: USENIX Association, 2011, p. 7.

[44] J. Wielemaker, T. Schrijvers, M. Triska, and T. Lager, "SWI-Prolog," *Theory and Practice of Logic Programming*, vol. 12, no. 1-2, pp. 67–96, 2012.

[45] F. Yang, Z. Yang, and W. W. Cohen, "Differentiable learning of logical rules for knowledge base reasoning," in *Proceedings of the 31st International Conference on Neural Information Processing Systems*, 2017, pp. 2316–2325.

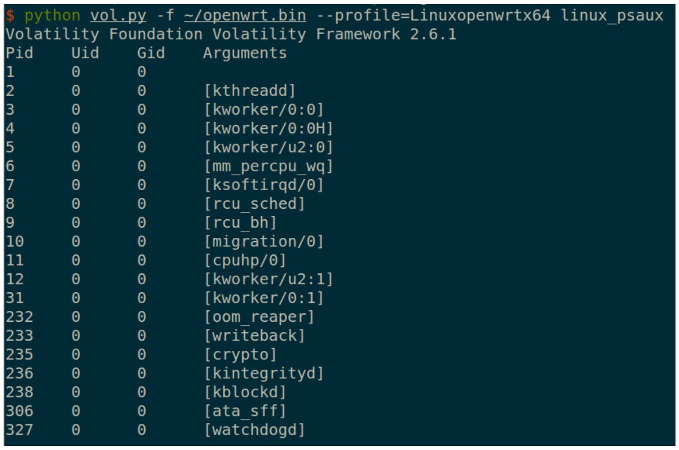
[46] H. Yin, D. Song, E. Manuel, C. Kruegel, and E. Kirda, "Panorama: Capturing system-wide information flow for malware detection and analysis," in *Proceedings of the 14th ACM Conferences on Computer and Communication Security (CCS'07)*, October 2007.

[47] S. Zhang, X. Meng, and L. Wang, "An adaptive approach for linux memory analysis based on kernel code reconstruction," vol. 2016, no. 1, Dec. 2016. [Online]. Available: https://doi.org/10.1186/s13635-016-0038-z

[48] S. Zhang, X. Meng, L. Wang, and G. Liu, "Research on linux kernel version diversity for precise memory analysis," in *Data Science*, B. Zou, M. Li, H. Wang, X. Song, W. Xie, and Z. Lu, Eds. Singapore: Springer Singapore, 2017, pp. 373–385.

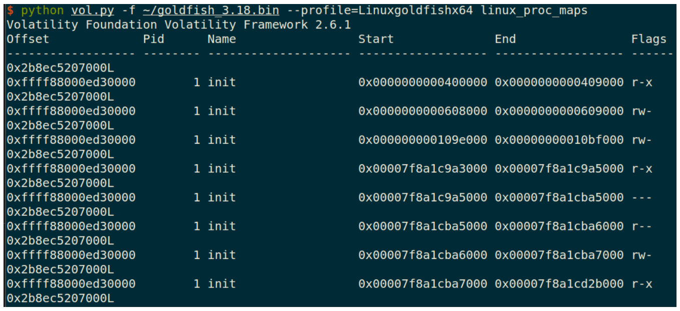
APPENDIX A
VOLATILITY PLUGIN OUTPUTS

We run LOGICMEM on memory dumps collected from an Android system and router OSes. Since there is no ground truth of profiles, i.e., Volatility failed to generate profiles for these systems, we run Volatility plugins with profiles generated by LOGICMEM and verify the correctness of plugin outputs based on domain knowledge and information obtained from live systems. Figure 7, Figure 8, and Figure 9 present screenshots of three Volatility plugins. Figure 10 and Figure 11 show the screenshots of using profile generated by LOGICMEM to support a rootkit detection plugin.



