KATANA: Robust, Automated, Binary-Only Forensic Analysis of Linux Memory Snapshots

Fabian Franzen
Technical University of Munich
Munich, Germany
franzen@sec.in.tum.de

Tobias Holl
Technical University of Munich
Munich, Germany
tobias.holl@tum.de

Manuel Andreas
Technical University of Munich
Munich, Germany
manuel.andreas@tum.de

Julian Kirsch
Technical University of Munich
Munich, Germany
kirschju@sec.in.tum.de

Jens Grossklags
Technical University of Munich
Munich, Germany
jens.grossklags@in.tum.de

ABSTRACT

The development and research of tools for forensically analyzing Linux memory snapshots have stalled in recent years as they cannot deal with the high degree of configurability and fail to handle security advances like *structure layout randomization*. Existing tools such as *Volatility* and *Rekall* require a pre-generated profile of the operating system, which is not always available, and can be invalidated by the smallest source code or configuration changes in the kernel.

In this paper, we create a reference model of the control and data flow of selected representative Linux kernels. Using this model, ABI properties, and Linux's own runtime information, we apply a configuration- and instruction-set-agnostic structural matching between the reference model and the loaded kernel to obtain enough information to drive all practically relevant forensic analyses.

We implemented our approach in KATANA[1], and evaluated it against *Volatility*. KATANA is superior where no perfect profile information is available. Furthermore, we show correct functionality on an extensive set of 85 kernels with different configurations and 45 realistic snapshots taken while executing popular Linux distributions or recent versions of Android from version 8.1 to 11. Our approach translates to other CPU architectures in the Internet-of-Things (IoT) device domain such as MIPS and ARM64 as we show by analyzing a TP-Link router and a smart camera. We also successfully generalize to modified Linux kernels such as Android.

CCS CONCEPTS

• **Applied computing** → **System forensics**; • **Security and privacy** → *Operating systems security*.

KEYWORDS

memory forensics, automated profile generation, binary analysis

[1] かたな: *Japanese (single-edged) sword*

RAID 2022, October 26–28, 2022, Limassol, Cyprus

ACM ISBN 978-1-4503-9704-9/22/10.
https://doi.org/10.1145/3545948.3545980

ACM Reference Format:
Fabian Franzen, Tobias Holl, Manuel Andreas, Julian Kirsch, and Jens Grossklags. 2022. KATANA: Robust, Automated, Binary-Only Forensic Analysis of Linux Memory Snapshots. In *25th International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2022), October 26–28, 2022, Limassol, Cyprus.* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3545948.3545980

1 INTRODUCTION

Memory forensics offers unique insights into the internal state of operating systems and userspace programs. Frameworks such as *Volatility* and *Rekall* have shown that it is possible to extract a large variety of information from memory dumps and have proven useful after malware and ransomware infections or to obtain disk encryption keys during governmental investigations [15].

The starting point for any such investigation is a memory dump of the target system, which can be easily obtained from virtual machines. This setting is also known as Virtual Machine Introspection (VMI). However, deriving semantic meaning from a VMI view of a virtual machine is a non-trivial challenge, commonly referred to as the *semantic gap* problem [3]. To bridge this gap, automated approaches use information collected over the lifetime of the virtual machine (e.g., instruction traces [5, 9, 23]) or rely on explicit support by the guest operating system in order to interpret data they obtain through a VMI interface [20]. On regular PCs, bare-metal servers, smartphones, and IoT devices, memory dumps can be extracted by injecting a driver or module into the running operating system or by using hardware debugging interfaces such as JTAG.

Traditional tools like *Volatility* and *Rekall Forensics* do not handle forensic investigations of Linux adequately and their analyses have become outdated over time. These tools extract the necessary information about the structure of the OS using debugging information to form a *profile* containing the memory location and layout of crucial OS data structures. However, debugging information about the target is not always readily available and in those cases these tools cannot be used. Especially Linux imposes unique challenges to a forensic analyst in need of a profile: There is a plethora of kernel binaries with slightly varying behaviors that a separate profile needs to be generated for. This is caused by a myriad of compile-time configurations and the kernel's support for different compilers and compiler versions, each pursuing its own code generation strategy [25]. Consider the current definition of the

`task_struct` structure: In Linux 5.8.14, it contains 66 preprocessor directives influencing the number and position of its members.

In most commercial operating systems, on the other hand, the limited customizability means that there is often an easy path for obtaining additional debugging information. For instance, we can retrieve Windows debugging symbols from the Microsoft symbol server based on an extractable identifier from a memory dump.

In the Linux context, however, most of this information is still encoded in the guest operating system. Even if no explicit support for forensic tools is present and no debugging information is available, *the code of the system itself* still needs to be able to load and unload drivers or to examine stack frame information for crash reports, etc. Our own implementation Katana exploits this in a *code-based* approach to deduce a profile usable for forensic analysis.

Concurrent to our own work, AutoProfile [19] was proposed suggesting a similar *code-based* approach to ours. Furthermore, LogicMem [21] utilizes a *runtime information-based* approach. In contrast to *code-based* approaches analyzing the code in the `.text` segment, only the memory dump of the volatile *runtime data* of the operating system is used (e.g., the task list) in a Prolog inference system to deduct a profile. We will further discuss similarities and differences in Section 7.

The key contributions of our work are as follows:

- We provide an implementation of *code-based* profile generation and release it to the public[2]. In contrast to AutoProfile, our implementation, Katana, is based on Ghidra's intermediate representation *P-Code* and works *across architectures*. We evaluate it on x86-64, ARM64 and MIPS.
- Furthermore, we analyze how the created profile generalizes to the full Linux kernel and modern analysis plugins, while existing works (like AutoProfile and LogicMem) focus on a small set of structures for a limited set of partially outdated *Volatility* analyses.
- We prove that Memory Extraction can also be done in a *binary-only* setting by using a Linux Kernel Module (LKM), as it was possible by using the *LiME* LKM in *Volatility*.
- We perform an extensive evaluation of Katana on 85 self-compiled kernel builds with different configurations across 7 kernel versions, and perform various real-world analyses on 45 different kernels from common Linux distributions including Android. We demonstrate Katana's cross-architectural capabilities by analyzing memory dumps of MIPS-based devices (a TP-Link router and a camera), as well as an ARM memory dump.

2 BACKGROUND

In order to motivate our design decisions for the analysis framework Katana, we first discuss two mechanisms used internally by the Linux kernel to organize the mapping between symbolic names and virtual addresses. Afterwards, we explain the details of *structure layout randomization*; a relatively new (April 2017[3]) security feature of the Linux kernel. Then, we discuss why the wide variety of configuration options of the Linux kernel drastically complicates forensic analysis. Finally, we provide a quick introduction to *P-Code*; the intermediate representation we built Katana upon.

2.1 Volatility and Rekall

Volatility[4] is an open-source framework for memory forensics with support for all three major operating systems (Windows, MacOS, and Linux). In our work, we primarily refer to *Volatility* as a comparison, since it is by far the most commonly used tool, even though it was not designed to work with differing configurations and structure layout randomization. *Volatility* supports a large variety of analysis passes such as listing the currently running processes, loaded kernel modules, active file descriptors and active network connections. Additionally, analyses scanning for known rootkit artifacts and integrity checking are available. Development appears to have stalled since the most recent stable release of *Volatility* 2.6 in 2016, resulting in many of the currently existing analyses failing on recent kernel versions.

In order to function, the *Volatility* framework relies on a "profile", which contains the layout of important kernel structures and the relative location of vital global variables. The analysis plugins then utilize this information to locate and parse the kernel's internal data structures in the memory dump in order to produce their reports. On Linux, these profiles are usually generated from the kernel debugging symbols.

In 2013, *Volatility* was forked to streamline the codebase. This fork became the *Rekall Forensics*[5] framework maintained by Google, but has been abandoned now. While a few analysis plugins might work differently, *Rekall* functions in the same way as *Volatility*.

2.2 Kernel Symbol Table

The symbol table (symtab) contains the virtual addresses of *exported* functions and variables that the kernel provides to loadable kernel modules (LKMs). If an LKM wants to log a message using the exported `printk` function, a lookup in the symtab is performed at module load. As LKMs can be loaded during runtime, complete information about exported symbols must be available during runtime, a circumstance used by Katana.

Note that even for kernels compiled with *all* optional features disabled[6]—i.e., even if LKM support is disabled—the kernel still contains a symbol table. It is essential to Linux's functioning and cannot be removed. The exact layout of the kernel symbol table changed over time, but it remains easily discoverable in a memory dump. Details can be found in Figure 6 in Appendix A.

2.3 Kallsyms

Kallsyms provides another way to resolve symbol names to virtual addresses during runtime. The kernel uses it to augment backtraces with symbol names for KGDB and requires it for Ftrace, Kprobes, and other modern kernel security features like control flow integrity checking and live patching. Kallsyms-enabled kernels contain a list of kernel symbols that is extracted during compilation, compressed, and saved in the data sections of the generated Linux image.

[2] Our tools and pre-generated databases for Linux 3.7 – 5.15 are available at https://github.com/tum-itsec/katana.
[3] https://www.openwall.com/lists/kernel-hardening/2017/04/06/14
[4] https://www.volatilityfoundation.org/releases
[5] http://www.rekall-forensic.com/
[6] `make tinyconfig`

In contrast to the symtab, the kallsyms mechanism is aware of the addresses of non-exported kernel functions. If the configuration option `KALLSYMS_ALL` is enabled, it even includes names and virtual addresses of symbols that reside in the data section. As such, the number of symbols on which kallsyms may provide information is magnitudes larger than what can be learned from symtab.

Code running in the kernel can access the kallsyms system by querying one of two exported functions: `kallsyms_on_each_symbol` (introduced in 2.6.30; 2009) allows code to iterate over all symbols that are stored, while `kallsyms_lookup_name` (introduced in 2.6.4; 2004) does a name-based symbol lookup of the respective address.

This interface has been stable since its introduction, but can be disabled at compile time as it is an optional feature. Fortunately, as we will see in Section 4, most systems leave the mechanism activated, including the symbols in the data section (`KALLSYMS_ALL`).

2.4 Structure Layout Randomization

Since version 4.13, the Linux kernel has offered *structure layout randomization* as an additional security feature. If enabled, the spatial order of members of structures marked with the attribute `__randomize_layout` is shuffled at compilation time. Structures containing only function pointers will always be shuffled, if not explicitly forbidden by using `__no_randomize_layout`. The shuffling is realized as a compiler plugin, which adds an additional optimization pass to the compilation pipeline. Shuffling is implemented deterministically, based on a 256-bit random seed. This design decision enables a shuffled kernel to load kernel modules that were compiled later than the main kernel image, but has the serious drawback that distributions need to publish the random seed, making security gains in general-purpose distributions questionable.

