AutoProfile: Towards Automated Profile Generation for Memory Analysis

FABIO PAGANI, UC Santa Barbara
DAVIDE BALZAROTTI, Eurecom

Despite a considerable number of approaches that have been proposed to protect computer systems, cyber-criminal activities are on the rise and forensic analysis of compromised machines and seized devices is becoming essential in computer security.

This article focuses on memory forensics, a branch of digital forensics that extract artifacts from the volatile memory. In particular, this article looks at a key ingredient required by memory forensics frameworks: a precise model of the OS kernel under analysis, also known as *profile*. By using the information stored in the profile, memory forensics tools are able to *bridge the semantic gap* and interpret raw bytes to extract evidences from a memory dump.

A big problem with profile-based solutions is that custom profiles must be created for each and every system under analysis. This is especially problematic for Linux systems, because profiles are not *generic*: they are strictly tied to a specific kernel version and to the configuration used to build the kernel. Failing to create a valid profile means that an analyst cannot unleash the true power of memory forensics and is limited to primitive carving strategies.

For this reason, in this article we present a novel approach that combines source code and binary analysis techniques to automatically generate a profile from a memory dump, *without* relying on any non-public information. Our experiments show that this is a viable solution and that profiles reconstructed by our framework can be used to run many plugins, which are essential for a successful forensics investigation.

CCS Concepts: • **Applied computing → System forensics;**

Additional Key Words and Phrases: Memory forensics, memory forensics profile, linux kernel

ACM Reference format:
Fabio Pagani and Davide Balzarotti. 2021. AutoProfile: Towards Automated Profile Generation for Memory Analysis. *ACM Trans. Priv. Secur.* 25, 1, Article 6 (November 2021), 26 pages.
https://doi.org/10.1145/3485471

1 INTRODUCTION

While traditionally focused on the analysis of the information stored on hard drives, in recent years digital forensics broadened its scope to cover other components of computer systems.

This project was supported by the European Research Council (ERC) under the European Unions Horizon 2020 research and innovation programme (grant agreement No 771844 BitCrumbs) and by the European Unions Horizon 2020 research and innovation programme under grant agreement No. 786669 (ReAct).

Authors' addresses: F. Pagani, UC Santa Barbara, USA; email: pagani@ucsb.edu; D. Balzarotti, Eurecom, France; email: davide.balzarotti@eurecom.fr.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2021 Copyright held by the owner/author(s). Publication rights licensed to ACM.

2471-2566/2021/11-ART6 $15.00
https://doi.org/10.1145/3485471

One of these components, the volatile memory, is becoming more and more crucial in many investigations, because it contains a number of artifacts which are not found elsewhere. Moreover, in large organizations, the analysis of volatile memory—best known as memory forensics—is nowadays not only used as part of incident response, but also as a proactive tool to periodically check machines and look for signs of compromise or infection. For example, Microsoft has recently announced Project Freta [32], a cloud-based solution to detect malicious processes and rootkits using memory forensics techniques.

The core idea behind memory forensics is to extract evidences from the data structures used by the operating system kernel. While some of these structures can be located by carving the memory for particular byte patterns, the true power of memory forensics comes from the so-called *structured* analysis. In most of the cases, this type of analysis starts by finding a set of global symbols inside a memory dump. From these variables, other kernel structures are then discovered by de-referencing pointers [28]. For example, a common task performed in memory forensics consists of listing the processes that were running inside the machine when the memory dump was acquired. Under Linux, a way to retrieve this information is to find the location of the global variable `init_task` and use it to traverse the list of `task_structs`. However, even this simple and seemingly straightforward operation can be performed only if the tool has a *very* detailed model of the system under analysis. In memory forensics, this detailed model is called a *profile*. A typical profile contains two different pieces of information: the address of kernel global variables and the layout of kernel objects. The latter is of particular interest because it is influenced by several different factors, including the kernel version, the way the kernel was configured at compile time, and the compiler optimizations. Without a complete profile, none of the existing memory forensics frameworks—such as Volatility, Rekall, and Project Freta—are able to analyze a memory dump [4].

For Microsoft Windows operating systems, retrieving the correct profile from the system under analysis is not really a problem, because the number of different kernels is limited and well known. Moreover, the layout can be retrieved from the kernel debugging symbols, which are generally available on a public server. On the other hand, memory forensics is more and more focusing on Linux-based operating systems, both for the analysis of servers and to support a wide range of appliances, mobile phones, and network devices. Unfortunately, when it comes to Linux there is no central symbol server and the number of combinations of kernel versions and possible configuration is countless.

However, it is important to understand that the main challenge is *not* to determine the specific version of the kernel under analysis. Thus, it is not important how much kernel structures change across kernel versions, but instead how much they change *within* a single version—because of user configurations or compiler options. Previous research [49] have empirically confirmed this effect and reported that the layout of important forensics structures is affected by the configuration used at compile time. Thus, forensics analysts have to create a profile for each and every system they want to analyze. Currently, this is a manual process that involves the compilation of a special kernel module. While this operation is generally performed on the machine under analysis, it may also be performed offline by cross-compiling the module on the analyst workstation. In both cases, this process has several important requirements. For instance, it requires access to the kernel headers, the kernel configuration file and, in certain cases, the very same compiler toolchain used to build the kernel (as different compilers or compiler versions can result in different data structure offsets and layouts). In some cases, like in the latest development version of Volatility—the *de facto* standard framework when it comes to memory forensics—the profile generation even requires to have access to the full debugging symbols or to recompile the entire kernel itself. While the previous constraints might not be an obstacle for a common desktop machine, the required information are rarely available for kernels running on network appliances, `IoT` devices, smartphones, or highly

optimized servers—thus effectively limiting the applicability of memory forensics. Moreover, with the advent of cloud-based solutions, system administrators, cloud providers, and forensics analysts are faced with the challenge of diagnosing thousands of machines. This was also recently highlighted by Project Freta's developers, when they remarked how "*no commercial cloud has yet provided customers the ability to perform full memory audits of thousands of **virtual machines (VMs)** without intrusive capture mechanisms and a priori forensic readiness.*" and how they intend "*to automate and democratize VM forensics*" to the point where it can be done "*with the push of a button*" [32]. In this scenario, building a model through a trial and error process is unpractical and no assumption can be made on how a kernel was configured and built.

To make things worse, the source code of the kernel module needs to be manually updated every time a new kernel is released [17]. In fact, the definition of several structures used by memory forensics tools is not exported from header files and therefore must be copied into the module source code. For example, the definition of the `mount` structure contained in the current version of the kernel module shipped by Volatility [45] does *not* match the one of the current stable kernel. A similar problem arises also in the context of kernel *backports*, i.e., new features that are retrofitted to older kernels by manufacturers and Linux distributions. In these cases, by only looking at the kernel version of the machine under investigation is not possible to infer the correct definition of a kernel structure used for memory forensics. This is a severe problem because an analyst does not have any way to assess the integrity and the correctness of the profile and the target machine might not be available anymore when the error is detected.

Finally, modern kernels include the ability to perform *structure layout randomization*, which poses a serious "threat" to memory forensics. Originally developed as a protection mechanism by Grsecurity [40] and later studied by other researchers [5, 19, 22], structure layout randomization is nowadays present in the latest versions of the Linux kernel as well. This compile-time option randomizes the layout of sensitive kernel structures, as an effective way to harden the kernel against exploitation. As a side effect, the authors highlight that enabling this option will "prevent the use of forensic tools like Volatility against the system".

All of the previous limitations are also shared by Rekall [6], another well-known forensics framework. In fact, even if Rekall stores the OS profiles in a different format, the process of extracting the layout of kernel structures is identical to the one implemented by Volatility.

The memory forensics community is well-aware of all of these problems, as recently emphasized once again by Case and Richard [4] in their overview of memory forensics open challenges. In the article, the authors urge the community to create a public database of Linux profiles—which nowadays exists only in the form of a commercial solution. Unfortunately, they also note how a "considerable amount of monetary and infrastructure" is needed to create such a database and how, in any case, this approach can only cover settings used by pre-compiled kernel shipped as part of Linux distributions. This is the case of the Volatility community repository [44], which unfortunately has received only a few contributions in the past few years. While this repository contains more than 230 profiles (both for `x86` and `x86_64`), the latest versions of widely-used distributions are not present. For example, the most recent profile for OpenSuse dates back to 2013, while the latest profile for Ubuntu targets version 18.04.

In the past years, researchers have also proposed partial solutions to the profile generation problem. For example, Case et al. [3] and subsequently Zhang et al. [48], suggested that the layout can be retrieved from the analysis of kernel code. Unfortunately, their *manual* approach cover only a handful of kernel structures layout, while nowadays memory forensics requires several hundreds of them. On the other hand, approaches such as the proposed one by Socała and Cohen [38] still requires the configuration that was used to build the kernel under analysis.

For these reasons, we believe it is time to move away from costly manually-curated profiles and investigate the possibility to design a holistic and fully automated approach to memory analysis. As a first step in this direction, in this article we propose AutoProfile, a novel approach to automatically create Linux profiles. To the best of our knowledge, this is the first solution to create entire profiles based *only* on information publicly available or extracted from the memory dump itself. Our experimental results show how the profiles extracted by AutoProfile support several Volatility plugins—such as those that list the running processes and the open files—when targeting a very diverse set of kernels. This set includes a version of a Debian kernel that use structure layout randomization, an Android kernel, a kernel running on Raspberry Pi devices, a kernel shipped by Openwrt (a project targeting network devices), and an old version of the Ubuntu kernel released more than a decade ago.

2 RECOVERING OBJECTS LAYOUT FROM BINARY CODE

In this section, we discuss a practical example of how the layout of an object is shaped by the configuration used at compile time, thus making it impossible to deduce the correct offsets of its fields by reasoning only on its definition. We then introduce the core idea behind this article and how it can be generalized to recover the layout of all kernel objects used in memory forensics.

2.1 Problem Statement

The key ingredient that makes memory forensics possible is the availability of the kernel *profile*: a detailed model of the symbols and data types required to perform the analysis. In the case of Linux memory forensics, a profile contains two separate pieces of information: the addresses of global variables and kernel functions, and the exact layout of kernel objects. The latter is of particular interest for different reasons. First of all, this information is lost during the compilation process and the only way to preserve it is to ask the compiler to emit the debugging symbols. This is often the case for kernels shipped by common Linux distributions that usually provide them in a separate debugging package. Moreover, the Linux kernel is a highly customizable piece of software, designed to run on a large variety of devices and architectures and to suit different needs. This means that the very same kernel version tailored to two different systems can result in dramatic differences between the layout of the kernel objects.