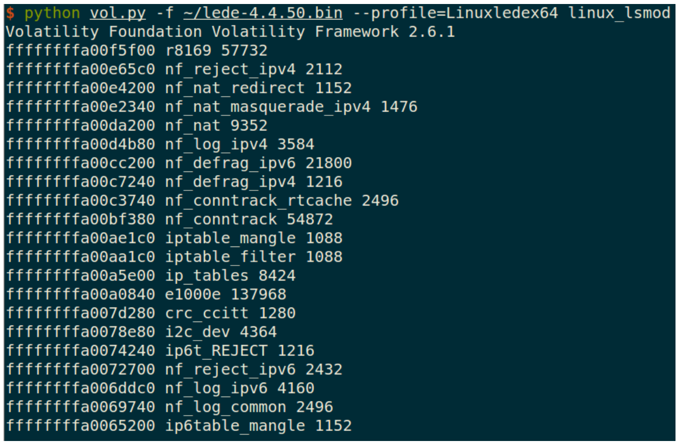
```
$ python vol.py -f ~/openwrt.bin --profile=Linuxopenwrtx64 linux_psaux
Volatility Foundation Volatility Framework 2.6.1
Pid    Uid    Gid    Arguments
1      0      0
2      0      0      [kthreadd]
3      0      0      [kworker/0:0]
4      0      0      [kworker/0:0H]
5      0      0      [kworker/u2:0]
6      0      0      [mm_percpu_wq]
7      0      0      [ksoftirqd/0]
8      0      0      [rcu_sched]
9      0      0      [rcu_bh]
10     0      0      [migration/0]
11     0      0      [cpuhp/0]
12     0      0      [kworker/u2:1]
31     0      0      [kworker/0:1]
232    0      0      [oom_reaper]
233    0      0      [writeback]
235    0      0      [crypto]
236    0      0      [kintegrityd]
238    0      0      [kblockd]
306    0      0      [ata_sff]
327    0      0      [watchdogd]
```

Fig. 7: The output of linux_psaux plugin on an OpenWRT memory dump using a profile generated by LOGICMEM.



```
$ python vol.py -f ~/goldfish_3.18.bin --profile=Linuxgoldfishx64 linux_proc_maps
Volatility Foundation Volatility Framework 2.6.1
Offset             Pid      Name                 Start              End                Flags
------------------ -------- -------------------- ------------------ ------------------ ------
0x2b8ec5207000L
0xffff88000ed30000        1 init                 0x0000000000400000 0x0000000000409000 r-x
0x2b8ec5207000L
0xffff88000ed30000        1 init                 0x0000000000608000 0x0000000000609000 rw-
0x2b8ec5207000L
0xffff88000ed30000        1 init                 0x000000000109e000 0x00000000010bf000 rw-
0x2b8ec5207000L
0xffff88000ed30000        1 init                 0x00007f8a1c9a3000 0x00007f8a1c9a5000 r-x
0x2b8ec5207000L
0xffff88000ed30000        1 init                 0x00007f8a1c9a5000 0x00007f8a1cba5000 ---
0x2b8ec5207000L
0xffff88000ed30000        1 init                 0x00007f8a1cba5000 0x00007f8a1cba6000 r--
0x2b8ec5207000L
0xffff88000ed30000        1 init                 0x00007f8a1cba6000 0x00007f8a1cba7000 rw-
0x2b8ec5207000L
0xffff88000ed30000        1 init                 0x00007f8a1cba7000 0x00007f8a1cd2b000 r-x
0x2b8ec5207000L
```

Fig. 8: The output of linux_proc_maps plugin on an Goldfish memory dump using a profile generated by LOGICMEM



```
$ python vol.py -f ~/lede-4.4.50.bin --profile=Linuxledex64 linux_lsmod
Volatility Foundation Volatility Framework 2.6.1
ffffffffa00f5f00 r8169 57732
ffffffffa00e65c0 nf_reject_ipv4 2112
ffffffffa00e4200 nf_nat_redirect 1152
ffffffffa00e2340 nf_nat_masquerade_ipv4 1476
ffffffffa00da200 nf_nat 9352
ffffffffa00d4b80 nf_log_ipv4 3584
ffffffffa00cc200 nf_defrag_ipv6 21800
ffffffffa00c7240 nf_defrag_ipv4 1216
ffffffffa00c3740 nf_conntrack_rtcache 2496
ffffffffa00bf380 nf_conntrack 54872
ffffffffa00ae1c0 iptable_mangle 1088
ffffffffa00aa1c0 iptable_filter 1088
ffffffffa00a5e00 ip_tables 8424
ffffffffa00a0840 e1000e 137968
ffffffffa007d280 crc_ccitt 1280
ffffffffa0078e80 i2c_dev 4364
ffffffffa0074240 ip6t_REJECT 1216
ffffffffa0072700 nf_reject_ipv6 2432
ffffffffa006ddc0 nf_log_ipv6 4160
ffffffffa0069740 nf_log_common 2496
ffffffffa0065200 ip6table_mangle 1152
```

Fig. 9: The output of linux_lsmod plugin on an Lede memory dump using a profile generated by LOGICMEM.