2.5 Kernel Configuration

Because Linux targets a wide variety of use cases, it provides a large number of compile-time switches that can be used to enable, disable, and modify certain features in the kernel. Each of these *Kconfig* variables is available to both the kernel's custom *Kbuild* build system and the source code, where they allow conditional compilation of certain code fragments or source files (e.g., to configure support for specialized hardware). The same system is used to select which features are not embedded in the kernel but are provided through kernel modules, and to configure a large number of other settings ranging from the relatively benign (e.g., the default host name) to critically important (e.g., CPU endianness).

This significantly complicates any analysis of the Linux kernel, because measurements obtained on one configuration may not be valid on another. Moreover, rare combinations of configuration options may affect the system in unexpected ways or reveal subtle bugs [25]. For our purposes, the key differences between different configurations are the functions that are available for analysis and the layouts of structures in the kernel: Since many features add members related to their functionality to core kernel types, many different configuration switches can independently change the layouts of these types. This creates a vast number of possible structure layouts even when randomization is disabled. For example, the presence or absence of the 66 conditionally compiled segments in the `task_struct` structure on Linux 5.8.14 depends on 56 different Kconfig variables.

2.6 P-Code

P-Code is an intermediate representation specifically designed for reverse engineering applications and is implemented by the popular software reverse engineering suite *Ghidra*[7].

Processor-specific instructions are lifted to a corresponding sequence of P-Code operations. This means that analyses built on top of P-Code are architecture-agnostic, as long as the lifting process is implemented for the architecture in question. Currently, Ghidra has support for a broad range of architectures (the current version claims to support 77 architecture variants), including the most popular ones such as x86-64, MIPS, ARM, Sparc, PowerPC, etc.

Another advantage of P-Code is that it greatly simplifies implementation of higher-level analysis: The number of different P-Code operations is limited to around 60, while CISC instruction sets such as x86-64 are much more complex with somewhere between 1000 and 4000 distinct instructions depending on the method of counting.

Ghidra itself uses P-Code internally to implement many of its analyses, the most notable being its decompiler.

3 KATANA

In the following, we discuss a new, binary-only, fully automated approach for performing forensic memory analysis.

3.1 Design Goals

When we built Katana, we placed special emphasis on the provision of forensic analysis capabilities in a post-mortem, binary-only, automated, and robust fashion. Below, we briefly explain each of our design goals.

Post-Mortem Availability In many cases, it is desirable to monitor a production system that has not previously been prepared for forensic analysis. To this extent, our analysis approach needs to be able to operate on a physical memory snapshot containing a vanilla Linux kernel and user space without the presence of special debugging information, the respective kernel configuration, or the `System.map` file. A snapshot of the architectural state of the CPU (containing model-specific registers and control registers) can be present to speed up analysis, but is not strictly required.

Katana supports many different sources of memory snapshots, whether acquired using physical tools (e.g., JTAG), using hypervisor support (e.g., to analyze a compromised VM), or directly from the target system (e.g., `/proc/kcore`). For situations in which we would need Katana's output in order to obtain a memory dump in the first place (e.g., IoT devices without JTAG ports), we provide a kernel-mode memory dumping utility (UDM) that can be used without detailed knowledge about the target system (cf. Section 3.7).

Binary Only All information about the system should be derived from the compiled kernel and the respective data structures. We do not require the source code or the build toolchain of the kernel at analysis time, even if some custom kernel patches are applied. Of course, there are some limits to how many changes Katana will be able to accommodate, but commonly used kernels with such modifications (e.g., Android) are supported out-of-the-box. This

[7] https://ghidra-sre.org/

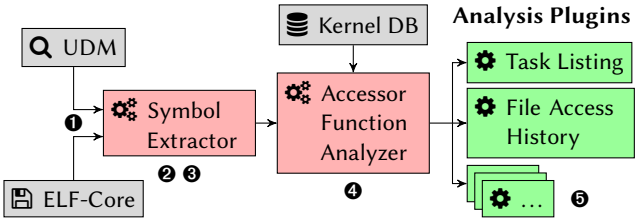

Figure 1: Design of Katana

eases the analysis of embedded devices such as Wi-Fi routers and IoT cameras that do not use "standard" off-the-shelf Linux distributions, where manufacturers may be hesitant to provide source code or debugging information even upon request.

Robust Automated Layout Derivation Katana should be capable of finding all required data structures on its own, without the analyst having to provide external information, such as the kernel version or the layout of core data structures by means of a profile-like database. This includes information such as the virtual and physical KASLR bases, as well as the binary layout of randomized kernel structures. This property offers a highly valuable complement to existing analysis frameworks such as *Rekall* and *Volatility*, as both struggle with the presence of KASLR, structure layout randomization, or configuration changes that propagate to the binary layout of the Linux kernel. The necessary maintenance effort is minimized as Katana adjusts to changes in the kernel source code and configuration without manual intervention, which still remains the predominant method in existing tools to solve this problem [2].

3.2 Overview

Katana operates in four core steps illustrated in Figure 1. Before analysis ❶, a memory dump must be obtained from a virtual machine or a bare-metal device (e.g., via our UDM or as ELF-Core file). ❷ We scan the virtual address space for the kernel's symbol table to obtain a list of functions and their respective virtual addresses. ❸ Using this list, we invoke the `kallsyms_on_each_symbol` iterator function using the popular *Unicorn* emulator [18] to obtain a more complete list of symbols provided by the kernel and all loaded modules. ❹ We match the code within the memory snapshot against a pre-generated database of *accessor functions*, which are known to access or modify specific members of data structures. This matching step is made architecture and instruction encoding independent by first lifting the identified function to P-Code and processing the resulting operation sequence. Suitable candidates for *accessor functions* and information about *invariant members* are generated *a priori* using a custom analysis plugin for the GNU C compiler (GCC) that observes the compilation of a reference kernel. We distribute pre-generated databases for all kernels between versions 3.7 and 5.15 and some older kernels alongside Katana (depicted as *Kernel DB* in Figure 1).

Katana infers the layout of frequently used kernel structures in a completely automated fashion based on the analysis results of the GCC plugin. The analysis is remarkably robust in an overwhelming number of cases, as the most important types are used at various locations in the kernel resulting in a large number of potential accessor functions. Should the analysis fail to obtain the structure layout from any function, it can easily recover using the remaining functions.

The output of step ❹ is called *profile* and could be used to drive different kinds of forensic analysis tasks, e.g., listing all running processes of a system. Katana has its own set of analysis plugins (❺), but the *profile* could also be converted to drive analyses implemented in the *Volatility* or *Rekall* frameworks.

3.3 Database Generation

To find the structure members we are interested in, we introduce the notion of an *accessor function*. As the name indicates, this is a kernel function that *accesses* a certain structure member. Note that while *pure* accessor functions, which only have the *designated* purpose of returning the value of a certain struct member, are a rather uncommon programming construct in the Linux kernel, there are plenty of functions which just *access* the desired data. For example, the `send_sig_all` function in the kernel enumerates all processes while sending a signal to each process. Katana can make use of this fact to derive the location of the process list, sidestepping the *actual* signaling purpose of the function. As such, a myriad of functions can serve as accessor functions in the context of our work. We generate a mapping between accessed structure members and accessing functions using a GCC compiler plugin.

It is important to note that basing the analysis on the output of a GCC compiler plugin does *not* contradict the binary-only design goal of our work. In fact, the source code analysis needs to be performed only *once* on a kernel version reasonably close to that of the target memory dump and can then be reused for other memory dumps. Version and configuration mismatches are only an issue if the code base of the relevant accessor functions changes significantly. Otherwise, such a mismatch may mean missing out on *additional* information that would have been present if changes are taken into account, or a few of the results of the GCC plugin becoming invalid. Neither will significantly worsen the analysis results. We can further improve accuracy by selecting the database matching to the Linux version, which can be extracted from the memory dump without the use of accessor functions. We evaluate the impact of configuration differences in Section 4.

GCC's plugin system allows inserting arbitrary transformation and optimization passes between those that are performed by default. We insert a custom non-modifying compiler pass behind the *einline* pass, at which point many short functions that will always be inlined (e.g., those marked in code with the `always_inline` attribute) are already inlined. At this stage, GCC represents each function using an intermediate language called *GIMPLE*. This representation still closely resembles the structure of the original code, albeit transformed into static single assignment (SSA) form.

We locate accesses of structure members by recursively walking the operand trees in each of the GIMPLE statements and searching for `COMPONENT_REF` (member access) and `MEM_REF` (dereference) nodes. The context in which each node appears reveals how the structure member is used. For example, if the accessed member is passed into a function call, the function argument will refer to an SSA node.

Performing a dataflow analysis across the SSA assignments yields a `COMPONENT_REF` node that refers to the member in question.

In order to support many different kernel versions with different required minimum/maximum GCC versions, the plugin supports any GCC version starting with version 4.8. Using this setup, we generated databases of structure accesses for Linux kernels starting at version 2.6.33, although earlier kernels may also be supported.

3.4 Obtaining Kernel Symbols

With any memory snapshot, the first step is to map virtual addresses to physical memory. If the memory dump already comes with page table information, or if it contains the location of the page tables in register data (such as on x86 via the CR3 register), this is straightforward. Otherwise, we can identify candidate page tables in memory by their recursive structure, validate them based on architecture-specific constraints, and find the kernel page table by choosing the candidate with the largest number of valid kernel memory mappings. Memory dumps created by our toolkit (c.f. Section 3.7) already contain paging information.

Reading the Kernel Symbol Table Armed with the virtual to physical mappings, we scan the virtual address range for the location of the Linux kernel symbol table (`.ksymtab`; described in more detail in Appendix A). By using the `.ksymtab` data, KATANA obtains the locations of exported function starts and global variables in the memory dump.

Augmenting the Symbol List Using Kallsyms We are now equipped with the names and addresses of all symbols the Linux kernel exports using the `EXPORT_SYMBOL` macro. This list can be substantially augmented using additional information provided by the *kallsyms* mechanism.

The `kallsyms_on_each_symbol` function has been present in the kernel since version 2.6.30. It iterates through all symbols known to the *kallsyms* mechanism. For each iteration, this function transfers control to a user-provided callback function. With the information contained in the symbol table, KATANA is able to find the location of `kallsyms_on_each_symbol` in virtual memory and to subsequently invoke it in the context of the memory snapshot. To be tolerant to implementation changes, such as the presence of the `KALLSYMS_RELATIVE_BASE` configuration flag, we execute this function in the *Unicorn* CPU emulator. Inside the emulator, we call `kallsyms_on_each_symbol` with a prepared callback function as a parameter. The emulated code will in turn hand control to the callback function for every entry present in the *kallsyms* database. This allows us to obtain a list of memory locations of all non-static Linux kernel functions. In the event that `KALLSYMS_ALL` is enabled, we are also able to retrieve the addresses of symbols residing in the data segment of the kernel.