To illustrate how the customization of the Linux kernel is in fact a problem for memory forensics, we present a practical example in Figure 1. In the left part of this figure, we show a short code snippet responsible for the set up of a task which, in this example, is represented by the `task` object. Every task has a pointer to the next task, some credentials, and a name. Moreover, in case the macro `CONFIG_TIME` was defined at compile time, a task also includes the field `start_time`. The function `setup_task` initializes a task and its fields. In the right part of the Figure, we instead report the disassembly of two versions of this function, one in which the macro was defined at compile time ①, and one in which it was not ②.

The first difference between the two versions is present at lines 3 and 1, respectively. The semantic of these two instructions is equivalent: they store the argument `new_name` (passed in the `rsi` register) into the `name` field. However, the offset of this field is different between the two version, and so is the displacement from `rdi` (which contains the `t` argument). This is a consequence of the fact that in ① the compiler had to reserve 8 bytes before the field `name` for the `start_time` field, while in ② the latter was entirely removed by the preprocessor. On the other hand, the same displacement is used to access the field `gid` of `creds` at lines 4 and 2. This is because the field `cred` - and subsequently the field `gid` therein contained—precedes `start_time` and thus is not concerned by its presence or by its absence.

```c
1   struct creds{
2     uint32_t uid;
3     uint32_t gid;
4   };
5
6   struct task{
7     struct task *next;
8     struct creds cred;
9   #ifdef CONFIG_TIME
10    uint64_t start_time;
11  #endif
12    char *name;
13  };
14
15  void setup_task(struct task *t,
16                  char *new_name,
17                  int gid)
18  {
19    t->name = new_name;
20    t->cred.gid = gid;
21  #ifdef CONFIG_TIME
22    t->start_time = time(NULL);
23  #endif
24  }
```

```
(1) CONFIG_TIME defined
1   push   rbx
2   mov    rbx,rdi
3   mov    QWORD PTR [rdi+0x18],rsi
4   mov    DWORD PTR [rdi+0xc],edx
5   xor    edi,edi
6   call   0x1030 <time@plt>
7   mov    QWORD PTR [rbx+0x10],rax
8   pop    rbx
9   ret
```

```
(2) CONFIG_TIME not defined
1   mov    QWORD PTR [rdi+0x10],rsi
2   mov    DWORD PTR [rdi+0xc],edx
3   ret
```

Fig. 1. On the left the C source code we use in our examples, on the right its compiled form.

While this is a trivial example, it introduces a very common pattern that is present thousands of times in the kernel codebase. For example, the definition of the `task_struct` alone—which is one of the most important object in memory forensics—is shaped by more than 60 different *#ifdef*s. The large number of combinations that derive from these definitions make it impractical to enumerate all possible offsets where a field can be located. However, as we saw in our example, this information is encoded into the compiled code and therefore we believe the only practical way to precisely recover the layout of kernel objects is by *extracting* it from the kernel binary itself.

2.2 Data Structure Layout Recovery

The intuition behind this article is that, while the precise structures' layout is lost during the compilation process, it affects the code generated by the compiler. More specifically, the displacement used to access the fields of a given object must reflect the layout of the data structures and therefore can be extracted if we know *where* each field is used across the entire codebase, and *how* the code is accessing the field. These two pieces of information allow us to locate the functions that operate on the requested field, and to follow the access pattern that led the code to a particular object. For example, a piece of data can be passed as parameter, but it can also being referenced by a global variable, reached by traversing another object, or obtained by calling a separate function.

Back to our example, let's assume we want to recover the offset of the *name* field. First, by looking at the source code, we can tell that the function `setup_task` accesses this field and also that the variable `t` is passed as parameter. Given that the Abstract Binary Interface (ABI) of `x86-64` [24] specifies that the first parameter is passed using the `rdi` register, we can perform a data-flow analysis and track every memory access whose value depends on the `rdi` register. In version (1), this happens at lines 3 and 4, but also at line 7 because `rbx` was initialized from `rdi`.

Table 1. A Review of Previous Attemps at Automated Profile Generation

Approach	Limitations
Case and Zhang [3, 48]	Manual approach which requires high knowledge of the kernel internals
ORIGEN [11]	Requires dynamic analysis of a running kernel similar to the target one
Layout Expert [38]	Requires the configuration file used to compile the kernel
Type Inference (discussed in Section 9)	Designed to recover the types, and not to distinguish fields in a structure
Memory Carving (discussed in Section 9)	Orthogonal to structured memory forensics, does not require a profile

It is important to note that it is very difficult to tell which of the three access is the one operating on the field we are interested in. In fact, functions often access dozens of different fields and compilers optimizations often change the exact order and number of those accesses in the binary code. However, we can leverage the fact that the *name* field is also probably accessed in other functions, and therefore we can combine and cross-reference multiple candidate locations to narrow down its exact offset. In Section 7.1, we will describe in detail the numerous challenges the layout recovery algorithm needs to face when dealing with complex kernel code and the solutions we adopted to overcome these problems.

3 PAST APPROACHES

The forensics community is well aware of this problem and over the years have proposed some preliminary solutions, which are summarized in Table 1. The first attempt at solving this problem was published by Case et al. [3] in 2010 and, quite similarly, by Zhang et al. [48] in 2016. The two approaches are quite straightforward: after locating a set of defined functions, the authors extracted the layout of kernel objects by looking at the disassembly of these functions. While we believe this was a step in the right direction, these approaches had several limitations. First of all, both the functions and the corresponding objects were selected manually. This limited the scalability of the solution, and in fact the authors were only able to manually recover a dozen fields in total—while our experiments show how Volatility uses more than two hundred fields. Moreover, to locate the functions in the memory dump, previous solutions rely on the content of `System.map`, therefore suffering from some of the problems and limitations we discussed in Section 1. Finally, since the authors used a simple pattern-matching algorithm to extract the offsets from the disassembled code, those approaches worked only on small functions and only if the instructions emitted by the compiler followed a certain predefined pattern.

`ORIGEN` [11] was one of the first attempts to automate these steps. The system combines both static and dynamic analysis to generate a model for a base version of a program by identifying the so-called *Offset Revealing Instructions*. Then, this model is matched against a subsequent version of the program, allowing the system to perform a cross-version memory analysis. While their results look promising, the system was only tested against six fields of `task_struct` and by their same own admission, the system may not work "when a software version is significantly different from the base version". Moreover, this approach requires to perform dynamic analysis of a running kernel configured in a way similar to the one under analysis. Our solution requires instead only to perform static analysis of the kernel source code, independently of how the target system has been configured. Finally, the authors assume that the program under analysis *uses* the structure they want to recover during their dynamic labeling phase. While these uses are trivial to observe for frequently used structures (such as `task_struct`), it might become more challenging for less used ones.

Case et al. [3] and Zhang et al. [48] presented also another way to find the offset of a field based on the relationship among global kernel objects. Both authors noted that, for example, the field `comm` of the variable `init_task` always contains the string "`swapper`" and that the field `mm` of the same variable always points to another global variable (`init_mm`). With this information is

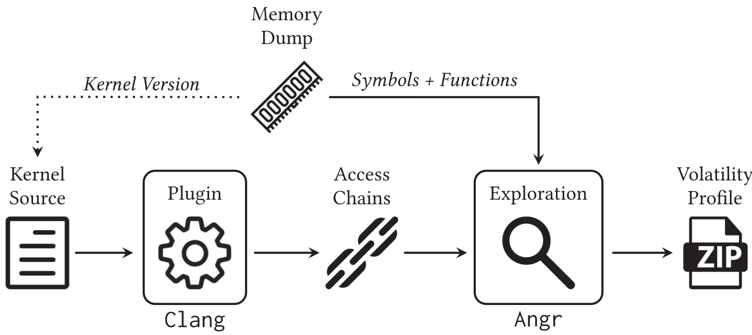

Fig. 2. AutoProfile overview.

trivial to extract the offsets of these two fields, because it is enough to find the starting address of `init_task` and scan the following chunk of memory. Unfortunately, not all the object types in the kernel have a corresponding global variable, thus limiting this approach to a very narrow subset of data structures.

Finally, in 2016 Socała and Cohen [38] presented an interesting approach to create a profile on-the-fly, without the need to rely on the compiler toolchain. Their tool, `Layout Expert`, is based on a *Preprocessor AST* of kernel objects, that retains all the information about the `ifdefs`. This special AST is created offline and then specialized to the system under analysis, only when the analyst has access to it. Nevertheless, the specialization process still needs the kernel configuration and the `System.map`, making this technique not applicable to our scenario.

4 APPROACH OVERVIEW

In this section, we explain our approach to automatically extract a valid memory forensics profile from a memory dump. Our system can be conceptually divided in three independent phases as illustrated in Figure 2. In the first phase, we find the location of all symbols in the memory dump and we identify the version of the running kernel. During the second phase we use a compiler plugin to analyze the source code of the identified version and emit a set of models—which we call *access chains* —that describe the way the code operates over a selected set of kernel objects. It is important to note that we only need access to the public source code but not to the exact configuration (kernel options, compiler settings, randomization seed, etc.) that was used to build the kernel captured in the memory dump. The chains extracted in this phase are finally fed into the third component, the *exploration engine*, which matches them to the actual kernel binary code extracted from the memory dump. The final output of AutoProfile is a working memory forensics profile, which can be used by Volatility to extract evidences from a memory dump.

For example, during the second phase AutoProfile would discover that the field `vm_file` of the structure `vm_area_struct` is used in the function `shm_close` and that the variable at the base of the access chain is the first parameter of the function. Then, during the third phase, our tool locates the aforementioned function by using the symbols extracted in the first phase, and tracks the offsets of every memory access that depends on the first parameter. This process produces the position (or a set of candidate positions which are then compared and intersected with the same information extracted from other functions) for the `vm_file` field.

5 PHASE I: KERNEL IDENTIFICATION AND SYMBOLS RECOVERY

The goal of the first phase is to recover two key pieces of information: the version of the kernel and the location of its symbols (functions and global variables).

Locating Kernel Symbols. As we already explained in Section 1, existing memory forensics tools require to know the location of certain global symbols to bootstrap their analysis. On top of that, AutoProfile also requires the location of some kernel functions, which will serve as basis for our analysis.