APPENDIX B
KERNEL SYMBOL RECOVERY

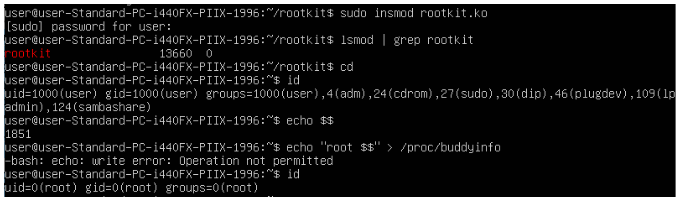
We identify value patterns for data structures that store symbol addresses and names. As presented in Figure 3, symbol virtual addresses are stored in *kallsyms_addresses* (defined in kernel/kallsyms.c). We use the pattern that these addresses are valid kernel addresses with the same prefix, in a increasing order and reside continuously in kernel data section. After Linux 4.6, a new configuration is introduced to save memory usage by storing a relative base address and an array of offsets. *Kallsyms_offsets* contains a list of 4-byte integers with patterns similar to *kallsyms_addresses*. Symbol names are compressed and stored in three data structures. *kallsyms_names* contains lengths of symbol names and indexes to *kallsyms_token_index*, and *kallsyms_token_index* contains indexes to *kallsyms_token_table*, which stores compressed strings of symbol names. Therefore, *kallsyms_token_table* contains printable strings terminated with zero. *kallsyms_token_index* contains a list of unsigned short numbers in increasing order and start from zero. We then follow the kernel function *kallsyms_expand_symbol* (defined in kernel/kallsyms.c) to decompress symbol names.

APPENDIX C
KERNEL EVOLUTION ANALYSIS

Table V presents the full kernel evolution analysis results for objects required by Volatility. According to the analysis, we found that most of fields required by Volatility are not changed along kernel upgrades. In the experiments, we did encounter cases where the layout of a kernel object has noticeable changes and invariants cannot be summarized. For example, after Linux 4.5, four fields in `module` structure were wrapped into other data structures and the order of those fields also changed. Also, there are 53 field swaps happened

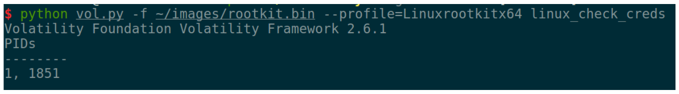
TABLE V: Linux kernel object evolution analysis on 56 kernel objects over 47 kernel versions. *pid_link was deleted since Linux 4.19. *module_layout was introduced since Linux 4.5.

Object Name	Tot. # of added fields	Tot. # of deleted fields	Tot. # of modified fields	Tot. # of swapped fields	Avg. # of all fields	# of required fields	# of changed required fields	% of unchanged fields
task_struct	146	66	19	7	213	15	1	97.6%
mm_struct	26	20	4	0	58	13	0	98.2%
vm_area_struct	7	6	0	2	20	6	0	98.5%
fs_struct	0	0	0	0	7	2	0	100%
mount	11	6	2	7	27	8	0	97.9%
dentry	3	2	1	1	18	6	0	99.2%
cred	4	1	0	0	27	4	0	99.6%
pid_namespace	14	4	1	0	17	4	1	97.7%
pid	1	0	0	0	6	1	0	99.6%
upid	0	1	0	0	3	1	1	99.4%
pid_link*	0	0	0	0	2	2	0	100%
file	10	3	0	0	22	2	0	98.8%
qstr	2	2	0	0	3	2	0	97.8%
vfsmount	0	0	0	0	4	3	0	100%
fdtable	1	1	2	0	6	2	0	100%
inode	19	6	7	1	53	4	1	97.1%
module	24	21	7	0	47	8	4	97.7%
module_layout*	1	0	0	0	7	3	0	99.3%
seq_operations	0	0	0	0	5	1	0	100%
tty_driver	3	4	1	0	23	7	0	99.2%
tty_struct	14	40	14	1	47	5	0	96.9%
tty_ldisc	2	2	0	0	3	1	0	97.3%
file_operations	14	7	2	0	29	7	0	98.4%
rb_root	0	0	0	0	1	1	0	100%
file_system_type	3	1	1	0	14	3	0	99.3%
kset	0	0	0	0	5	1	0	100%
kobject	1	0	1	0	13	4	1	99.7%
kernel_param	3	1	1	0	9	4	0	98.8%
kparam_array	0	0	0	0	6	5	0	100%
module_kobject	1	0	0	0	6	2	0	99.6%
kref	0	0	0	0	2	1	0	100%
neigh_table	6	2	0	0	27	4	0	99.3%
neigh_hash_table	2	2	0	0	5	1	0	98.3%
neighbour	5	1	0	1	26	6	0	99.4%
net_device	67	28	11	32	123	7	0	97.6%
in_device	3	0	0	0	20	2	0	99.7%
in_ifaddr	5	0	1	0	16	2	0	99.2%
socket	1	3	0	0	9	4	0	99.1%
inet_sock	6	3	0	0	21	4	1	99.1%
sock	36	13	4	53	71	2	0	96.9%
ipv6_pinfo	10	10	0	0	28	2	1	98.5%
unix_sock	5	7	18	0	12	4	0	95.1%
sockaddr_un	0	0	0	0	3	1	0	100%
cpuinfo_x86	13	12	0	0	32	2	0	98.4%
resource	1	0	0	0	8	7	0	99.7%
radix_tree_node	13	31	2	1	10	2	0	88.2%
tcp_seq_afinfo	1	4	0	0	4	4	2	97.6%
udp_seq_afinfo	1	4	0	0	5	4	2	98.1%
files_struct	5	2	0	0	10	2	0	98.6%
list_head	0	0	0	0	2	2	0	100%
hlist_head	0	0	0	0	2	2	0	100%
hlist_node	0	0	0	0	2	2	0	100%
rb_node	1	1	0	0	4	2	0	98.9%
sock_common	2	3	32	0	30	6	2	98.1%
proc_dir_entry	12	5	2	14	16	12	4	95.7%
timekeeper	19	10	4	8	19	6	4	95.4%



```
user@user-Standard-PC-i440FX-PIIX-1996:~/rootkit$ sudo insmod rootkit.ko
[sudo] password for user:
user@user-Standard-PC-i440FX-PIIX-1996:~/rootkit$ lsmod | grep rootkit
rootkit                13660  0
user@user-Standard-PC-i440FX-PIIX-1996:~/rootkit$ cd
user@user-Standard-PC-i440FX-PIIX-1996:~$ id
uid=1000(user) gid=1000(user) groups=1000(user),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),109(lp
admin),124(sambashare)
user@user-Standard-PC-i440FX-PIIX-1996:~$ echo $$
1851
user@user-Standard-PC-i440FX-PIIX-1996:~$ echo "root $$" > /proc/buddyinfo
-bash: echo: write error: Operation not permitted
user@user-Standard-PC-i440FX-PIIX-1996:~$ id
uid=0(root) gid=0(root) groups=0(root)
```