In the unlikely case that *kallsyms* is fully disabled in the analysis target, it is necessary to solve the function identification problem using another approach. On its own, the information derived from only the symbol table is insufficient to obtain good results. However, this is not an issue in practice: *kallsyms* is enabled on *all* systems we analyzed (including resource-constrained IoT devices), and a required dependency for other kernel features (cf. Subsection 2.3).

Because using the kallsyms API allows modules to circumvent licensing restrictions in the kernel, the `kallsyms_on_each_symbol` function is no longer exported (but still available internally) starting with Linux 5.7. In this case, we recover its location using the related *kprobes* API, which is ordinarily used to place arbitrary breakpoints in kernel code and returns the address where the breakpoint has been set.

3.5 Structure Layout Reconstruction

In order to reason about the contents of the concrete memory dump, we have to infer the offsets of structure members by matching the machine code of the accessor functions with our pregenerated *Kernel DB*. The matching process uses several properties maintained during the compilation process: First, each accessor function, when not inlined, has to obey the respective architecture-dependent Application Binary Interface (ABI). Second, certain pointer calculations, such as the `container_of` macro, and the use of global variables result in recognizable instruction patterns.
We derived the following analyses from these observations:

ABI to caller function How arguments are passed to a function is defined by the ABI[8]. During analysis, KATANA will exploit this by tainting incoming arguments and tracking every access that happens relative to a tainted register. In the example depicted in Figure 2, we track the value of the `rdi` register (or its P-Code equivalent). Note that its value is first moved to `rbx`, whose value is preserved across function calls (as specified by the ABI). Our taint analysis would now propagate the tainted property to the assigned register (i.e., `rbx`). Then, the actual dereferencing operation happens in line 5 (①).

ABI to callee functions Similarly, calls to non-inlined functions must follow the respective ABI. Therefore, we resolve the address of every called function (e.g., `printk` in Figure 2) inside the function body to its name. If our static analysis of the C code observed that a call to this function contains relevant arguments (i.e., a field access), KATANA tracks the affected arguments backwards by following the observed data flow (e.g., across simple assignments). Inside this slice, we find the last indirect memory access, and consider the displacement used in that instruction as a potential offset for the target field. For example, we can see this occurring in Figure 2, when `t->pid` (②) is passed to `printk`.

ABI from callee functions The return value location is specified by the respective ABI as well. Similar to the way we tracked accesses to parameters, we track pointers that are being returned by tainting the memory location that contains the return value after a successful function call.

ABI from caller function Just as functions that are being called must return their result in a predetermined location, the function we are currently analyzing has to as well. If the currently analyzed function returns a struct member, we will follow the data flow backwards to identify the last indirect memory access that affected the return register. In our imaginary function `foobar`, the pid member of *t* is returned (④). The data flow analysis shows that the value in `eax` (the ABI-specified return register) comes from an access with relative offset 0x3e0.

Access to globals Global variables in the kernel have to be accessed either relative to the current instruction pointer or through

[8]For example, on x86-64, the System V ABI mandates that the first six arguments of a function have to be passed in the registers `rdi`, `rsi`, `rdx`, `rcx`, `r8`, and `r9`.

Source code:

```
int foobar(struct task_struct *t)
{
  printk(KERN_DEBUG "Hey!\n");
  if(t->mm ① != NULL) {
    printk(KERN_INFO "PID: %d IPID: %d\n",
    t->pid ②, init_task->pid ③);
  }
  return t->pid ④;
}
```

① Access via calling function parameter
② Access via ABI to callee functions
③ Access via global properties
④ Return of a local member
● Visible structure offsets
● Composed offset

x86-64 disassembly (Intel syntax):

```
1  push   rbx
2  mov    rbx,rdi
3  mov    rdi,0xffffffff821fb5d4 # "Hey!"
4  call   ffffffff810b9229 # <printk>
5  cmp    QWORD PTR [rbx+0x3e0]①,0x0
6  je     out
7  mov    esi,DWORD PTR [rbx+0x490]②
8  mov    edx,DWORD PTR [rip+0x131d6fb]③ # <init_task+0x490>
9  mov    rdi,0xffffffff821fb5dc # "PID: %d..."
10 call   ffffffff810b9229 # <printk>
11 out:
12 mov    eax,DWORD PTR [rbx+0x3e0]④
13 pop    rbx
14 ret
```

Figure 2: Structure accesses in an example piece of C code, and the result of compiling them to assembly using GCC 9.

the variable's absolute address. While scanning a function, whenever such a reference to the data segment is encountered, we obtain the closest known symbol location before that address and treat the difference (within some reasonable limits) as the offset for the respective field. From the information stored in our database, we infer *which* of the fields was accessed. In Figure 2, `foobar` accesses the `pid` member of the global variable `init_task` (③). This pattern also allows us to deduce the locations of some missing globals if `KALLSYMS_ALL` is disabled.

The `container_of` macro To implement features such as linked lists and hashtables, the Linux kernel often uses the `container_of` macro to access parent objects containing other objects. For example, `task_struct` objects contain the member `tasks` of type `list_head`. In turn, `list_head` contains a member `next`, pointing to the next object in the list. Typically, one would expect `next` to directly point to the following `task_struct` object, however, it actually points to the `tasks` member inside the next `task_struct`. This allows the kernel to implement code that traverses these lists of type without depending on the offset of the `list_head` inside each of the stored objects. However, to access the actual object referenced by the list, the pointer to the next list entry needs to be adjusted by that offset, which is done in kernel sources by using the `container_of` macro.

We observed that in a large majority of cases, Ghidra will emit an INT_ADD P-Code operation with a negative immediate as opposed to a more natural INT_SUB operation with positive immediate. This Ghidra-specific heuristic is consistent across architectures in version 10.0.4 of Ghidra upon which we evaluated Katana. As we observed similar patterns in the underlying machine code, we believe this pattern will also be present in the lifting behavior of future versions. During analysis, we collect these unusual arithmetic operations and attempt to match them with known uses of the `container_of` macro, which we detect in the AST with our GCC plugin. Then, the displacement is likely the inverse of the offset of the member referenced by the `container_of` macro.

Invariant members All techniques listed so far are driven by Katana's ability to identify and exploit ABI features of the machine code during the invocation of a function. However, there are some important members, which are accessed only inside a long call chain of multiple static functions that are inlined into the calling function. Due to additional compiler optimizations, these functions are usually not matchable to the original dataflow for Katana. Instead, these members can be partially recovered by relying on a database of *invariant members*, i.e., structures that are *not at all* covered by `#ifdefs` (and therefore do not change with configuration changes) and are *not* marked for randomization. We extract these members using our compiler plugin and use them in our majority voting process with a double vote, but they can still be overruled given sufficient contradictory evidence.

Specifics of P-Code and the overall algorithm As mentioned earlier, Katana's structural matching is performed on P-Code, which we acquire by accessing the Ghidra API. The classic `[reg+offset*mul]` instruction pattern on x86-64 seen in Figure 2 is decomposed into several P-Code instructions. We track P-Code's equivalent to registers and memory locations (so-called "varnodes") and emulate mathematical calculations on constants in order to accurately follow the data flow. On every `LOAD` and `STORE` operation, we determine if we can match the accessed object to our database. Using P-Code also makes our analysis agnostic to compiler specifics, because the semantics of the lifted representations remain the same and will also be matched by our algorithm. Further details about the lifting process can be found in Appendix C.

Our analyses are complicated by the fact that the compiler applies various optimizations during compilation. The compiler is free to reorder operations, as long as the data flow permits it, to inline functions, and to eliminate dead code. We do not match conditions between the source code and the machine code, meaning that each time the control flow branches, inaccuracies may be introduced. To address this, we perform a weighted majority voting on all offsets recovered for a single struct member. The weights are applied based on the type of analysis that the offset was recovered from and reflect correctness probabilities that we empirically obtained from analyzing the 85 kernels displayed in Table 3. Together with this majority voting, we will see in Section 4 that these transformations do not harm our analysis in almost all cases if the field inside

Algorithm 1: P-Code to source code database matching

```
S ← recovered function symbols;
D ← access database for similar kernel version;
O ← ∅, mapping of members to offsets;
foreach recovered function f ∈ S do
    A ← ∅, ordered set of accesses;
    foreach P-Code instruction i ∈ f do
        if i is a structure member access on some o then
            ▷ Recover source or sink s and offset δ
            s, δ ← taint-tracking(o);
            A ← A ∪ {(s, δ)};
        end
        if i is a register access with fixed offset δ < 0 and
          |δ| < |δ_max| then
            A ← A ∪ {(containerof, −δ)};
        end
    end
    foreach P-Code analysis λ do
        R ← reference accesses from database D(f, λ);
        foreach matching access a ∈ λ(A) do
            ▷ Fetch type t and member m
            t, m ← next(R);
            O(t, m) ← O(t, m) ∪ {offset(a)};
        end
    end
end
```

the structure is accessed frequently enough. Katana may pick an incorrect candidate for a struct offset in a single accessor function, but can correctly vote out the final offset.

A broad overview of our P-Code-to-database matching is depicted in Algorithm 1. We sequentially iterate over the P-Code instruction stream of every recovered function, following direct branches and analyzing both control flows of conditional branches, combining their results and removing duplicates. When we encounter an instruction that could conceivably be a structure member access, we follow our description of the taint tracking information both forwards (where the result of the member access passes to data sinks such as function call arguments) and backwards (to data sources such as global variables and function parameters) as described above, and log the sink or source and the recovered offset in an ordered list of accesses. Candidate accesses using the `container_of` macro are treated similarly, but no taint tracking is necessary. After obtaining the set of accesses for a function, we separately consider each of our analysis passes. Reference accesses from the database (containing type and member information) for that specific analysis pass are matched one-by-one to the accesses obtained from P-Code (which contribute the recovered offset).

3.6 Analysis Plugins

In total, *Volatility* implements 66 different analyses for Linux memory dumps. However, 14 analyses either do not work correctly as per *Volatility's* own source-code comments (e.g., `linux_arp`) or work only on extremely outdated Linux versions. Therefore, creating a profile for these analyses and executing them would not allow for a fair evaluation on modern kernels. Instead, we identified and reimplemented a set of important analyses found in existing work (e.g., in Ligh et al. [15] and *Volatility*) and include them directly into the Katana framework. This design decision also allows us to enable fallback access patterns for complex analyses and to avoid using excessive structure accesses where it is not needed to fulfill the analysis task. These excessive structure accesses tend to happen in *Volatility*'s analyses as its profiles are always correct, whereas for Katana, inaccurate offsets may cause the whole plugin to fail.