The recovery of this information is greatly complicated by two different factors. First of all, unlike other memory forensics tools, we cannot rely on the `System.map` file, which is instead always part of a memory forensics profile. Moreover, we want AutoProfile to be resilient against **Kernel Address Space Layout Randomization** (**KASLR**)—which is nowadays enabled by default by almost every Linux distribution. To date, the forensics community already proposed several systems to recover kernel symbols from a memory dump, for example, `ksfinder` [15], `volatility-android` [41], and the solution presented by Zhang et al. [48]. These approaches leverage the fact that some symbols of the kernel are exported using the `EXPORT_SYMBOL` macro that allows kernel modules to transparently access kernel objects and functions. Whenever a symbol is exported with this macro, the kernel initializes and inserts in the `___ksymtab` section a `kernel_symbol` structure that contains two fields: one with the virtual address of the exported symbol and the other one pointing to a string representing the symbol name—which in turn is placed in the `__ksymtab_strings` section. To locate a given symbol, all previous approaches scan the memory dump to find the physical address of the string representing the symbol and then assume they can translate this physical address to a virtual one by adding a constant, based on the virtual base address of the kernel. With this information they are then able to scan the memory dump and match the corresponding `kernel_symbol` object.

The first problem with this solution is that exported symbols constitute only a tiny subset of all the kernel symbols. For this reason, Zhang et al. [48] introduced a way to recover another larger subset of symbols—called the `kallsyms` —which are usually accessible from userspace from a file under `/proc`. However, since there are tens of thousands of symbols, in order to save space they are stored in a compressed form using a table lookup algorithm. As a result, they are much harder to locate in a memory dump, and several kernel global variables are needed to decode their names. To overcome this problem, Zhang suggested to locate these variables from the disassembly of the function `update_iter` - which can be found by carving the corresponding `kernel_symbol`. Once these variables are found, by manually re-implementing the decoding algorithm the authors were finally able to reconstruct the `kallsyms`. Unfortunately, this approach requires a considerable manual effort and the authors did not discuss an automated way to retrieve the address of the global variables used in their approach. The second, much more severe, limitation of all existing solutions is that they fail on modern `X86_64` platforms with KASLR, where both the virtual and the physical base addresses are randomized.

For these reasons, we designed a novel and generic way to automatically extract the addresses of all kernel functions and global variables. Our approach extends the ideas presented so far, but it relies on automatically finding and *executing* the `kallsyms_on_each_symbol` function. This function is present in the kernel tree since more than 10 years, it is exported with the `EXPORT_SYMBOL` macro and it is responsible to handle the symbol decoding process, making it a perfect match for our purpose. AutoProfile starts by carving a number of candidate `ksymtab` tables based on few constraints (e.g., the structure needs to include include two side-by-side valid kernel addresses, `value` and `name`, greater than `0xffffffff80000000` and at least 500 contiguous `kernel_symbol` objects). We also know that the symbol representing the function `kallsyms_on_each_symbol` must be contained in one of these candidates. To find the correct one, we leverage the fact that even when KASLR is enabled, the randomization happens at the page granularity and hence offsets inside a page are left unaltered. So we scan again the memory and record every physical address matching the string `kallsyms_on_each_symbol`. Given the previous fact, we select the

kernel_symbols that have a name pointer with the same page offset of one of the matched strings. To translate the value field from the virtual to the physical address space we leverage the fact that the kernel is always contiguously mapped in the both address spaces and thus the following equation holds: $\texttt{value}_{va} - \texttt{name}_{va} = \texttt{value}_{pa} - \texttt{name}_{pa}$.

Therefore, to find the physical address of a candidate kallsyms_on_each_symbol function, we sum the physical address of the string to the difference between the value and the name virtual addresses. AutoProfile can now extract the function code from the memory dump and execute its code by using the Unicorn emulator [30]. Since one of the function's parameters is a callback function that is invoked for each decoded symbol, we pass a function under our control and retrieve from there the name and the address of each symbol, as they are processed. Finally, we will release this technique as a standalone tool or to embedded it in current memory forensics tools to effectively determine the kernel layout randomization shift.[1]

Kernel Version Identification. Multiple techniques exist to identify the version of a kernel contained in a memory dump. The straightforward approach consists of grepping for strings that match the format of a Linux kernel banner. However, even thought the kernel is generally loaded in the first few megabytes of the physical address space and therefore the correct version should be in the first few matches, this technique can potentially result in several false positives, depending on the content of the memory dump. Because of this, we resort to a more precise identification by extracting the global variable init_uts_ns and the corresponding textual representation contained in the variable linux_banner. The location of these variables is retrieved together with all other symbols as described in the previous section. Others orthogonal approaches to retrieve this information were presented by Roussev et al. [33] and Lin [21] and are based on matching fuzzy hash and SigGraph signatures previously generated from a set of kernels.

6 PHASE II: CODE ANALYSIS

At the end of the first phase we identified the version of the running kernel, which we can use to download its corresponding source code. In this second phase we automatically analyze the code to extract three pieces of information: the type definitions, the pre-processor directives, and the access chain models.

The bulk of our analysis is performed by a custom plugin for the Clang compiler, which operates on the **Abstract Syntax Tree** (**AST**) of the Linux kernel. While the analysis we need to perform would be much easier and more practical if performed at a later stage of the compilation process—i.e., by working on the compiler intermediate representation—working on the AST provides the advantage of being compatible with *all* version of the Linux kernel. In fact, while recent versions of the kernel can compile with Clang and few older versions are supported through a set of manually created patches, for the vast majority of kernel versions Clang is not able to produce an intermediate representation. However, Clang is "fault tolerant" when it builds the AST and thus it creates one for all versions of the Linux kernel, regardless of being able to compile the sources.

To recover the aforementioned pieces of information, we compile the kernel configured with allyesconfig with our plugin, which is triggered every time an AST representing a function or a record is created. The choice of this particular configuration comes from the fact that, by turning on all the configuration options, it increases the coverage of our plugin over the kernel codebase. Nevertheless, we decided to manually turn off several debugging configuration options which are never present in production kernels. The actual analysis starts at the root node of a function and recursively visits the whole tree by using a depth-first strategy.

[1] https://github.com/pagabuc/kallsyms-extractor.

6.1 Pre-processor Directives

The first piece of information we save from the compilation process is the position of `macro` and `ifdef` directives. To extract this information we use `pp-trace`, a standalone tool from the Clang framework that traces the preprocessor activity. For each of the previous directives `pp-trace` emits where they begin, where they end and, in the case of macros, also their names. This information is used for several purposes. First of all, we ignore chains extracted from lines included in `ifdef` statements, because their code is dependent on a specific configuration setting and thus might not be included in the kernel under investigation. Our tool also saves where the compiler directives related to structure randomization are used. In this way, by matching this information with the definition of a structure, our system knows which structures are affected by layout randomization. Finally, as we will explain in Section 7.1, by combining this information with the definition of kernel objects, it is possible for our tool to safely deduce the offset of certain fields.

6.2 Types Definition

Along with the functions' AST, our plugin also visits the AST representing the definition of kernel objects. When traversing this tree it saves the type of each object along with the name, the type, and the definition line of its fields. As a special case, when exploring unions, the tool marks the fields they contain accordingly.

The information gathered from parsing a record definition plays an important role in our system. For example, by looking at the order in which the fields are defined, our exploration system can constrain the candidate offsets for a given field. Moreover, the offset of certain fields can be statically deduced (e.g., we safely assume the first field in a structure is always at offset zero).

6.3 Access Chains

To model the way the code accesses kernel objects we introduce the concept of *access chain*, defined as a triple *{Location, Transitions, and Source}*. In the triple, the *Location* defines where the access is performed, in terms of a file name, a function name, and a line number. The *Transitions* element is a list containing the type of the objects and the name of the fields of every data structure involved in the chain. For example, the chain describing the access at line 3 of Figure 3 would contain three elements:

```
struct task->next|struct task->cred|struct creds.gid
```

Finally, the third element of an access chain is its *Source*, that represents how the first variable of the chain is initialized. This information is essential to select among the memory accesses contained in a function only those belonging to a target object. In the previous example, since the base variable is `task`, the source of the chain would be marked as the first parameter of the function `free_next`. AutoProfile supports three different types of sources: function *parameters*, *global variables*, and values returned from a function invocation (*function returns*). The representation of the source depends on its category: parameters are expressed as numerical position in the argument list, while the other two categories are expressed, respectively, through the name of the global variable or the name of the function.

Local variables, which can be legitimately used as base variables for an access, are not valid sources. This is because local variables must be initialized before they can be used and their initialization must fall in one of the previous categories. As we will explain in the next section, a core aspect of the plugin is that it keeps a *map* from variables to their initialization. This enables the plugin to correctly determine the source for each access chain.

```
1   int free_next(struct task *task){
2     struct task *t = task->next;
3     int gid = t->cred.gid;
4     if (strcmp(t->name, "init")){
5       free(t)
6       return gid;
7     }
8     return -1;
9   }
```

Fig. 3. Example used to explain how the Clang plugin works.

The plugin extracts access chains from the kernel source code by parsing three types of nodes in the AST: assignments and declarations, object accesses, and function calls and returns.

Assignments and Declarations are used to maintain the map of all variables and the way they are initialized. For instance, when we encounter the node representing the declaration at line 2 of Figure 3, the plugin first extracts the variable used in the **left-hand side** (**LHS**) of the statement. If the type of the variable is a `struct` or a `void` pointer, the plugin proceeds by analyzing the **right-hand side** (**RHS**) of the statement. In case the RHS is already a valid source (parameter, global variable, and function call) or an object access then we update the map with this information. On the other hand, if the RHS represents another local variable, then we lookup in the map how this other local variable was initialized and copy this information in the map entry of the LHS variable. This mechanism ensure that, at any given point inside a function, our plugins knows how a variable is initialized.

To simplify the analysis, our plugin only keeps track of one path, and not all possible paths where a variable can be assigned. However, to extract the offset corresponding to a given access is sufficient to find *one* path inside a function that reaches that access, rather than exploring *all* of them.