Fig. 10: Escalating the privilege of a user process using Average Coder kernel rootkit.



```
$ python vol.py -f ~/images/rootkit.bin --profile=Linuxrootkitx64 linux_check_creds
Volatility Foundation Volatility Framework 2.6.1
PIDs
--------
1, 1851
```

Fig. 11: The Volatility rootkit detection plugin output using profile generated by LOGICMEM.

to `sock` structure, and most of them are between Linux 4.9 and 4.10. In such cases, it is hard to summarize invariants and have generic logic rules for all observed versions. Therefore, we build different sets of logic rules to accommodate kernel changes that break our invariants.

TABLE VI: Logic rules and descriptions

Logic Rules	Description
Field = [Offset, Val]	Offset and value representing an inferred field
Ptr_fields(Field, ...)	Create an array containing all inferred fields that should be pointers
Num_fields(Field, ...)	Create an array containing all inferred fields that should be numbers
Str_fields(Field, ...)	Create an array containing all inferred fields that should be strings
Ptr_facts	An array that contains all pointer type facts collected in the inferred object
Num_facts	An array that contains all number type facts collected in the inferred object
Str_facts	An array that contains all string type facts collected in the inferred object
tuples_in(Ptr_fields, Ptr_facts)	Constraint all inferred pointer fields should be in collected pointer facts
chain(Field_A[offset], Field_B[offset], ..., '<')	Constraint spatial order of inferred fields
Field_B[offset] = Field_A[offset] + distance	Constraint spatial distance between two inferred fields
Field_A[offset] < Range_constraint	Constraint the offset of a field within a certain range
Field_A[value] >/</= Value_constraint	Constraint the value of a field with in certain range
query_obj_name(obj_addr)	A query that contains all logic rules designed for an inferred object, with the object address as an argument
procress_create(obj_name, obj_addr)	Create a sub-process to infer a typed pointer, with the target address as an argument

TABLE VII: Eight Volatility plugins that are functional using generated profiles. ● means the plugin is functional using the generated profile, and ○ means the plugin does not work due to misidentifications of some fields.

	Goldfish-3.8	Android-4.4-release	Android-4.9-release	Android-4.14-stable	Android-4.19-stable	Android11-5.4-lts	Android12-5.10-lts	Openwrt/Linux-4.5	Lede/Linux-4.11
linux_pslist	●	●	●	●	●	●	●	●	●
linux_psaux	●	●	●	●	●	●	●	●	●
linux_pstree	●	●	●	●	●	●	●	●	●
linux_lsof	●	●	●	●	●	●	●	●	●
linux_memmap	●	●	●	●	●	●	●	●	●
linux_proc_maps	●	●	●	●	●	●	●	●	●
linux_dump_map	●	●	●	●	●	●	●	●	●
linux_lsmod	●	●	●	●	●	●	●	●	●