We have reimplemented the following analyses in Katana:

- We obtain the current kernel version *banner* from a symbol.
- We extract the kernel *dmesg* ringbuffer logging output from the memory dump. The corresponding Volatility plugin is an example of an outdated plugin, because the internal structures have changed in version 5.10 (released in December 2020) and *Volatility* has not adapted its analysis.
- We derive the list of *modules* currently loaded in the kernel.
- We obtain a *process listing* from the snapshot. This includes both the process names, execution state, and the user ID with whose permissions the process is running. From there, we use the memory mappings stored by the kernel to produce ELF core files containing the virtual address space of each userspace process. This allows us to match memory segments in the snapshot to their respective processes. The resulting core files can then be analyzed using classical reverse engineering, debugging, or forensic analysis tools.
- Based on the process list information, we retrieve information about *environment* variables, a list of currently *opened files*, and currently open sockets and active *network* connections akin to the `netstat` command. Using this information, an analyst can quite accurately reconstruct the state of userspace processes at the time the snapshot was taken.
- We also provide the ability to list all network neighbor tables including the *ARP table*. This allows the analyst to reconstruct a list of IPv4 and IPv6 machines the analysis target was communicating with at snapshot time.
- An analysis walking through the kernel heap's set of allocated and formerly allocated objects is included. Of particular interest is the heap cache containing `dentry` (*d*irectory *entry*) objects. We associated each of these objects with corresponding metadata, which allowed us to build a timeline of file accesses. *Volatility*'s `dentry_cache` plugin performs essentially the same analysis, but has not been updated since 2014 and requires a global symbol that was removed with Linux 3.6 (dated September 2012). Therefore, it operates only on kernels that use the SLAB allocator, while we also support newer kernels with the now-default SLUB allocator. Furthermore, additional security features (i.e., pointer mangling of free list pointers) have been developed since then and are, therefore, not supported.
  This plugin is one example requiring the knowledge of many kernel structures, where our own implementation utilizes an alternative access strategy if a key structure offset could not be recovered.

3.7 Module-based Snapshot Creation

While capturing a memory dump from a virtual machine or via hardware debugging primitives such as JTAG should generally be preferred, these approaches are often highly specific to each device or virtualization environment, and not all devices expose the necessary interfaces for this kind of access.

For analyzing Android devices, for example, it has been an established practice to insert a kernel module to dump the system memory from within the same privilege domain as the operating system. One frequently used solution that takes this approach is the *LiME* toolkit [15, p. 580].

However, building a kernel module generally requires access to the kernel headers, the kernel configuration, and the seed used in structure layout randomization, none of which are available in a *binary-only* analysis setting. Unfortunately, it is also not feasible to precompile *LiME* for the different kernels one may encounter: Linux kernel modules are generally compatible only with the specific version of the kernel for which they were compiled.

To allow also for memory snapshot acquisition in a binary-only setting, the Katana framework includes a module that adapts dynamically to the targeted kernel version. The snapshotting processes is described in the following. First, a custom loader and the LKM are compiled for the target architecture (currently x86-64, ARM64, and 32-bit MIPS systems are supported). Second, the loader and the LKM are transferred to the analysis target. Afterward, the loader will analyze the existing kernel modules on the system and alter the `.modinfo` section of the LKM accordingly, parameterize the LKM with the exact Linux version, and issue the `insmod` system call to insert the LKM. The LKM is designed to avoid direct accesses to structure members that are influenced by kernel configuration options, because their offsets are still unknown at this point in time. Where APIs have changed over the years, we use the injected kernel version information to choose between alternative implementations.

In a last step, our module sends a full memory dump of any supported system to a server on the local network, including the full page table of the CPU on which the memory dump is taken. The server can then carry out the snapshot analysis steps of Katana. Our kernel module is loadable on any Linux kernel since version 2.6.18 (dated September 2006) and enables us to obtain memory snapshots in situations where other tools may not be usable. These situations could arise when the device does not allow direct physical exfiltration, e.g., with an Android phone or a bare-metal server. Such devices usually do not offer debugging interfaces like JTAG.

4 LAB EVALUATION: COMPARISON TO VOLATILITY

For the first part of our evaluation, we create lab conditions (i.e., kernels with debug symbols for ground truth) such that we can quantitatively evaluate the performance of Katana. In consequence, this experiment illustrates the magnitude of the impact of varying kernel structures in practice compared to profiles created heuristically by Katana.

To conduct this experiment, we first assess the accuracy of the structure layouts recovered by Katana in the presence of different randomized kernel configurations, and compare the results with

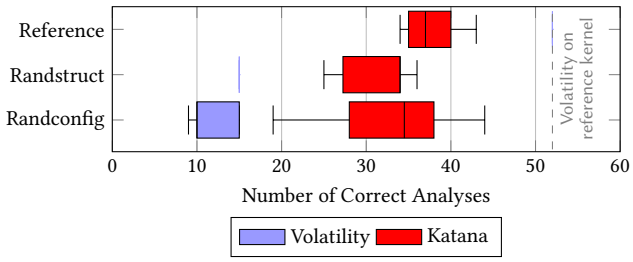

Figure 3: Working Volatility analysis plugins using Volatility and Katana offsets

the fields required by *Volatility* for its various analyses. We utilize *Volatility's* own analyses here for a fair comparison, but exclude 14 analyses that do not work correctly (see Section 3.6).

Experiment Setup To evaluate our automated structure layout recovery, we analyzed 85 different builds of 7 kernel versions and compared the recovered member offsets with the debugging symbols for these kernels. More specifically, to cover a large range of kernel versions, we chose to evaluate Katana on five current long-term support versions, an older, now unsupported, LTS version, and a recent stable kernel version. For each kernel version, we generate a reference database for Katana based on that kernel version's default configuration (*defconfig*). Afterwards, we compile additional variants of each kernel version with modified configurations, and examine how well the reference database generalizes to the changes.

To generate test kernels, we make use of Kbuild's *randconfig* option, which randomizes the features that are enabled in the kernel. While some of these kernels may not be bootable, this produces a set of unbiased configurations, in which features and their corresponding data structures differ from the reference kernel. Using this process, we generate 10 different configurations of varying size on each of the 7 kernel versions we investigated. Support for x86-64 with SMP multithreading, `printk`, and kernel module support is forcibly enabled on all kernels. Where supported, we generate two kernels with the default configuration and structure layout randomization enabled. Furthermore, we save ground truth structure layout information for all kernels. In order to avoid booting each of the 85 kernels — after all, some of them might not even be bootable — we let Katana extract structure offsets from the `.text` sections of the unbooted kernel images.

We identify the structures and members required by *Volatility*'s analyses by inspecting their implementation. For these members, we compare the structure offsets that Katana extracted from the kernel images against the ground truth information. Since Volatility uses the offsets of the reference profile to power the 52 analysis plugins, we have to consider a single differing offset in the target kernel as leading to analysis failure.

Results Across the 70 different kernels with randomized configurations, we correctly recover between 65.4% and 79.3% (average: 73.58%) of all members identified by the reference database. This also includes fields used only internally by drivers or other specialized code that may not even be active on the target system and

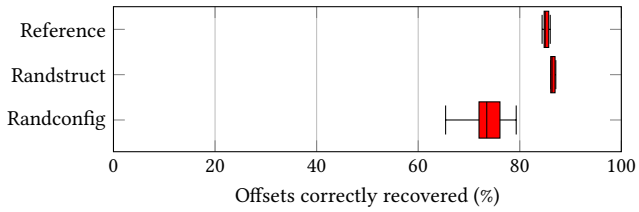

Figure 4: Fraction of correctly recovered structure member offsets using KATANA (all kernel structures)

that would usually be irrelevant for forensic analysis. However, it would be incorrect to exclude these members from the statistical evaluation ahead of time, because they may become interesting for future analyses, and a manual review of all members is impossible. On average, wrong offsets are recovered by KATANA for 15.14% of members (8.1% – 20.8%), and no value can be recovered for the remaining 11.28% of members (6.4% – 16.9%).

Due to the structure differences introduced by the configuration changes, *Volatility* is able to successfully perform only 9 to 15 of the 52 analyses (median: 10), while KATANA correctly recovers enough offsets to perform between 19 and 44 analyses (median: 34.5).

If structure layout randomization is enabled, KATANA's performance improves further: We now correctly recover an average of 86.46% of known offsets. Again, KATANA significantly outperforms Volatility; we manage to run between 25 and 36 analyses (median: 34), while *Volatility* can only perform exactly 15 on each of the 8 tested kernels.

On the reference kernels for each version, *Volatility* (with perfect debugging information) is able to run all 52 analyses. KATANA correctly recovers the offsets of 85.17% of known members on average, and can perform between 34 and 43 of the 52 analyses. Figures 3 and 4 summarize these results; details can be found in Table 3, including the total number of members to which offset recovery relates[9].

In general, we can see that, when ground truth information is not available, KATANA outperforms *Volatility* with regard to the number of correctly performed analyses. Often, KATANA can still execute central analyses including listing processes (`linux_pslist`, supported 41x by KATANA, but by *Volatility* only in the 7 reference kernels) and environment (`linux_psenv`, 68x vs. 7x), network connections (`linux_netstat`, 66x vs. 7x), and module listings (`linux_lsmod` 78x vs. 42x). Where KATANA fails to perform *Volatility*'s analyses, this is most frequently caused by the architectural differences between the two: *Volatility* generally opts to require more structure members than strictly necessary for the analysis. For example, the `linux_check_syscall` plugin verifies the integrity of the system call table, but needs to recover a file from memory in order to do so, even though both syscall numbers and the target pointers are known ahead-of-time. In total, 11 of the plugins perform integrity checking with the goal of detecting the presence of malware (e.g., `linux_check_inline_kernel`), which requires a particularly large number of members. Since analyses are all-or-nothing, i.e., a single misidentified member leads to analysis failure, there is a significant threshold effect: slight differences in recovery can lead to large differences in the number of successfully performed analyses. This can be observed, e.g., in the kernels with structure layout randomization for kernels 5.8.14 and 5.4.70.

[9]Note that since kernels with structure layout randomization contain *un-randomized* debugging information, we instead report the number of members as observed by our GCC plugin during compilation, which for Linux 4.19 and 4.14 differs significantly from the set of members reported by debugging information.

5 REAL-WORLD EVALUATION

To evaluate KATANA in real-world usage scenarios, we cannot rely on the availability of ground-truth information. Instead, we executed all of our implemented analyses (see Section 3.6) on a set of test systems and manually validated the correctness of the output. The set contains a variety of popular Linux distributions, snapshots from non-x86 architectures, a VM infected with malware, and off-the-shelf IoT devices. Our results are described in the following sections.

5.1 Linux Distributions and Variants

To obtain a comprehensive overview of common Linux kernel configurations, we collected a broad variety of popular Linux distributions and variants including Android, with 45 kernels ranging from version 3.9.5 (June 2013) to 5.11.16 (May 2021). We generate snapshots of fully booted QEMU virtual machines using QEMU's `dump-guest-memory` command and analyzed them with all analysis plugins we implemented.