Object Accesses (as modeled by `MemberExpr` in Clang terminology) are the nodes that, for example, represent the right part of the statement at line 3 of Figure 3. Since in this case there are several objects chained together, the plugin keeps track of every field name and object type when traversing this sub-tree. When it reaches the base of the access, represented in this case by the variable `t`, a number of things can happen. If the base is a valid source itself (e.g., a parameter, a global variable, or a function) then the chain can be already emitted. Otherwise, if the base is a local variable then we recursively visit its initialization, appending in front of the chain the object types and the field names. This recursive process ends when a valid source is found and thus the chain can be emitted. For example, when the plugin traverses the sub-tree representing line 3, it first extracts the type of the object and the field name, i.e., `struct creds.gid` and `struct task→ creds`, and appends them to the chain. Then, since the variable `t` is a local variable, it checks in the definition map how this variable is initialized. Since `t` is initialized from an object access at line 2, it recursively traverses this access and it appends to the chain the element `struct task→ next`. At this point the process ends because the base variable `task` is a valid source.

When traversing the objects involved in a chain, the plugin keeps track of how fields are accessed. While the C standard defines the arrow and the dot operator as the only way to access a field, we are also interested in other operators that may affect an access. The first is related to the `offset_of` extension and in particular to the macro `container_of`, which is built on top of it. This macro is extensively used in linked list and trees implementations, and it defines a sort of parent-child relationship between kernel objects. In fact, given a child structure and its offset inside the parent structures, the macro is used to retrieve a pointer to the parent object. For example, supposing that `c` is a pointer to a `struct creds`, the task containing it can be retrieved by calling `t = container_of(c, struct task, cred)`. A chain containing this macro needs

to be treated carefully—not only because an offset is rather subtracted than added to the base pointer—but also because the compiler often merges a `container_of` element and the subsequent displacement in a single instruction. The other operator the plugin keeps track of is the reference operator (&). As we will explain in the next section, this is of particular importance when chains are joined, because it may transform an arrow in a dot operator. Finally, fields defined as array are generally accessed in a different way and thus need a particular technique during the exploration process. Therefore, if an element of a chain is a `container_of`, an array, or it contains a reference operator we mark it accordingly in our model.

Function Calls and Returns are the last two types of nodes explored by the plugin. This information is essential to extract accesses in functions which are inlined by the compiler. When our plugin encounters a function call, we save the name of the called function and its arguments. Similarly to how object accesses are represented, every argument is expressed as an access chain. The only difference is that these chains might have an empty *Transitions* element. This happens, for example, when one function calls another and it passes as parameter one of its own arguments or a global variable. A similar approach is applied to return statements.

6.4 Non Unique Functions

Another problem when dealing with projects in the size of the Linux kernel is that function names are not always unique. In fact, the `static` identifier is used to limit the scope of a function to a file. For example, this happens with the function `s_next` that, in kernel version 5.1, is defined 5 different times. This is a problem for our system, because whenever we analyze a function we must ensure that we are dealing with the correct "instance" of the function. Since there is no straightforward way to extract this information using Clang, we employed Joern [47]. This tool, among other things, contains a fuzzy parser for C and C++. The output of Joern after parsing the kernel sources, is a list of functions and the filename where they are defined. This information is used whenever AutoProfile extracts a function from a memory dump. In case the function has a non-unique name, we exploit the fact that functions defined in the same compilation unit ends up in the same object file and thus are also contiguous in the kernel binary. In this way, by checking the functions in the vicinity of the target one, our system is able to select the correct function.

Finally, for optimization reasons, the compiler can decide to remove a parameter from a function or even split a function in two or more parts. Fortunately, when these optimizations are applied, the compiler also adds a suffix—respectively, `.isra` and `.part` —to the name of the function. In the first case, we simply ignore the function, while in the second one AutoProfile is able to extract and join all the different pieces.

7 PHASE III: PROFILE GENERATION

It is important to point out that a profile includes the layout of only a small subset of all kernel data structures—those that are needed to complete the forensic analysis tasks supported by a given tool. For this reason, our system focuses on recovering only the information actually used by Volatility. However, manually listing the objects used by every Volatility plugin is a tedious and error prone process, and it is further complicated by the fact that some of these objects vary depending on the kernel version. Therefore, for our tests we decided to instrument Volatility to log every field it traverses and then we recovered the full list by executing each plugin against a memory dump containing the same kernel version of the one under analysis.

As a result, the actual number of different fields and unique data structures vary among the experiments, ranging from 234 and 239 targets. As we will explain in the next sections, finding the correct offset of a field enables AutoProfile to test other chains that *depends* on this field. For this reason, we add to the initial set of targets any field that represent a *dependency* of a field

used by Volatility in any access chain. Moreover, to *constrain* even more the offsets extracted for a structure, we expand the set of targets by adding three fields which are defined before or after any Volatility target.

7.1 Binary Analysis

To match the chains extracted during the source code analysis against the functions extracted from an actual memory dump we use `angr` [35] and its symbolic execution capabilities as a taint engine [16, 31]. Therefore, we decided to perform our exploration by symbolizing the source of a chain and run the function while tracking every time the symbolic variable is used as a base for a memory access. To avoid state explosion—one of the major problem of symbolic execution—we wrote a custom *exploration technique*. An exploration technique drives the symbolic engine and decides how the program is explored, by selecting which states can advance and which should be discarded. In our case, it keeps track of every state generated by the symbolic execution engine and prunes those which have already been explored more than a certain amount of time, effectively limiting the state space. Since the constraints associated with a state evolve during the symbolic exploration, our technique uses the instruction pointer as a mean to decide whether a state must be explored or discarded. Moreover, we also instruct `angr` to check the satisfiability of the constraints belonging to a state as infrequently as possible, rather than checking them when a new state is created. For example, assuming two states are created from a branch instruction then both states will be kept, *regardless* of their satisfiability. These two expedients allow the number of state to be contained but also to entirely cover the code contained in a function.

While tracking the memory accesses is independent from the source of a chain, it dictates how the system is initialized and run. Parameters and function returns are the most straightforward sources to handle. In the first case, a symbolic variable is stored in the corresponding register, while in the second—whenever the function specified in the source is called—we set the `rax` register as symbolic. On the other hand, global variables require two different strategies to handle both pointers and normal variables. In both cases, whenever the address of the variable is stored in a register we symbolize the register itself. Moreover, when the variable is not a pointer, the compiler might have already pre-computed the address of the field. If this is the case, we directly extract the offset and append it to the list of results. Since the size of non-pointer variables is known from the `kallsyms` —by subtracting the address of the `kallsym` following the one representing the global variable —our system can discern cases where more than one non-pointer variables are accessed in the same function.

Field Dependencies – AutoProfile often needs to deal with chains spanning multiple objects. For instance, let us consider again our sample chain:

```
struct task->next|struct task->cred|struct creds.gid
```

The code reaches the target `gid` by first traversing the `next` pointer of the `task` structure, thus defining a *dependency* among the two fields. In other words, we first need to recover the offset of `next` before we are able to extract the second half of the chain.

In this case, we create multiple symbolic variables and appropriately store them when a memory access belonging to an element is detected. However, since the final assignment of a field offset is obtained by a global algorithm by majority voting, it is possible that a chain cannot be fully analyzed in one pass, but instead requires a recursive approach to first identify all its dependencies.

Nested Structures – A particular type of dependency occurs when the target field is accessed through a nested structure. In C, this may appear, for example, in the form of

struct a.struct b.target. In this case, the compiler may split the access in two parts, by first loading the base address of struct b (for instance, located at 0x20 bytes from the beginning of struct a) and then adding the offset of the field (e.g., 0x16 bytes into struct b). However, this is often optimized by computing the total offset from the base structure at compile time, resulting in a single instruction like lea rax, [rsi+0x36].

This requires our tool to keep track of this displacement, as 0x36 is not the correct offset of struct b.target, and to obtain the right value we need to remove the offset of struct b, which (like in the case of field dependencies) needs to be already discovered in a previous pass.

7.2 Dealing with Inlined Functions

Since the kernel is always compiled with the optimizations turned on, the compiler is quite aggressive when it comes to function inlining. For example, compiling the Linux kernel 5.1 with the default configuration results in the inlining of more than 200,000 call sites. For this reason, being able to cope with function inlining dramatically increase the number of chains our exploration system can test.

When we analyze a memory dump and discover that a given function call has been inlined, we trigger a dedicated routine in charge of merging and inheriting its chains. Our process starts by labeling every chain of the inlined function as *forward* or *backward*. Forward chains are those that starts from a parameter, while backward ones are those that terminates in return statements. For example, in the following snippet:

```
1 inline struct task* foo(struct task *t, char *n){
2   t->name = n;
3   if(t->cred)
4     return t->next;
5   else
6     global_task->next = 0;
7   ...
```

the chain at line 2 is a forward chain, while the one at line 4 is both a forward and backward chain. Our algorithm is divided in two independent parts: in the first one chains are joined, while in the second one they are inherited.

The first one starts by iterating over every pair of caller and callee. If the callee is *not* inlined, and thus is present in the list of functions extracted from the memory dump, then no action is required. Otherwise, each argument—which is also represented with a chain—is joined with every forward chain of the callee that has the same parameter position as source. Joining is not a commutative operation: the source and the location of the argument chain are left untouched, while the list of objects of the callee chain are appended to the one of the argument chain. A similar treatment is reserved for backwards chain, but this time in the opposite direction. Every chain of the caller that has source equal to an inlined function, is joined with the backward chains of this function. Since the inlining depth can be greater than 1, i.e., functions called from inlined functions can be inlined as well, we repeat this process in a loop to propagate the presence of freshly joined chains, until any new chain is generated.

The second part of the process deals with inheriting from inlined functions all the chains which are not forward or backwards one, for example, those who access a global object. In this case, the chain is left unaltered and only added to the set of chains of the caller. In our example, as result of this process, a function that calls foo will have as well the chain representing the global access at line 6. Similarly to the previous process, we also propagate inherited chains by repeating this process in a loop.

Once this two steps are finalized, AutoProfile passes over the resulting chains to *clean* and *adjust* them. The cleaning process is needed because a target can be present in multiple same-source chains of a function. For this reason, given a target, we delete the chains which are a superset of others, thus ensuring that the target is tested only once. On the other hand, the *adjustment* deals with chains containing the reference operator or `container_of`. In the first case, we translate the arrow following a reference in a dot, but only if the chain is not used as parameter for a function. Given the following example:

```
1 void set_gid(struct creds *c, int g){
2   c->gid = g;
3 }
4 ...
5 struct creds *c = &t->cred;
6 set_gid(c, 0);
```

if `set_gid` is inlined, then the compiler will most likely merge the accesses to fields `cred` and `gid` in a single one. As we explained in section 7.1, this chain can be explored only if the offset of either `cred` or `gid` is known. On the other hand, if the function is not inlined, no action is required and the chain containing `cred` can be safely explored.

The adjustment of `container_of` deals with a similar problem. In the following example:

```
1 t = container_of(c, struct task, cred);
2 t->next = NULL;
```

the compiler may effectively subtract from `c` the offset of `cred` and then add the offset of `next`, or merge the previous two operations and add to `c` the distance between `cred` and `next`. In this case, to represent these two possibilities, we duplicate the chain, explore both of them and merge their results.

7.3 Object Layout Inference

At the end of the binary exploration phase, each target (i.e., each field whose offset we need to extract) has its own list of candidate offsets. Since the lists associated to different fields can overlap, it is now a global optimization problem to find the set of offsets that maximizes the number of recovered fields. For instance, let's assume that, according to our chain-matching algorithm, three fields of the same data structure can be located, *respectively,* at offsets `{72, 74}`, `{40, 72}` and `{40}`. In this example, since the third field was found to be at offset `40`, we can exclude that the second field can be located at the same offset `40`, and in turn this rules out the possibility of the first to be at offset `72`.

We solve this problem by creating a `z3` model [8] where all the fields and respective candidates are added in the form of constraints. We call these constraints *soft*, in contrast to *hard* constraints that are based on the definition of a structure. In particular, we add *hard* constraints based on the position of a field, because the order of the fields in the source code definition must be respected in the offsets layout (e.g., `cred` < `name`), and we also assert that the first field in a structure is always at offset zero (`next` == 0). Moreover, since pointers have a predictable size on 64 bit machines, we assert that the 8 bytes following a pointer must also respect the definition order ($\bigwedge_{i=1}^{8} next + i < cred$). Similarly, if a field represents a nested structure, then we can count how many pointers (not enclosed in any `ifdef` directive) it contains and use this as a constraint of the minimum distance between this field and the next. Special care is given to unions, since in this case we assume the fields they contain have the same offset. Some of the previous constraints cannot be applied when structure randomization is in place. For example, when this feature is enabled, the first field of

Table 2. The Linux Kernels Used in Our Experiments

Version	Release Date	Configuration	Used Fields	Extracted Fields
4.19.37	04/2019	Debian	234	220 (94%)
4.19.37	04/2019	Debian + RANDSTRUCT	234	194 (83%)
5.6.19	03/2020	Raspberry Pi	227	217 (95%)
4.4.71	06/2017	OpenWrt	236	216 (92%)
3.18.94	05/2018	Goldfish (Android)	239	220 (92%)
2.6.38	03/2011	Ubuntu	226	213 (94%)

a structure might not be at offset 0. Moreover, we also relax other predicates, by changing the arithmetic operators from *less than* ($<$) to *not equal* ($!=$).

A problem with this approach is that if the candidates of a field are wrong and *contradict* the position constraints, then the model become unsatisfiable. To overcome this limitation, when we run into an unsatisfiable model, we explore the solution space by recursively removing a *soft* unsatisfiable constraints.

Finally, the *knowledge* gained from the previous modeling process is added to the system. This new piece of information will most likely satisfy the *dependency* or the *displacement* of other chains that were previously not testable. Hence, we go back and forth between the binary analysis component that resolve the chains and the layout inference component that solves the extracted candidates and constraints until no other chain is available.

8 EXPERIMENTS

To test AutoProfile we collected a number of memory dumps from systems running different Linux kernels. The list of kernels (summarized in Table 2) was chosen to reflect different major versions (including 2.6, 3.1, 4.4, 4.19, and 5.6) and different configurations. In particular, the first experiment was conducted with the latest version of the kernel shipped by Debian. In the second experiment we reused the same configuration, but this time with structure layout randomization turned on. To study how different randomization seeds can impact our approach, we recompiled the kernel 10 times and reported an average value in Table 2. The last four experiments aimed instead to test AutoProfile against less common memory forensics scenarios, when the traditional approach to create a profile would be difficult to apply. For one test we retrieved the kernel used for Raspberry Pi devices, for another test we targeted the kernel used by OpenWrt, a project that targets network devices; in another we recreated a scenario involving a memory dump of an Android device, and for our last test we chose a 10 years old version of the Linux kernel that does not support Clang. While certain of the aforementioned kernels are targeted towards the embedded and IoT world, the current implementation of AutoProfile supports only `x86-64`, and we therefore configure and compile the kernels accordingly. The only architecture-dependent components of AutoProfile are the kallsyms extractor and the symbolic exploration. However, these components are, respectively, based on `Unicorn` and `angr`, and therefore we believe that AutoProfile can be engineered to support other architectures as well.

To run our experiments we downloaded the kernel sources and configurations from the respective repositories. Each kernel was compiled twice, one time to be used in our experiments and the other to perform our source-code analysis. The first version was configured with the configuration shipped with the distribution, and compiled it with a supported version of `gcc`, while the second version was instead configured with `allyesconfig` and analyzed with our Clang compiler plugin. We then proceeded by installing the first version in a `QEMU` virtual machine, booting the machine

and acquiring an atomic memory dump using the QEMU console. Moreover, we also used the first version to manually create a *ground truth* Volatility profile. We were able to create this profile for each experiment except for the ones using `RANDSTRUCT`. While we empirically checked that the kernel code was correctly reflecting this option and that the randomization seed was present in the kernel tree—the debugging information did not reflect the change in the structures layout. We later discovered this to be a known issue already discussed by several researchers in online forums [1, 7]. It is still unclear if the problem is due to a bug in `gcc` or in the randomization plugin, but in any case the erroneous information prevents Volatility from generating a profile even when the randomization seed is available.

For this reason, to generate the ground truth required to perform this test, we developed a custom kernel module that, using inline assembly statements, loads in a specific register the offset of a field using the `offsetof` compiler builtin. Therefore, by compiling and disassembling this module, we were able to write a script to automatically extract the correct offset of every field used by Volatility.

8.1 Analysis Time

Building the profiles using our automated system took approximately eight hours in each experiment. The first phase was the fastest and the only one that depends on the size of the memory dump. Nevertheless, since the kernel is usually loaded in the lower part of the physical memory, our prototype required few seconds to analyze 2GB of memory and retrieve all kernel symbols. The static analysis performed in Phase two took three hours on a eight-core machine. In this phase, most of the time is spent compiling the kernel configured with `allyesconfig` and extracting the access chains using our compiler plugin. Finally, the exploration of kernel functions using `angr` and the generation of the final profile is the most time-consuming phase of our experiments and took in average five hours on a cluster of 64 cores.

8.2 Results

The fourth column of Table 2 shows how many unique fields are used by Volatility for the given image. The value range from 227 to 239 but, quite surprisingly, the intersection of these fields counts more than 180 elements. This means that, even if new features frequently land in the kernel tree, a large fraction of fields used by memory forensics is not affected by the kernel development. These fields are mostly related with process management (e.g., `task_struct`), process memory (e.g., `mm_struct` and `vm_area_struct`), and filesystem information (e.g., `dentry` and `file_operations`). The last column of Table 2 shows instead how many fields AutoProfile was able to correctly extract from the memory dump. The recovery rate ranged from 83% to 95%, but this value alone does not tell us much about how many Volatility plugins are working with the extracted profile. In fact, in most of the cases it is enough that one field was wrongly extracted to undermine the result of an entire plugin.

To answer this question, Table 3 breaks down, for each plugin, the number of fields that were correctly located by AutoProfile and the number of fields for which we extracted a wrong offset. Unfortunately, it is not sufficient to compare the list of fields accessed by a plugin to tell which plugin is correctly supported by our profile. For example, our instrumented version of Volatility reports that the plugin `linux_pstree` accesses the field `gid` of `struct cred` but this information is neither used in the analysis nor displayed to the analyst. Therefore, we decided to compare two runs of Volatility against the same memory dump: one by using the profile extracted by AutoProfile and the other by using the one we manually created. The result of this comparison is shown in "Working" columns in Table 3. Each cell represents whether our profile contains all the necessary information for a given plugin (●) or not (○). In addition, in certain cases it is possible

Table 3. Column **S** Reports the Status of A Plugin: Symbol ● Denotes a Plugin is Working, ◑ Partially Working, ○ not Working, and — not Supported by Volatility