We found that the kallsyms feature is enabled on all of the distributions and release versions examined, and that `KALLSYMS_ALL` (adding global variables and non-exported functions to the set of symbols available through kallsyms) is enabled on all systems except for Debian Jessie and Android 9.

We were able to successfully recover the kernel version *banner*, *dmesg* ringbuffer contents, and the full *module* listing on all memory dumps except for Android 11, where we mis-predicted an offset. In all other cases, KATANA recovered the offsets for the `module` struct, and the address of the internal linkage symbol `modules` successfully. On the other three Android snapshots, KATANA correctly recovered the offsets in the `module` struct, but no modules were loaded.

Our more complex analyses were able to automatically produce task listings and core dumps for the userspace processes as well as the list of open file descriptors for all 45 images. We verified that the generated core dumps matched the memory maps reported by the `/proc` entry for that process on the virtual machines themselves.

Enumerating the processes' environment variables succeeded on all but two images (where empty environment strings were reported). Network-related analyses appeared to be more fragile: listing the ARP table and network connection information both succeeded in 35 of the images. Our analysis of the kernel cache to discover a history of file accesses succeeded on all but seven images taken from standard Linux distributions, but failed on all taken from Android systems.

A listing of the members used by our analysis passes can be found in Appendix B. Furthermore, a detailed listing of all Linux systems showing which of our analyses work can be found in Table 2.

5.2 IoT Devices

Katana is specifically designed to also support forensic analysis of IoT devices. IoT devices are by nature restricted to binary-only approaches, since many vendors do not publish debugging symbols, build configurations, or modifications to the kernel source. We therefore evaluate Katana on two MIPS-based physical devices and one ARM64-based VM (results are also included in Table 2). All analyses were performed cross-architecture against a database generated on x86-64.

Katana was able to perform all of our analyses except the file access history on the memory dump from an ARM64 VM running Linux 4.19[10]. On a TP-Link *TL-WR740N* router and a *Tapo C200* smart camera, Katana managed to recover all data necessary to correctly extract both system (modules, dmesg log, etc.) and process information (including environment variables and open file descriptors). Missing offsets for some rarely used members prevented us from running some of the more complex analyses (e.g., the file access history). The router contains a MIPS 32-bit big endian processor running an extremely outdated Linux 2.6.31; the camera runs Linux 3.10 on a little-endian MIPS 32-bit chip. We attached to both devices using UART and produced memory dumps using Katana's custom dumper.

5.3 Real-World Malware Analysis

In order to prove Katana's practical utility in a realistic post-compromise scenario (malware infection with persistence), we deployed a sample[11] of the RedXOR malware [13], which we randomly selected from VirusShare, on an isolated virtual machine (Ubuntu 18.04, Linux 4.15) and analyzed a memory snapshot using Katana. We were able to successfully recover the malware (and its process image) from the list of processes. From the memory dump, Katana also obtained a timeline of file accesses and the list of files currently open at the time the snapshot was taken (revealing the source of the infection and its persistence mechanism), and observed that there were no currently open network connections or sockets. Our results (obtained in a post-mortem setting) closely matched those obtained both by malware analysis frameworks using live introspection, and manual analysis [13]. Detailed results can be found in Appendix D.

5.4 Performance

We evaluated Katana's performance on the real-world snapshots from Section 5.1. Overall, Katana performs fast enough to be used on a daily basis by an analyst. We exclude the IoT device snapshots from the data below due to their significantly smaller size.

Database Generation During our evaluation, we did not observe a noticeable impact of our GCC plugin on kernel compilation times. This means that creating the database of accessor functions is approximately as fast as normal *defconfig* kernel build (about five minutes on our hardware, depending on the kernel version), and can be performed using multiple cores. We used an AMD Ryzen 2950X with 32GB of RAM and compiled the kernel with eight parallel threads (`-j8`). During normal analysis, this database will be precomputed in almost all cases; we will distribute databases for kernels 3.7 through 5.12 alongside Katana.

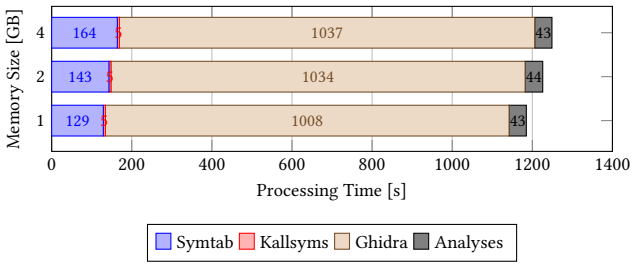

Figure 5: Performance on differently sized memory snapshots

Layout Reconstruction Locating the symbol table took between 32 and 193 seconds depending on the symbol table layout, for an average of 72 seconds. Emulating kallsyms takes between 1.5 and 6.1 seconds; the impact on end-to-end performance is negligible. Recovering the structure layout using Ghidra takes the largest amount of time. On average, processing any of the 45 non-IoT snapshots listed in Table 2 took 883 seconds, with values between 10.5 and 18.5 minutes.

Analyses The time taken for further analysis is generally negligible. All analyses, except those for which userspace paging needs to be reconstructed (environment variables, and process core dump creation), finished within six seconds on all 45 non-IoT snapshots. This matches the speed of comparable *Volatility* analyses. The file access history had both the highest average and maximum runtime, at 2.4 and 6.2 seconds, respectively. The performance impact of recreating userspace paging greatly depends on the amount of memory that needs to be mapped and its page table layout. For example, extracting 4.5GB of core files across 107 processes from the Android 10 snapshot took 169s, while memory dumps with fewer processes were much faster (on average, 32 seconds for 470MB of output).

Impact of Snapshot Size The size of the memory dump itself does not significantly affect the end-to-end analysis time. This is due to the fact that the analysis is strongly affected by *what* data is in the snapshot, rather than *how much*. Figure 5 compares the average performance of each of the layout reconstruction steps for otherwise identical memory dumps of different sizes (averaged over multiple dumps). We see that the overall performance is essentially independent of the size of the snapshot. This is expected for kallsyms emulation and later steps (the size of the input to these steps is bounded by the specifics of the kernel in question, rather than the amount of kernel data). The symbol table search does scale with memory size, but only slightly — more time is spent validating matches than actually scanning the memory.

6 DISCUSSION

Using P-Code allows Katana to target a wide range of devices when an appropriate P-Code implementation is available. We showed its effectiveness for analyzing targets such as Linux distributions (cf. Section 5.1) and IoT devices (cf. Section 5.2). Even a mismatch between the architecture of the generated database of accessor

[10] At the time of evaluation, QEMU did not implement dumping memory of ARM64 guests, so we deferred to using Katana's memory dumper instead (Appendix 3.7).
[11] SHA256: `4f159f6a745752e3211ca1146830c86075fd8f5db60f704605a57db904dcf5c5`

functions and the running kernel still allowed Katana to provide valuable forensic information. In case Katana produces false positives, an analyst should easily be able to tell them apart from accurate information. During our evaluation, we only encountered results that could be misinterpreted as a false negative once: the empty module list on Android kernels.

6.1 Design Decisions and their Impact

In previous iterations of Katana, we based our analysis implementation on the *Capstone* disassembler and its metadata. During the transition to P-Code, we observed a general increase in accuracy of our analyses when compared to the assembly-based approach. Ghidra's ability to derive higher-level meaning from machine code during the lifting process (a feature heavily relied upon by Ghidra's decompiler) greatly improved the quality of our results, and we profited from out-of-the-box support for multiple architectures.

Preserving additional information on *invariant members* and injecting it into the majority vote provides a significant improvement in the recovery rate compared to only using accessor functions. In essence, this combines the *Volatility* approach of assuming offsets never change (which we can guarantee in the case of invariant members as long as the kernel was not modified) with the accessor function approach (which allows us to overrule the former in case our assumptions are not correct). This maintains most of the flexibility of the latter while still benefiting from ground-truth information.

We decided to re-implement some of *Volatility*'s analyses with Katana in order to update the analyses to work with recent kernels and to reduce dependence on technically optional structure members. *Volatility* can assume these are always known because of perfect profiles, but this makes it significantly more difficult for Katana to fully drive *Volatility*'s analyses in cases where almost all, but not all members are recovered correctly.

Instead of relying on ABI characteristics for matching, another naïve approach would be to utilize *BinDiff* to perform an instruction-to-instruction matching at binary level and match the changed offsets to structure members. However, this is particularly sensitive to changes in optimization level or compiler version and to larger code changes (e.g., `#ifdefs`). We found that even if we let *BinDiff* utilize kallsyms information for function identification, a precise instruction-to-instruction matching cannot be performed in the majority of cases (e.g., Figure 9), and mapping instructions to structure members remains a difficult task. Even with debugging information available, DWARF can only map address ranges to line ranges, which is a too coarse to identify individual accesses.

6.2 Rootkit Resilience

Finally, we discuss our ideas in the context of Direct Kernel Structure Manipulation (DKSM) [1]. This usually necessitates a lengthy discourse on the presence of rootkits and the implications for the trustworthiness of any results obtained from an infected system. We acknowledge that Katana could be circumvented by attackers who manipulate the data structures, with potentially disastrous implications for the analysis results. However, we would like to point out that a DKSM attack not only has to change the desired core data structure of the operating system, but also *every single* accessor function operating on such structures to maintain system stability. This is a challenging task for an attacker, especially because updating the kernel code means that Katana will also receive the updated information from the newly generated code. The only way for an attacker to adapt would be to carefully rearrange the basic blocks of the functions inspected by our approach in a way that disassembly logic becomes oblivious to the member locations within structures. This is a non-trivial task, even for an advanced attacker.

Other approaches such as LogicMem [21] are much easier to fool. LogicMem starts its analysis by scanning the memory for the string `"swapper/0"` to find the first process in the task list. A rootkit with full access to kernel memory might create a fake task list and rename the `"swapper/0"` process afterwards. While it is possible to create a second kernel image in memory containing wrong offsets, it is not easily possible to hide the `.text` segment of the operating kernel. In such a case, Katana can detect that two conflicting kernels are placed in memory and warn the analyst.

As Katana's ideas rely solely on the structure of the `.text` segment of the kernel, the structure layout derivation is not affected by attacks like Direct Kernel Object Manipulation (DKOM). Of course, analysis passes that access manipulated objects may still be impeded by DKOM, but not to a greater extent than other tools that analyze memory.

6.3 Real World Impact

We showed that Katana can perform vital analyses of *Volatility* on most major Linux distributions, even without the presence of debugging symbols or in the presence of distributor patches. While the Volatility Foundation maintains a repository of pre-generated profiles for most major Linux distributions, this repository has become outdated and can, by design, not include every custom kernel built for IoT devices. If no ready-to-use profile is available, an analyst might consider switching to Katana in order to avoid the lengthy process of creating a *Volatility* profile. Moreover, those profiles can be generated only for kernels where debugging information is published, which is not the case for distributions with self-compiled kernels (e.g., Gentoo), rolling release distributions where the kernel quickly becomes outdated (e.g., Arch Linux) or IoT devices where manufactures are hesitant to provide access to build toolchains, kernel configurations, and source code. Here, a binary-only analysis is the only viable option.