Volatility Plugin	Debian			RANDSTRUCT			Raspberry PI			Openwrt			Android			Ubuntu		
	S	**C**	**W**	**S**	**C**	**W**	**S**	**C**	**W**	**S**	**C**	**W**	**S**	**C**	**W**	**S**	**C**	**W**
`linux_arp`	●	11	0	○	8	3	●	11	0	●	11	0	●	12	0	●	12	0
`linux_banner`	●	0	0	●	0	0	●	0	0	●	0	0	●	0	0	●	0	0
`linux_check_afinfo`	●	10	0	○	6	4	●	10	0	●	40	0	●	42	0	○	35	4
`linux_check_creds`	●	4	0	●	4	0	●	4	0	●	4	0	●	4	0	●	4	0
`linux_check_fop`	○	73	3	○	70	9	○	69	5	○	73	1	●	73	0	●	65	0
`linux_check_idt`	●	0	0	●	0	0	●	0	0	●	0	0	●	0	0	●	0	0
`linux_check_modules`	●	11	0	○	7	1	●	11	0	●	10	0	○	9	1	●	10	0
`linux_check_syscall`	○	31	0	○	30	1	○	31	0	○	30	0	●	30	0	●	29	0
`linux_check_tty`	○	13	1	○	11	3	●	14	0	○	12	1	●	13	0	●	12	1
`linux_cpuinfo`	●	2	0	◑	1	1	●	2	0	●	2	0	●	2	0	●	2	0
`linux_dmesg`	○	0	0	○	0	0	○	0	0	○	0	0	○	0	0	●	0	0
`linux_dump_map`	●	9	0	●	9	0	●	9	0	●	9	0	●	9	0	●	9	0
`linux_dynamic_env`	●	6	0	●	6	0	●	6	0	●	6	0	●	27	0	●	26	0
`linux_elfs`	●	24	0	◑	23	1	●	23	0	●	23	0	●	24	0	●	22	0
`linux_enumerate_files`	●	24	0	●	24	1	●	24	0	●	24	0	●	24	0	●	23	0
`linux_find_file -L`	●	24	0	●	24	1	●	24	0	●	24	0	●	24	0	●	23	0
`linux_getcwd`	●	16	0	●	16	0	●	16	0	●	16	0	●	16	0	●	16	0
`linux_hidden_modules`	○	8	0	○	8	0	○	8	0	○	7	0	●	7	0	●	7	0
`linux_ifconfig`	◑	11	1	◑	11	1	○	11	1	●	12	0	●	12	0	◑	11	1
`linux_info_regs`	—	—	—	—	—	—	—	—	—	●	11	0	○	9	2	●	11	0
`linux_iomem`	●	5	0	●	5	0	●	5	0	●	5	0	●	5	0	●	5	0
`linux_keyboard`	●	1	0	●	1	0	●	1	0	●	1	0	●	1	0	●	1	0
`linux_ldrmodules`	●	30	0	◑	28	2	●	29	0	○	27	2	●	30	0	●	28	0
`linux_library_list`	●	9	0	●	9	0	●	9	0	●	8	0	●	9	0	●	9	0
`linux_librarydump`	●	9	0	●	9	0	●	9	0	●	9	0	●	9	0	●	9	0
`linux_list_raw`	○	32	0	○	27	3	○	32	0	○	4	2	○	32	0	○	31	0
`linux_lsmod`	●	6	0	●	6	0	●	6	0	●	5	0	●	5	0	●	5	0
`linux_lsof`	●	24	0	◑	22	2	●	24	0	●	24	0	●	24	0	●	23	0
`linux_malfind`	●	15	0	○	14	1	●	15	0	○	13	2	●	15	0	●	15	0
`linux_memmap`	●	6	0	●	6	0	●	6	0	●	6	0	●	6	0	●	6	0
`linux_moddump`	○	8	3	○	7	4	○	8	3	○	8	3	○	8	3	○	7	3
`linux_mount`	●	20	0	◑	20	1	●	20	0	●	20	0	●	20	0	●	19	0
`linux_netscan`	◑	16	1	◑	15	2	○	16	1	◑	15	2	◑	15	2	◑	15	2
`linux_netstat`	○	29	1	○	27	3	○	29	1	○	29	1	○	29	1	○	29	1
`linux_pidhashtable`	○	18	4	○	19	3	○	21	1	○	21	3	○	18	6	◑	20	1
`linux_plthook`	●	25	0	●	24	1	●	24	0	●	22	0	●	25	0	●	22	0
`linux_plthook -a`	●	25	0	●	24	1	●	24	0	●	22	0	●	25	0	●	22	0
`linux_proc_maps`	●	37	0	◑	34	3	●	37	0	●	35	2	●	37	0	●	35	0
`linux_proc_maps_rb`	●	39	0	◑	35	4	●	39	0	●	37	2	●	39	0	●	37	0
`linux_procdump`	●	7	0	○	6	1	●	7	0	●	7	0	●	7	0	●	7	0
`linux_psaux`	●	11	0	●	11	0	●	11	0	●	11	0	●	11	0	●	11	0
`linux_psenv`	●	8	0	●	7	1	●	8	0	●	8	0	●	8	0	●	8	0
`linux_pslist`	◑	13	3	◑	14	2	●	16	0	◑	15	1	◑	13	3	◑	12	1
`linux_psscan`	◑	10	1	◑	10	1	●	11	0	●	11	0	●	11	0	◑	12	1
`linux_pstree`	●	11	0	●	11	0	○	9	2	○	9	2	○	9	2	●	11	0
`linux_psxview`	◑	16	1	◑	16	1	◑	16	1	◑	18	2	◑	17	3	●	20	0
`linux_recover_fs`	○	34	1	○	35	1	○	35	0	○	31	4	○	32	3	○	33	1
`linux_threads`	●	6	0	●	6	0	●	6	0	●	6	0	●	6	0	●	6	0
`linux_tmpfs -L`	●	20	0	●	20	1	●	20	0	●	20	0	●	20	0	●	19	0
`linux_tmpfs -D`	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	○	30	0
`linux_truecrypt`	●	3	0	●	3	0	●	3	0	●	3	0	●	3	0	●	3	0
Total Working Plugins	**34**			**22**			**36**			**34**			**40**			**40**		

Columns **C** and **W** represents the number of fields used by a plugin, that were correctly and wrongly extracted by AutoProfile, respectively.

that, even if one field was not correctly extracted, the plugin is still able to function with reduced functionality (◑). For example, this happens in 3 out of 6 cases for the `linux_ifconfig` plugin where the name and the IP address of a network interface are listed correctly by our profile, but the MAC address has a value not associated with any vendor. Finally, cells containing the dash sing (`--`) denotes that the corresponding plugin was not supported on the kernel under analysis or that it crashed while using the ground truth profile. The only exception to this approach are those

plugins that do not produce any output in our tests. For example, the `linux_malfind` plugin searches for traces of malware infection but if the machine is not infected—like in our experiments—then the plugin does not produce any output. Similarly the plugin `linux_hidden_modules` searches for kernel modules that were un-linked from the modules list. Therefore, for these cases we resorted to check that the offsets of all fields accessed by the plugins were correctly recovered to determine whether the plugin was supported or not by our profile. We use the same comparison metrics for the randomized tests and report an average value by counting how many plugins are supported by the extracted profiles.

Overall, on the non-randomized memory dump, between 68% (for Openwrt) and 78% (for Ubuntu) of the plugins worked correctly with our profile, and between 74% (for Openwrt) and 88% (for Ubuntu) of the plugins had at least reduced functionality. In particular, the profile automatically created by AutoProfile was able to support many plugins which are fundamental for a forensics analysis. This include the support to extract the list of running process—except their starting time—and many related information such as their memory mappings, credentials, opened files, and environment variables. Moreover, our profile can be used to successfully list the content of `tmpfs` and to list the loaded kernel modules.

In other cases, AutoProfile was not able to recover the right offsets for the required fields. For instance, the field `num` of `struct tty_driver` prevents the `linux_check_tty` plugin to run in the Debian experiment. However, even if we report the plugin as not supported in the results of Table 3, in practice an analyst could often overcome this limitation by testing different profiles. For instance, for the previous field, our system extracted two possible offsets, one of which was the correct one. In other words, our technique could be used to generate two profiles and simply ask the analyst to try both during the analysis. Overall, the percentage of fields for which AutoProfile extracts a wrong or empty model is 4%, while the number of models that contains two or three offsets—one of which is the correct one—accounts for almost 40% of the missing fields.

Another interesting observation is that in rare cases plugins are reported as not functional, even if all the involved fields were correctly extracted by our framework. By carefully inspecting these cases, we discovered that some of them also require to know the total size of certain structures. For example, the `hidden_modules` plugin requires the size of the `latch_tree_root` structure. While AutoProfile can find the offset of the last field, in some cases this may not be sufficient, as discussed in Section 10.

Finally, the experiment on the randomized kernel shows that the *hard* constraints play an important role in our system. More than 140 of the 234 fields used by Volatility are contained in structures affected by layout randomization, and currently AutoProfile is able to correctly extract the offset of 79% of them.

8.3 Chains Extraction

Table 4 shows detailed statistics about our analysis. Because of space constraints we could not include all 230 fields and we decided therefore to limit the table to the fields belonging to `mm_struct` and `vm_area_struct`. For each field, the table reports the number of chains extracted in Phase two (**Total**), the number of chain explored in Phase three (**Expl.**), the number of chain that contained at least one dependency or displacement not satisfied (**Dep.** and **Disp.**), and finally the number of offsets generated by AutoProfile (a ✓ sign means that the tool identified the right offset, while a number means that the model was not only containing the correct offsets, but also other possible candidates). There are several interesting information that can be deducted from this table. First of all, both the `mmap` and the `vm_start` fields were never explored, because they are the first field of the respective structures and thus their offset (zero) was automatically deducted. Moreover, it

Table 4. An Excerpt of the Fields used by Volatility and Some Statistics Associated to their Exploration

	Step 1					Step 2				
	Total	**Expl.**	**Dep.**	**Disp.**	**Model**	**Total**	**Expl.**	**Dep.**	**Disp.**	**Model**
mm_struct→ mmap	—	—	—	—	✓	—	—	—	—	✓
mm_struct→ arg_end	5	3	2	0	✓	2	2	0	0	✓
mm_struct→ arg_start	6	3	3	0	✓	3	3	0	0	✓
mm_struct→ brk	9	2	7	0	✓	7	7	0	0	✓
mm_struct→ context	34	2	19	13	✓	32	0	0	32	✓
mm_struct→ env_end	5	3	2	0	✓	2	2	0	0	✓
mm_struct→ env_start	5	3	2	0	✓	2	2	0	0	✓
mm_struct→ mm_rb	13	7	6	0	✓	6	6	0	0	✓
mm_struct→ owner	8	3	5	0	✓	5	5	0	0	✓
mm_struct→ pgd	77	64	13	0	✓	13	11	2	0	✓
mm_struct→ start_brk	8	1	7	0	4	7	7	0	0	✓
mm_struct→ start_code	5	4	1	0	2	1	1	0	0	2
mm_struct→ start_stack	7	1	6	0	✓	6	6	0	0	✓
vm_area_struct→ vm_start	—	—	—	—	✓	—	—	—	—	✓
vm_area_struct→ vm_end	158	127	31	0	✓	31	22	9	0	✓
vm_area_struct→ vm_next	57	48	9	0	✓	9	8	1	0	✓
vm_area_struct→ vm_mm	135	126	9	0	✓	9	5	4	0	✓
vm_area_struct→ vm_flags	198	180	18	0	✓	18	15	3	0	✓
vm_area_struct→ vm_pgoff	100	92	8	0	✓	8	6	2	0	✓
vm_area_struct→ vm_file	130	124	6	0	✓	6	5	1	0	✓

shows the first two iterations of our recursive approach. For example, the model of start_brk contained four candidates at the end of the first step because some chains were not analyzed as they depended on the offset of other fields that were still unknown. However, at the second iteration AutoProfile was able to analyze seven more chains, and that additional information was sufficient to narrow down the choice to a single, correct, offset.

8.4 Comparison with Past Attempts

The first approach to extract a valid profile from a memory dump was presented by Case [3], and subsequently refined by Zhang et al. [48]. Unfortunately, neither of these papers reports how many structure fields they were able to extract. Moreover, both approaches target a restricted number of manually-picked kernel functions to extract a field's offset. This design restricts the applicability of these techniques. For instance, it does not deal with cases where a target function was inlined by the compiler, and the list of target functions must be kept in par with the kernel source code. In comparison, AutoProfile is able to automatically deal with these situations, and once we completed the development *no* changes had to be made to analyze any of the evaluated kernels. Finally, Zhang's approach to extract the kernel symbols does not support kernels randomized with KASLR.

A second attempt to solve this problem was ORIGEN [11]. While this article reports a precision of 90%, it is difficult to draw conclusions on the effectiveness of this tool, because it was tested on only 6 fields (5 fields of task_struct, and 1 of mm_struct). Unfortunately, as we highlighted with our experiments, nowadays memory forensics uses hundreds of structure fields. Moreover, ORIGEN is *heavily* based on a dynamic labeling phase, where instructions reading or writing structure fields are collected. This dynamic phase is problematic in real-world investigations: on one hand, tracing a kernel running in production might undermine its stability, on the other, we are not aware of any solution that is able to extract *and* run a kernel from a memory dump.

9 RELATED WORK

Type inference on binary code has been a very active research topic in the past twenty years. In fact, the process of recovering the type information lost during the compilation process involves several challenges and can be tackled from different angles. The applications that benefit from advances in this field are the most diverse, including *vulnerability detection*, *decompilation*, *binary code reuse*, and *runtime protection mechanisms*. Recently, Caballero and Lin [2] have systematized the research in this area, highlighting the different applications, the implementation details, and the results of more than 35 different solutions. Among all, some of these systems are able to recover the layout of records, and in some cases to associate a primitive type (for example, `char`, `unsigned long`..) to every element inside a record. Examples of these systems are `Mycroft`, `Rewards`, `DDE`, `TDA`, `Howard`, `SmartDec`, `ObjDigger`, and `dynStruct` [13, 18, 23, 25, 26, 36, 37, 42]. Unfortunately, these approaches have limited applicability to our problem because they fundamentally answer a different question. While AutoProfile tries to retrieve, for example, the offset of a specific field `X` inside an object `Y`, previous approaches were instead interested in reconstructing the types of the fields inside `Y`. There is an important difference between being able to locate a particular integer field (e.g., a process identifier) among a dozen of other integer fields within the same data structure —which is the goal of our article—from pinpointing that a field at a particular offset in a data structure is an integer—which is what previous techniques were designed for. For example, `task_struct` contains more than 60 integers and 30 `unsigned longs`. Therefore, even assuming a perfect accuracy of the aforementioned systems (which is far from their real results), an analyst would be left with dozens of possible choices. As this is for just a single field, the process should then be repeated hundreds of time. Finally, many previous approaches assume they can run dynamic analyses on the target program, for example, to collect execution traces. Unfortunately, this represents a great challenge in our current scenario because resurrecting a kernel from a memory dump is not as straightforward as executing a userspace program.

An orthogonal approach to structured memory forensics is memory carving, where pattern matching techniques are used to locate kernel structures. The different approaches presented in literature can be roughly divided in two different categories. On the one hand, we have solutions that focus on generating constraints at the field level. For example, Dolan-Gavitt et al. [10] proposed a system to find the *invariants* of a kernel structure—i.e., those fields which cannot be tampered by rootkits without affecting the stability of the operating system. The authors then used this information to automatically generate signatures for a given structure. Dimsum [20] uses instead a mix of boolean constraints generated from the definition of a data structure and then applies some probabilistic inference to match data structures in *dead* memory. On the other hand, we have techniques that rely on points-to relations between kernel objects to generate graph-based signatures [12, 21, 43]. A common problem of all these previous approaches is that they require to build their model for each target OS/kernel the analyst wants to analyze [39]. But again, at least when targeting the Linux kernel, this is only possible if the models were built with a kernel similar to the one under analysis. To overcome this limitation, Song et al. [39] recently presented DeepMem. This approach is divided in two stages. In the first one, the *training* stage, a Graph Neural Network model is trained by using several different memory graphs, a labeled representation of a memory dump. Then, in the *detection* phase, the neural network model accepts an unlabeled memory graph and classifies it. Using this machine learning approach, DeepMem is able to automatically learn the features of a kernel object across different operating system versions. Unfortunately, even DeepMem does not solve our problem. In fact, its memory graph relies on the concept of *segments*—that represent contiguous chunks of memory between two pointer fields. However, the presence of *ifdefs* or the use of structure randomization change the distance between two pointer fields, thus breaking the DeepMem segments.

Recently, the need to recover kernel structure layouts has also manifested in areas different from memory forensics. For example, Pustogarov et al. [29] solved the problem of analyzing Android device drivers of an *host* kernel, by loading them in a second *evasion* kernel running inside QEMU. In order to correctly load the driver, the layouts of the structures `device` and `file` must match between the two different kernels. The authors solved this problem by hand-picking a few kernel functions that access the aforementioned structures. Then, by extracting and comparing the function's binary code from the two kernels, they are able to recompile the evasion kernel after adjusting its configuration.

Finally, the area of **Virtual Machine Introspection** (**VMI**) has been quite flourishing in the past decade [17]. Systems such as Virtuoso [9], VMST [14], and HyperLink [46] are all able to extract different information from a running virtual machine but also require to operate from the vantage point of the **virtual machine monitor** (**VMM**).

10 DISCUSSION AND FUTURE WORKS

In this section, we discuss the limitations and some potential improvements to our approach.

`KALLSYMS_ALL` – Through our experiments, we assume that the kernel was compiled with the configuration option `KALLSYMS_ALL`. When this configuration is enabled, `kallsyms` does not only contain kernel functions but also the address and the name of kernel global variables. Fortunately, a subset of global variables—precisely those variables which are exported—can still be recovered from the candidate `ksymtab`, independently from this configuration option. This configuration is enabled by default in all major distributions, and in four out of six of the images we used in our experiments, but we acknowledge that it might not always be the case.

To assess the impact of the possibly missing information, we run our experiments twice: once with `KALLSYMS_ALL` enabled and once without. The results show that the percentage of correctly extracted fields is similar between the two experiments. However, some of the missing global variables were later required by Volatility to apply the generated profile to a memory dump. In fact, at first the framework was not able to run *any* analysis because `init_level4_pgt` (the symbol that points to the kernel page tables) was missing from both the `kallsyms` and the `ksymtab`. Luckily, this can be easily solved for the Linux kernel by extracting a reference to this global variable from the function `startup_64`. Alternatively, more general solutions for this problem also exists, for instance, by employing an algorithm to automatically locate kernel paging structures, as the one proposed in 2010 by Saur and Grizzard [34].

Once the page tables have been located, 30 out of 50 plugins were working correctly. The remaining twenty were still malfunctioning because of a missing global variable—with two variables (`modules` and `mount_hashtable`) responsible for 50% of the errors.

Nevertheless, we believe this problem might be tackled by instructing our compiler plugin to save in which functions these global variables are used and then make use of this information to extract their addresses from the kernel binary code itself. This is also facilitated by the fact that global variables can be easily identified in the code as their address is typically loaded in specific way (e.g., in `x86_64` they are expressed as constants or loaded via `rip` relative addressing).

`CONFIG_IKCONFIG` – This particular configuration option saves the configuration used to build the kernel in the kernel image itself and makes this information available to user-space through the `/proc` filesystem. From a memory forensics standpoint, this means that the configuration file is included in the memory dump and can thus be used to build a valid profile. To the best of our knowledge, none of the existing memory forensics framework tries to recover this information, and none of the major distributions ship a kernel compiled with this option. Nevertheless, since

this information is referenced by a kernel symbol (`kernel_config_data`), could be trivially expand our kallsyms recovery technique to extract the kernel configuration.

Threat Model – One of the most important applications of memory forensics is investigating attacks and malicious behaviors. Therefore, forensics tools must be resilient against attacks that tamper with their inner workings. We argue that any modification to kernel memory done with the intent of tricking AutoProfile to extract a wrong profile, is highly unlikely. First of all, kernels can be hardened against malicious modification of their code and data [27]. Moreover, even if these defenses are not deployed, certain modification might have negative consequence on the stability of the running kernel; something that rootkit authors certainly want to avoid. In particular, the only two pieces of information extracted and used by AutoProfile are the kernel symbols and the kernel code. Tampering with the first can negatively impact any kallsyms user—for example, kernel modules or the `perf` subsystem. On the other hand, modifying the offsets used in kernel instructions will most certainly bring the kernel into an unstable state, or even to a crash, when the modified instructions are executed.

Access Chains Improvements – The operation of accessing structure fields is ubiquitous across the kernel code base and the number of functions which only access a single field of a structure is rather small. For this reason, a major improvement to AutoProfile is to save more details about an access chain. For example, our compiler plugin could save the type of access performed on a field, i.e., if the field is only read or also written. In this way, during the exploration phase, AutoProfile could automatically filter memory accesses belonging to one type or the other. Moreover, when a field is written with a constant defined at compile time, this value could be saved in the access chain and used during the matching process. Finally, another distinctive feature might be the *destination* of a chain to know, for example, if the chain is used as a parameter to another function or used as return value. Overall, we believe that all these new details can drastically reduce the number of candidates extracted from a function and thus improving the layout models.

Extracting the size of a structure – Despite having a correct model for all the used fields, four plugins also require to know the size of certain structures to working properly. A first way to extract this information would be to find the offset of the last field of a structure and adding its size. A problem is that the compiler might have decided to add some padding at the end of this structure, thus the computed value might need some adjustment. However, this does not depend on the user config, but only on the compiler toolchain—and padding is often limited to few values. Moreover, if a global variable has the type of this structure, then the structure size can be deducted from the distance between the following global variable. Also in this case, the compiler might have padded the global variable instance, so minor adjustment are required. Finally, this value might also be present in the kernel binary, when the `sizeof` operator is used.

Volatility Targets – Instead of trying to reconstruct the layout of each and every structure defined in the kernel codebase, in this article, we focused on extracting the offsets of the fields used by a considerable number of Volatility plugins. For this reason, we acknowledge that the list of targeted fields might be not exhaustive or cover every possible forensics analysis that will be developed in the future. On the other hand, we don't believe this is a limitation of AutoProfile itself, because a way to solve this problem is simply to re-run our Phase three analysis whenever a new field is needed.

11 ARTIFACTS

We will share all the artifacts generated by our study to foster more research in this field at the following url: https://github.com/pagabuc/autoprofile. This includes the prototype tool we

developed to generate the profiles, the tool to retrieve the kernel symbols from a memory dump, and all the memory images we used in our experiments.

REFERENCES

[1] GCC Bugzilla. 2018. Bug 84052 - Using randomizing structure layout plugin in linux kernel compilation doesn't generate proper debuginfo. Retrieved November 2020 from https://gcc.gnu.org/bugzilla/show_bug.cgi?id=84052.