Katana can also be used to generate profiles suitable for *Volatility* in order to allow reusing existing analyses. However, *Volatility*'s codebase is barely maintained and stuck on Python 2. Its Python 3 successor, *Volatility 3*, does not yet support many Linux analyses. Other binary analysis frameworks like Avatar^2 [17] could also be enriched with forensic information. Taken together, we believe Katana closes an important gap in obtaining forensic information on Linux.

7 RELATED WORK

Our closest competitor is AutoProfile [19], which evolved as concurrent research to our own. It is also based on *code-based* analysis extracting structure offsets from the kernel's code segment. While

the overall derivation approach is similar, there are important differences. First, we base our analysis on Ghidra's P-Code intermediate representation whereas AutoProfile's taint engine is deeply based on x86-64 and currently only supports this architecture. The lead author stated to us on request that even though extending AutoProfile to other architectures would be possible, it would require a substantial amount of engineering effort. Furthermore, we showed Katana's capabilities to perform analysis cross-platform, i.e. to analyze a MIPS-based IoT device with a x86 profile, which is not possible at all with AutoProfile. Second, we do not rely on the repeated use of an SMT solver in the final processing step to resolve conflicting structure offsets. Repeated solving of an SMT problem containing constraints for *all* structures in the kernel will take a substantial amount of time. We assume this to be the key reason for our much quicker analysis (AutoProfile claims an analysis time of 8 hours for a 2 GB memory dump vs 20min in case of Katana).

LogicMem [21] solely analyzes the volatile *runtime data* of the Linux operating system and focuses on finding the task list inside the memory dump. This limits the number of analyses that can be performed to structures that are related to the `task_struct` structure definition. Particularly, an analysis on heap objects of the kernel is not possible. In contrast to Katana, generating the inference rules for every structure involves manual work. Furthermore, LogicMem is unable to handle structure layout randomization.

Besides these two recent proposals, a multitude of methods for monitoring the state of virtual machines (VMs) using the hypervisor have been suggested. These approaches are typically grouped into Guest Assisted, Debugger Assisted, Compiler Assisted, Binary Analysis, and Manual [2]. In the following, for every group, we highlight a few selected approaches.

Manual Approach Despite the fact that manual analysis is a tedious task, it is still chosen by a majority of academic forensic analysis projects [2]. Manual approaches rely on the human to solve the semantic gap problem and as such are naturally not scalable. *VMscope* [11] intercepts all CPU instructions executed on the VM. If an instruction performing a system call is encountered, the corresponding handler is executed. These handlers were implemented manually for every supported system call, each of which derives semantic meaning from the respective system call operation. *RAMPARSER* [10] attempts to recover key fields of a specific set of kernel structs. These key fields are recovered by reverse engineering certain manually picked functions that access or modify them. Additionally, heuristics that are specific to the respective kernel struct are employed to further deduce offsets to key struct fields. This has been shown to work well, but requires substantial manual effort for each specific struct and field that needs to be recovered. Other manually assisted approaches like *Panorama* [27] and *Ekkys* [7] rely on manual reconstruction of important kernel abstractions like processes or files.

In contrast to the previously named tools, there are also signature-based tools that perform the memory snapshot analysis in a bottom-up way, i.e., the analysis does not start from a known global pointer. Instead, a brute-force scan of all available memory is performed in order to detect interesting kernel structures. This memory scanning technique allows detection of kernel objects that have intentionally been hidden by malware, for example, by disconnecting it from the global object graph. However, all signature based detection tools generate their signatures from *known* kernel structure layouts [6, 14]. Most likely this is done to keep the false positive rate low.

Debugger Assisted Approach Besides *Volatility* and *Rekall*, which have already been discussed, also *libVMI*[12] can be put into this category with the same drawbacks. *HookMap* [26] leverages the `System.map` file to obtain the position of kernel symbols. However, the `System.map` file severely limits possibilities of analysis as no knowledge of the layout of kernel structs is contained.

Binary Analysis Approach Binary analysis approaches do not rely on any external information like debugging symbols and instead operate only on the raw binary image. *RAMAnalyzer* [28] starts out the analysis by scanning the binary image for a specific crash information string, which is generated early on in the Linux Kernel boot process. This string contains the position of certain symbols that enable recovery of the page global directory. Access to *kallsyms* then allows recovery of exported and unexported kernel symbols. To conduct analysis on the recovered symbols, structure layouts are identified in a fashion similar to *RAMPARSER* [10]. In contrast to *RAMAnalyzer*, Katana automatically recovers a wide variety of structure fields and, therefore, eases development of future analyses, which might require a completely new set of fields.

BinDiff [29] and *Diaphora* [12] are frameworks that aim to match functions between different versions of the same binary. They do not recover structure layouts by themselves. *ORIGEN* [8] uses *BinDiff*'s binary-to-binary matching in order to translate layout information between kernels. To identify structure accesses ("offset revealing instructions") in the reference kernel, they rely on a combination of static and dynamic analysis (including tracing dynamic allocations of the target types). Then, *ORIGEN* obtains a one-to-one instruction matching using *BinDiff*, and attempts to recover the offsets from the equivalent instruction in the target kernel. However, the large number of different structures makes a manual tagging approach of *all* potentially interesting types an infeasible task, and obtaining full coverage in live execution is even more difficult: E.g., consider a rare access that only happens under specific hardware conditions — ORIGEN's dynamic analysis cannot find the access and, in turn, will not be able to recover the offset. Furthermore, ORIGEN's evaluation on the kernel is limited to the `task_struct`; adding further types seems to require tedious manual work.

Compiler Assisted Approach *InSight* [24] constructs a map of kernel objects by starting with global objects and following the pointer members of each object. However, many code paths in the Linux kernel cast pointers to a different type or perform various other operations to them before actually accessing the pointed to memory. To avoid analysis faults caused by this dynamic behavior, *InSight* performs static code analysis on the kernel source code in order to infer the dynamic behavior of pointer members.

SigGraph [16] utilizes a custom compiler pass to infer the graph structure formed by pointers between different kernel objects of interest. By performing a brute-force scan over the entire memory, it is possible to find instances of the targeted data structures.

Guest Assisted Approach Guest-assisted approaches generally require access to the running system, such that either a program can be installed on the guest or the OS itself can be modified. Therefore, these approaches are not suitable for a post-mortem

[12] http://libvmi.com/

analysis setup. *Virtuoso* [5], *VMST* [9], and *Hybrid-Bridge* [23] try to solve the semantic gap problem by converting an in-guest analysis program to an out-of-guest analysis program, by recording one or many instruction/memory access traces of the overall system. This usually does not allow post-mortem analysis. *TZB* [4] and *PoKeR* [22] similarly record memory accesses for forensic analysis from the hypervisor.

8 CONCLUSION

We presented Katana, a tool for Linux forensics. It derives symbol information of the running Linux system from the *symtab* and extends them with the symbol information made available through the *kallsyms* feature. From there, Katana is able to partially reconstruct the memory layout of central operating system structures in a *fully automated way* so that essential analyses of *Volatility* can be conducted. Our database generation is based on the concept of *accessor functions*, whose disassembly leaks information about the memory layout of kernel structs.

Furthermore, we are the first who conducted a large study on how *code-based* profile generation approaches perform for *all* structures in the Linux kernel using a set of *modern* analysis plugins on an extensive set of 85 different kernels. Another important result of our research is that forensic profiles can generalize from a default kernel configuration to other kernel configurations (like those used by most major distributions) and CPU architectures sufficiently well in order to perform common forensic analysis tasks.

We conclude that the combination of aggregate symbol information and structure layout derivation from compiled machine code is a promising approach for building robust, automated and binary-only analysis tools. Katana, which we release to the public, serves as a prototype implementation of our vision to enhance practical Linux forensics, and hopefully sparks further research ideas.

ACKNOWLEDGMENTS

We would like to thank Fabio Pagani and Marius Muench for the inspiring discussions on our research. Furthermore, we would like to thank the numerous reviewers for their valuable feedback.

REFERENCES

[1] Sina Bahram, Xuxian Jiang, Zhi Wang, Mike Grace, Jinku Li, Deepa Srinivasan, Junghwan Rhee, and Dongyan Xu. 2010. DKSM: Subverting virtual machine introspection for fun and profit. In *2010 29th IEEE Symposium on Reliable Distributed Systems*. IEEE, 82–91.

[2] Erick Bauman, Gbadebo Ayoade, and Zhiqiang Lin. 2015. A survey on hypervisor-based monitoring: Approaches, applications, and evolutions. *ACM Computing Surveys (CSUR)* 48, 1 (2015), 10.

[3] Peter Chen and Brian Noble. 2001. When virtual is better than real. In *Eighth Workshop on Hot Topics in Operating Systems*. IEEE, 133–138.

[4] Brendan Dolan-Gavitt, Tim Leek, Josh Hodosh, and Wenke Lee. 2013. Tappan Zee (north) bridge: Mining memory accesses for introspection. In *2013 ACM SIGSAC Conference on Computer & Communications Security*. ACM, 839–850.

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

[6] 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*. 566–577.

[7] Manuel Egele, Christopher Kruegel, Engin Kirda, Heng Yin, and Dawn Song. 2007. Dynamic spyware analysis. In *Proceedings of the 2007 USENIX Annual Conference (USENIX ATC)*. USENIX Association, 233–246.

[8] 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*. 11–22.

[9] 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 *2012 IEEE Symposium on Security and Privacy*. IEEE, 586–600.

[10] Richard Golden, Andrew Case, and Lodovico Marziale. 2010. Dynamic Recreation of Kernel Data Structures for Live Forensics. *Digital Investigation* 7 (2010), 32–40.

[11] Xuxian Jiang and Xinyuan Wang. 2007. "Out-of-the-box" Monitoring of VM-based High-Interaction Honeypots. In *International Workshop on Recent Advances in Intrusion Detection*. Springer, 198–218.

[12] Joxean Koret. 2015-2021. Diaphora: A Free and Open Source Program Diffing Tool. http://diaphora.re/. Accessed: 2021-06-08.

[13] Joakim Kennedy and Avigayil Mechtinger. 2020. New Linux Backdoor RedXOR Likely Operated by Chinese Nation-State Actor. https://www.intezer.com/blog/malware-analysis/new-linux-backdoor-redxor-likely-operated-by-chinese-nation-state-actor/.

[14] Bin Liang, Wei You, Wenchang Shi, and Zhaohui Liang. 2011. Detecting stealthy malware with inter-structure and imported signatures. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*. 217–227.