[2] Juan Caballero and Zhiqiang Lin. 2016. Type inference on executables. *ACM Computing Surveys* 48, 4 (2016), 65.

[3] Andrew Case, Lodovico Marziale, and Golden G. Richard III. 2010. Dynamic recreation of kernel data structures for live forensics. *Digital Investigation* 7, 1 (2010), S32–S40.

[4] Andrew Case and Golden G. Richard III. 2017. Memory forensics: The path forward. *Digital Investigation*, Special Issue on Volatile Memory Analysis 20 (2017), 23–33.

[5] Ping Chen, Jun Xu, Zhiqiang Lin, Dongyan Xu, Bing Mao, and Peng Liu. 2015. A practical approach for adaptive data structure layout randomization. In *Proceedings of the European Symposium on Research in Computer Security*. Springer, 69–89.

[6] Michael Cohen. 2014. Rekall memory forensics framework. Retrieved October 15, 2021 from www.rekall-forensic.com.

[7] Redhat crash utility Mailing List. 2018. Using crash with structure layout randomized kernel. Retrieved November 2020 from https://crash-utility.redhat.narkive.com/WZYTWez6/using-crash-with-structure-layout-randomized-kernel.

[8] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 337–340.

[9] Brendan Dolan-Gavitt, Tim Leek, Michael Zhivich, Jonathon Giffin, and Wenke Lee. 2011. Virtuoso: Narrowing the semantic gap in virtual machine introspection. In *Proceedings of the 2011 IEEE Symposium on Security and Privacy*. IEEE, 297–312.

[10] Brendan Dolan-Gavitt, Abhinav Srivastava, Patrick Traynor, and Jonathon Giffin. 2009. Robust signatures for kernel data structures. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*. ACM, 566–577.

[11] Qian Feng, Aravind Prakash, Minghua Wang, Curtis Carmony, and Heng Yin. 2016. Origen: Automatic extraction of offset-revealing instructions for cross-version memory analysis. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*. ACM, 11–22.

[12] Qian Feng, Aravind Prakash, Heng Yin, and Zhiqiang Lin. 2014. Mace: High-coverage and robust memory analysis for commodity operating systems. In *Proceedings of the 30th Annual Computer Security Applications Conference*. ACM, 196–205.

[13] Alexander Fokin, Egor Derevenetc, Alexander Chernov, and Katerina Troshina. 2011. SmartDec: Approaching C++ decompilation. In *Proceedings of the 2011 18th Working Conference on Reverse Engineering*. IEEE, 347–356.

[14] Yangchun Fu and Zhiqiang Lin. 2012. Space traveling across vm: Automatically bridging the semantic gap in virtual machine introspection via online kernel data redirection. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy*. IEEE, 586–600.

[15] Mariano Graziano. 2016. ksfinder - Retrieve exported kernel symbols from physical memory dumps. Retrieved November 2020 from https://github.com/emdel/ksfinder.

[16] Christophe Hauser, Jayakrishna Menon, Yan Shoshitaishvili, Ruoyu Wang, Giovanni Vigna, and Christopher Kruegel. 2019. Sleak: Automating address space layout derandomization. In *Proceedings of the 35th Annual Computer Security Applications Conference*. 190–202.

[17] Bhushan Jain, Mirza Basim Baig, Dongli Zhang, Donald E. Porter, and Radu Sion. 2014. Sok: Introspections on trust and the semantic gap. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy*. IEEE, 605–620.

[18] Wesley Jin, Cory Cohen, Jeffrey Gennari, Charles Hines, Sagar Chaki, Arie Gurfinkel, Jeffrey Havrilla, and Priya Narasimhan. 2014. Recovering C++ objects from binaries using inter-procedural data-flow analysis. In *Proceedings of the ACM SIGPLAN on Program Protection and Reverse Engineering Workshop 2014*. ACM, 1.

[19] Jonghwan Kim, Daehee Jang, Yunjong Jeong, and Brent Byunghoon Kang. 2019. POLaR: Per-allocation object layout randomization. In *Proceedings of the 2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. IEEE, 505–516.

[20] Zhiqiang Lin, Junghwan Rhee, Chao Wu, Xiangyu Zhang, and Dongyan Xu. 2012. Dimsum: Discovering semantic data of interest from un-mappable memory with confidence. In *Proceedings of the 19th Annual Network and Distributed System Security Symposium*.

[21] Zhiqiang Lin, Junghwan Rhee, Xiangyu Zhang, Dongyan Xu, and Xuxian Jiang. 2011. SigGraph: Brute force scanning of kernel data structure instances using graph-based signatures. In *Proceedings of the Network and Distributed System Security Symposium*.

[22] Zhiqiang Lin, Ryan D. Riley, and Dongyan Xu. 2009. Polymorphing software by randomizing data structure layout. In *Proceedings of the International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment.* Springer, 107–126.
[23] Zhiqiang Lin, Xiangyu Zhang, and Dongyan Xu. 2010. Automatic reverse engineering of data structures from binary execution. In *Proceedings of the 11th Annual Information Security Symposium*, 5.
[24] Michael Matz, Jan Hubicka, Andreas Jaeger, and Mark Mitchell. 2013. System v application binary interface. *AMD64 Architecture Processor Supplement, Draft v0* 99, (2013), 57.
[25] Daniel Mercier, Aziem Chawdhary, and Richard Jones. 2017. dynStruct: An automatic reverse engineering tool for structure recovery and memory use analysis. In *Proceedings of the 2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering.* IEEE, 497–501.
[26] Alan Mycroft. 1999. Type-based decompilation (or program reconstruction via type reconstruction). In *Proceedings of the European Symposium on Programming.* Springer, 208–223.
[27] Openwall.org. 2020. LKRG - Linux Kernel Runtime Guard. Retrieved November 2020 from https://www.openwall.com/lkrg/.
[28] Fabio Pagani and Davide Balzarotti. 2019. Back to the whiteboard: A principled approach for the assessment and design of memory forensic techniques. In *Proceedings of the 28th {USENIX} Security Symposium ({USENIX} Security 19).* 1751–1768.
[29] Ivan Pustogarov, Qian Wu, and Lie David. 2020. Ex-vivo dynamic analysis framework for android device drivers. In *Proceedings of the Symposium on Network and Distributed System Security.*
[30] Nguyen Anh Quynh and Dang Hoang Vu. 2015. Unicorn-The ultimate CPU emulator.
[31] Nilo Redini, Aravind Machiry, Dipanjan Das, Yanick Fratantonio, Antonio Bianchi, Eric Gustafson, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. 2017. Bootstomp: On the security of bootloaders in mobile devices. In *Proceedings of the 26th {USENIX} Security Symposium ({USENIX} Security 17).* 781–798.
[32] Microsoft Research. 2020. Toward trusted sensing for the cloud: Introducing Project Freta. Retrieved November 2020 from https://www.microsoft.com/en-us/research/blog/toward-trusted-sensing-for-the-cloud-introducing-project-freta/.
[33] Vassil Roussev, Irfan Ahmed, and Thomas Sires. 2014. Image-based kernel fingerprinting. *Digital Investigation* 11, Supplement 2 (2014), S13–S21.
[34] Karla Saur and Julian B. Grizzard. 2010. Locating ×86 paging structures in memory images. *Digital Investigation* 7, 1–2 (2010), 28–37.
[35] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Audrey Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. 2016. SoK: (State of) the art of war: Offensive techniques in binary analysis. In *Proceedings of the IEEE Symposium on Security and Privacy.*
[36] Asia Slowinska, Traian Stancescu, and Herbert Bos. 2010. DDE: Dynamic data structure excavation. In *Proceedings of the 1st ACM Asia-Pacific Workshop on Systems.* 13–18.
[37] Asia Slowinska, Traian Stancescu, and Herbert Bos. 2011. Howard: A dynamic excavator for reverse engineering data structures. In *Proceedings of the Symposium on Network and Distributed System Security.*
[38] Arkadiusz Socała and Michael Cohen. 2016. Automatic profile generation for live linux memory analysis. In *Proceedings of the Third Annual DFRWS Europe (DFRWS'16)*, Volume 38.
[39] Wei Song, Heng Yin, Chang Liu, and Dawn Song. 2018. Deepmem: Learning graph neural network models for fast and robust memory forensic analysis. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security.* ACM, 606–618.
[40] Bradley Spengler. 2006. Grsecurity. *Internet* Retrieved May 27, 2006 from http://grsecurity.net/lsm.php.
[41] Pavel Sviderski. 2016. Universal memory forensic analysis of Android systems. Retrieved November 2020 from https://github.com/psviderski/volatility-android.
[42] Katerina Troshina, Yegor Derevenets, and Alexander Chernov. 2010. Reconstruction of composite types for decompilation. In *Proceedings of the 2010 10th IEEE Working Conference on Source Code Analysis and Manipulation.* IEEE, 179–188.
[43] David Urbina, Yufei Gu, Juan Caballero, and Zhiqiang Lin. 2014. Sigpath: A memory graph based approach for program data introspection and modification. In *Proceedings of the European Symposium on Research in Computer Security.* Springer, 237–256.
[44] VolatilityFoundation. 2021. Volatility profiles for Linux and Mac OS X. Retrieved November 2020 from https://github.com/volatilityfoundation/profiles.
[45] Aaron Walters. 2007. The volatility framework: Volatile memory artifact extraction utility framework. Retrieved March 19, 2015 from https://www.volatilesystems.com/default/volatility.
[46] Jidong Xiao, Lei Lu, Haining Wang, and Xiaoyun Zhu. 2016. HyperLink: Virtual machine introspection and memory forensic analysis without kernel source code. In *Proceedings of the 2016 IEEE International Conference on Autonomic Computing.* IEEE, 127–136.

[47] Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. 2014. Modeling and discovering vulnerabilities with code property graphs. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy*. IEEE, 590–604.
[48] Shuhui Zhang, Xiangxu Meng, and Lianhai Wang. 2016. An adaptive approach for Linux memory analysis based on kernel code reconstruction. *EURASIP Journal on Information Security* 2016, 1 (2016), 14.
[49] Shuhui Zhang, Xiangxu Meng, Lianhai Wang, and Guangqi Liu. 2017. Research on linux kernel version diversity for precise memory analysis. In *Proceedings of the International Conference of Pioneering Computer Scientists, Engineers and Educators*. Springer, 373–385.

Received December 2020; revised June 2021; accepted September 2021