[15] Michael Hale Ligh, Andrew Case, Jamie Levy, and Aaron Walters. 2014. *The art of memory forensics: Detecting malware and threats in Windows, Linux, and Mac memory*. John Wiley & Sons.

[16] 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 *Network and Distributed System Security Symposium (NDSS)*.

[17] Marius Muench, Dario Nisi, Aurélien Francillon, and Davide Balzarotti. 2018. Avatar2: A multi-target orchestration platform. In *Proc. Workshop Binary Anal. Res. (Colocated NDSS Symp.)*. 1–11.

[18] Anh Quynh Nguyen and Hoang Vu Dang. 2015. Unicorn: Next generation CPU emulator framework. *BlackHat USA* (2015).

[19] Fabio Pagani and Davide Balzarotti. 2021. AutoProfile: Towards Automated Profile Generation for Memory Analysis. *ACM Transactions on Privacy and Security* 25, 1 (2021), 1–26.

[20] Jonas Pfoh, Christian Schneider, and Claudia Eckert. 2009. A formal model for virtual machine introspection. In *1st ACM Workshop on Virtual Machine Security (VMSec)*. ACM Press.

[21] Zhenxiao Qi, Yu Qu, and Heng Yin. 2022. LogicMem: Automatic Profile Generation for Binary-Only Memory Forensics via Logic Inference. In *Network and Distributed System Security Symposium (NDSS)*.

[22] Ryan Riley, Xuxian Jiang, and Dongyan Xu. 2009. Multi-aspect profiling of kernel rootkit behavior. In *4th ACM European Conference on Computer Systems*. ACM, 47–60.

[23] Alireza Saberi, Yangchun Fu, and Zhiqiang Lin. 2014. Hybrid-bridge: Efficiently bridging the semantic gap in virtual machine introspection via decoupled execution and training memorization. In *Proceedings of the 21st Annual Network and Distributed System Security Symposium (NDSS'14)*.

[24] Christian Schneider, Jonas Pfoh, and Claudia Eckert. 2012. Bridging the semantic gap through static code analysis. *2012 European Workshop on Systems Security (EuroSec)* (2012).

[25] Reinhard Tartler, Christian Dietrich, Julio Sincero, Wolfgang Schröder-Preikschat, and Daniel Lohmann. 2014. Static Analysis of Variability in System Software: The 90,000 #ifdefs Issue. In *2014 USENIX Annual Technical Conference (USENIX ATC)*. USENIX Association, 421–432.

[26] Zhi Wang, Xuxian Jiang, Weidong Cui, and Xinyuan Wang. 2008. Countering persistent kernel rootkits through systematic hook discovery. In *International Workshop on Recent Advances in Intrusion Detection*. Springer, 21–38.

[27] Heng Yin, Dawn Song, Manuel Egele, Christopher Kruegel, and Engin Kirda. 2007. Panorama: Capturing system-wide information flow for malware detection and analysis. In *Proceedings of the 14th ACM Conference on Computer & Communications Security*. 116–127.

[28] 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.

[29] Zynamics. 2021. Bindiff. https://www.zynamics.com/bindiff.html. Accessed: 2021-06-08.

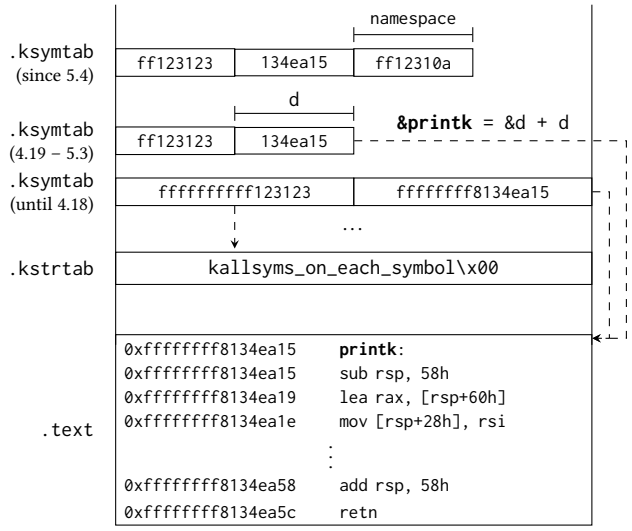

Figure 6: Structure of the symbol table on x86-64

A SYMBOL TABLE LAYOUT

The symtab is separated into two sections compiled into the kernel ELF file. The first section (`.kstrtab` or `.ksymtab_strings`) contains the ASCII representation of the *names* of all symbols separated by a zero byte (optionally compressing strings with matching prefixes). A second section (`.ksymtab`) contains pointers to the symbol names and their actual locations in memory. Figure 6 shows the structure of the symbol table across kernel versions. For space efficiency, on some 64-bit kernels starting with Linux 4.19, 8-byte absolute references inside `.ksymtab` were replaced by 4-byte *relative virtual addresses*: this encoding scheme replaces an absolute value `a` with the relative distance `d` between the target and the storage location of `d`. Since Linux 5.4, symbols are additionally organized in *namespaces* to optionally limit symbol visibility within subsystems[13]. This feature requires an additional 4-byte relative virtual address pointing to the name of the namespace to which the symbol belongs. During analysis, we scan for all possible variants.

B ANALYSIS PASSES

Table 1 lists the members of which we need to reconstruct the offsets for the first few of the analyses described in Section 5. Extracting any task-based information additionally requires the `init_task` symbol (available via the symtab), the module list requires the kallsyms-only `modules` variable (remember that data symbols are only available in presence of the `KALLSYMS_ALL` configuration option). The Linux version banner and *Dmesg* log only require global variables but no structure members.

C P-CODE LIFTING

In the following, we will give a more detailed example of how x86-64 assembly maps to *P-Code*. For this, we depicted a simple imaginary kernel function `pcode` in Figure 7 that accesses a member of its parameter and stores it in a local variable. Afterwards, the

[13] https://lkml.org/lkml/2018/7/16/566

Analysis	Data type	Member
Modules	`list_head`	`next`
	`module`	`list`
	`module`	`name`
Task listing	`cred`	`uid`
	`list_head`	`next`
	`mm_struct`	`pgd`
	`task_struct`	`comm`
	`task_struct`	`cred`
	`task_struct`	`mm` or `active_mm`
	`task_struct`	`pid`
	`task_struct`	`state`
	`task_struct`	`tasks`
Open files	`dentry`	`d_name`
	`dentry`	`d_parent`
	`fdtable`	`max_fds`
	`fdtable`	`fd`
	`file`	`f_path`
	`files_struct`	`fdt`
	`fs_struct`	`root`
	`list_head`	`next`
	`mount`	`mnt` (Linux ≥ 3.3)
	`path`	`dentry`
	`path`	`mnt`
	`qstr`	`name`
	`task_struct`	`comm`
	`task_struct`	`files`
	`task_struct`	`fs`
	`task_struct`	`pid`
	`task_struct`	`tasks`
	`vfsmount`	`mnt_mountpoint` (Linux ≥ 3.3)
Environment	`list_head`	`next`
	`mm_struct`	`pgd`
	`mm_struct`	`env_start`
	`mm_struct`	`env_end` (optional)
	`task_struct`	`comm`
	`task_struct`	`mm` or `active_mm`
	`task_struct`	`pid`
	`task_struct`	`tasks`

Table 1: Structure members used for the first few of the analyses described in Section 5

address of `mmap_sem` inside the local variables object is passed to the function `down_read`. When the first parameter access occurs, it is performed by a simple `mov` instruction with relative displacement to the register `rdi`.

In P-Code, values including registers and memory locations are both represented by *varnodes*: triples consisting of the relevant address space (RAM, registers, constants, and temporary values), an offset, and a size. Operations transform one or more input varnodes into an (optional) output varnode given an opcode such as INT_ADD that dictates the semantics of the operation. Here, the mapping between varnodes and x86-64 registers is displayed in the

Source code:

```
int pcode(struct task_struct *t)
{
    struct mm_struct* mm = t->mm;
    down_read(&mm->mmap_sem);
    return 0;
}
```

x86-64 disassembly:

```
sub   rsp, 0x8
mov   rdi, QWORD PTR [rdi + 0x380]
add   rdi, 0x38
call  0xffffffff810b9229
xor   eax, eax
add   rsp, 0x8
ret
```

Varnode	x86-64 register
(reg, 0x20, 8)	rsp
(reg, 0x38, 8)	rdi
(reg, 0x0, 4)	rax

P-Code operations:

```
(reg, 0x20, 8)  INT_SUB (reg, 0x20, 8), (const, 0x8, 8)
(uniq, 0x30, 8) INT_ADD (reg, 0x38, 8), (const, 0x380, 8)
(uniq, 0x38, 8) LOAD (const, 0x1b1, 4), (unique, 0x30, 8)
(reg, 0x38, 8)  COPY (unique, 0x38, 8)
(reg, 0x200, 1) INT_CARRY (reg, 0x38, 8), (const, 0x38, 8)
(reg, 0x20b, 1) INT_SCARRY (reg, 0x38, 8), (const, 0x38, 8)
(reg, 0x38, 8)  INT_ADD (reg, 0x38, 8), (const, 0x38, 8)
(reg, 0x207, 1) INT_SLESS (reg, 0x38, 8), (const, 0x0, 8)
(reg, 0x206, 1) INT_EQUAL (reg, 0x38, 8), (const, 0x0, 8)
(reg, 0x20, 8)  INT_SUB (reg, 0x20, 8), (const, 0x8, 8)
(------------)  STORE (const, 0x1b1, 8), (reg, 0x20, 8),
                      (const, 0xffffffff810b7320, 8)
(------------)  CALL (ram, 0xffffffff810b9229, 8)
(reg, 0x200, 1) COPY (const, 0x0, 1)
(reg, 0x20b, 1) COPY (const, 0x0, 1)
(reg, 0x0, 4)   INT_XOR (reg, 0x0, 4), (reg, 0x0, 4)
(reg, 0x0, 8)   INT_ZEXT (reg, 0x0, 4)
(reg, 0x207, 1) INT_SLESS (reg, 0x0, 4), (const, 0x0, 4)
(reg, 0x206, 1) INT_EQUAL (reg, 0x0, 4), (const, 0x0, 4)
(reg, 0x200, 1) INT_CARRY (reg, 0x20, 8), (const, 0x8, 8)
(reg, 0x20b, 1) INT_SCARRY (reg, 0x20, 8), (const, 0x8, 8)
(reg, 0x20, 8)  INT_ADD (reg, 0x20, 8), (const, 0x8, 8)
(reg, 0x207, 1) INT_SLESS (reg, 0x20, 8), (const, 0x0, 8)
(reg, 0x206, 1) INT_EQUAL (reg, 0x20, 8), (const, 0x0, 8)
(reg, 0x288, 8) LOAD (const, 0x1b1, 8), (reg, 0x20, 8)
(reg, 0x20, 8)  INT_ADD (reg, 0x20, 8), (const, 0x8, 8)
(------------)  RETURN (reg, 0x288, 8)
```

Figure 7: Example of the mapping between *x86-64* and *P-Code* operations.

table in Figure 7. In the corresponding *P-Code* operations, we can see that the relative displacement is carried out by an INT_ADD operation that stores the result in a temporary varnode. Then, the actual dereference happens in the LOAD operation, with the result ending up in another temporary varnode. Finally, the acquired value is copied to the varnode representing `rdi`. As a result, in order to find the member offset we are looking for, we simply need to search for an INT_ADD operation carried out on the varnode used in the LOAD operation.

Now, imagine our GCC plugin reported that the first parameter to a call to `down_read` was identified as a member access. When we encounter a CALL P-Code operation, we can map the input address back to its symbol and realize that `down_read` was called. Next, we will ask Ghidra for the varnode representing the first parameter in a function call as mandated by the calling convention. Having figured out the varnode of interest, we can now trace backwards to the first P-Code operation writing to this varnode. Quickly, we will arrive at the INT_ADD operation and can identify that the second input varnode represents the immediate offset 0x38.

D MALWARE ANALYSIS RESULTS

In this section, we will present the analysis results obtained from a RedXOR-infected machine (Subsection 5.3) in more detail, and examine how they compare to data obtained from a live sandbox[14] and manual analysis (cf. [13]).

[14] `https://www.virustotal.com/gui/file/4f159f6a745752e3211ca1146830c86075fd8f5db60f704605a57db904dcf5c5/behavior`

Process memory We used Katana to recover the userspace memory mappings of the malware process. While the human-readable part of the output (Figure 8a) only reveals the process PID, name, and UID, we can use the recovered memory layout to dump the memory contents of the process to disk. Analyzing the resulting file in a reverse engineering tool like IDA Pro or Ghidra reveals the malware's functionality. To our knowledge, the sample is not packed or obfuscated, so reverse engineering the sample directly is possible. However, in a forensic post-mortem setting, we may not have access to the sample until we extract it from the snapshot.

Open files Listing the open file descriptors (Figure 8b) reveals that the target process was started from a terminal (standard input is bound to `/dev/pts/0`), but that it redirected its output to `/dev/null`. It maintains an open reference to `/var/tmp/.2a4D53` as file descriptor 3 — which is consistent with both the automated analysis report and the manually reverse-engineered description of the backdoor's behavior.

File access history To discover other files accessed by the malware that are not currently open, we recover `dentry` objects from the allocator's memory cache (see Section 5), and create a timeline of accesses ordered by the access time (extracted from the entry's inode if it was present, and shown in Figure 8c). Besides the aforementioned temporary file, we observe the malware's accesses to its persistence mechanism in `/etc/rc*.d`, where it masquerades as a `polkit` service [13]. In entries accessed by RedXOR, the UID and GID appear corrupted; they take the correct values of 0 (for

```
 PID: 2859 (4f159f6a745752e ) State: 0x1 MM 0xffff92bb80f4c200 UID 0x00
Task struct @ 0xffff92bbb61c2e00 CR3 (0xffff92bb57392000         0x17392000)
```

(a) Extracted process metadata: PID, UID, name, memory mapping, and the root of the page table of the process.

```
2859 (4f159f6a745752e ) [0]: /pts/0
2859 (4f159f6a745752e ) [1]: /dev/null
2859 (4f159f6a745752e ) [2]: /dev/null
2859 (4f159f6a745752e ) [3]: /var/tmp/.2a4D53
2859 (4f159f6a745752e ) [4]: /dev/null
```

(b) Open file descriptors of the RedXOR process.

```
Timestamp                Size  Flags  UID        GID        Inode   Path
Fri Jun 04 2021 18:33:42 53901 m... 0 1314742961 2015032758 1320912 /var/tmp/.po1kitd-update-k
Sun Jun 06 2021 18:48:15 53901 .a.b 0 1314742961 2015032758 1320912 /var/tmp/.po1kitd-update-k
                            26 ma.b 0 1314742961 2015032758 263584  /etc/rc2.d/S99po1kitd-update
                            26 ma.b 0 1314742961 2015032758 263671  /etc/rc3.d/S99po1kitd-update
                            26 ma.b 0 1314742961 2015032758 263672  /etc/rc4.d/S99po1kitd-update
                             0 ma.b 0 1314742961 2015032758 525379  /var/tmp/.2a4D53
```

(c) File access history (note the corrupted UID and GID).

Figure 8: Excerpts of Katana's output analyzing a machine infected with the RedXOR malware.

root) and 1000 (for the default user) for the other entries. To conserve space, we only show the entries related to RedXOR here, but other user activity (related to package updates and extracting the malware sample) is visible as well.

Network connections Live analysis suggests that — at least initially — only a single DNS resolution takes place. On our virtual machine setup, DNS lookup uses a local resolver stub, which shows up on the list of network connections (next to mDNS discovery, DHCP, and the CUPS printer server). However, the snapshot did not capture the shortlived DNS query, because the socket is closed immediately once the DNS server responds.

Katana is also able to correctly recover the process' environment variables and other system information (e.g., the ARP table), though they do not appear to contain any additional indicators of malware infection.

System or Distribution	Kernel	GCC	Kallsyms	Banner	Dmesg	Modules	Task Listing	Open Files	Environment	ARP Table	Network	File History
Ubuntu 21.04	5.11	10.3	✓	✓	✓	✓	✓	✓	✓	✗	✗	✓
Ubuntu 20.10	5.8	10.2	✓	✓	✓	✓	✓	✓	✓	✗	✗	✓
Ubuntu 20.04	5.8	9.3	✓	✓	✓	✓	✓	✓	✓	✓	✗	✓
Ubuntu 19.10	5.3	9.2	✓	✓	✓	✓	✓	✓	✓	✓	✗	✓
Ubuntu 19.04	5.0	8.3	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Ubuntu 18.10	4.18	8.2	✓	✓	✓	✓	✓	✓	✓	✗	✓	✓
Ubuntu 18.04	4.15	7.3	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Ubuntu 17.10	4.13	7.2	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Ubuntu 17.04	4.10	6.3	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Ubuntu 16.10	4.8	6.2	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Ubuntu 16.04	4.4	5.4	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Ubuntu 15.10	4.2	5.2	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Ubuntu 15.04	3.19	4.9	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Ubuntu 14.10	3.16	4.9	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Ubuntu 14.04	3.13	4.8	✓	✓	✓	✓	✓	✓	✗	✓	✓	✓[p]
Ubuntu 13.10	3.11	4.8	✓	✓	✓	✓	✓	✓	✓	✓	✓[6]	✓
Debian 11	5.10	10.2	✓	✓	✓	✓	✓	✓	✓	✓	✓	✗
Debian 10	4.19	8.2	✓	✓	✓	✓	✓	✓	✓	✗	✓	✓
Debian 9	4.9	6.3	✓	✓	✓	✓	✓	✓	✓	✓	✓	✗
Debian 8	3.16	4.9	✓[a]	✓	✓	✓	✓	✓	✓	✓	✗	✗
CentOS 8	4.18	8.3	✓	✓	✓	✓	✓	✓	✗	✗	✗	✓
CentOS 7	3.10	4.8	✓	✓	✓	✓	✓	✓	✓	✓	✓[6]	✓
Fedora 31	5.3	9.2	✓	✓	✓	✓	✓	✓	✓	✓	✗	✓
Fedora 30	5.0	9.0	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Fedora 29	4.18	8.2	✓	✓	✓	✓	✓	✓	✓	✗	✓	✓
Fedora 28	4.16	8.0	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Fedora 27	4.13	7.2	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Fedora 26	4.11	7.1	✓	✓	✓	✓	✓	✓	✓	✗	✓	✓[p]
Fedora 25	4.8	6.2	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Fedora 24	4.5	6.1	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Fedora 23	4.2	5.1	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Fedora 22	4.0	5.1	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Fedora 21	3.17	4.9	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Fedora 20	3.11	4.8	✓	✓	✓	✓	✓	✓	✓	✓	✓[6]	✓
Fedora 19	3.9	4.8	✓	✓	✓	✓	✓	✓	✓	✓	✓[6]	✗[n]

System or Distribution	Kernel	GCC	Kallsyms	Banner	Dmesg	Modules	Task Listing	Open Files	Environment	ARP Table	Network	File History
OpenSuse 15.0	4.12	7.3	✓	✓	✓	✓	✓	✓	✓	✓	✓	✗[n]
OpenSuse 42.1	4.1	4.8	✓	✓	✓	✓	✓	✓	✓	✓	✓	✗[n]
Arch 21-05-01	5.11	10.2	✓	✓	✓	✓	✓	✓	✓	✓	✗	✓
Arch 20-02-01	5.5	9.2	✓	✓	✓	✓	✓	✓	✓	✓	✗[d]	✗
Arch 19-04-02	5.0	8.2	✓	✓	✓	✓	✓	✓	✓	✗	✓	✓
Arch 19-02-01	4.20	8.2	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓[p]
Android 8.1	3.18	4.9	✓	✓	✓	✓	✓	✓	✓	✓	✓[6]	✗
Android 9	4.4	4.9	✓[a]	✓	✓	✓	✓	✓	✓	✗	✗[u]	✗
Android 10	4.14	9.0[c]	✓	✓	✓	✓	✓	✓	✓	✓	✗[u]	✗
Android 11	5.4	12.0[c]	✓	✓	✓	✗	✓	✓	✓	✗	✓	✗
Debian 10 (ARM64)	4.19	10.2.0	✓	✓	✓	✓	✓	✓	✓	✓	✓	✗
WR740N (MIPS32)	2.6.31	4.3	✓[a]	✓	✓	✓	✓	✓	✓	✗	✗	✗
Tapo C200 (MIPS32el)	3.10	4.8	✓[a]	✓	✓	✓	✓	✓	✓	✗	✗	✗

[c] Android kernel compiled using Clang instead of GCC
[a] Kallsyms is enabled, but no KALLSYMS_ALL
[6] IPv6 disabled in the target system
[d] Recovery leads to incorrect destination addresses only
[u] Analysis is unable to recover UNIX sockets
[p] Recovery cannot find SLUB's per-CPU caches of partially filled slabs
[n] Recovery fails due to a missing offset for SLAB's kmem_cache->num

Table 2: KATANA used on multiple Linux memory snapshots

```
FFFFFFFF810712D0  do_send_sig_info
FFFFFFFF810712F5  MOV       EDI, EBP
FFFFFFFF810712F7  MOV       RSI, R12
FFFFFFFF810712FA  MOV       ECX, R13D
FFFFFFFF810712FD  MOV       RDX, RBX
FFFFFFFF81071300  CALL      send_signal
FFFFFFFF81071305  MOV       RDI, qword ptr [RBX + DAT_000006d8] ❶
FFFFFFFF8107130C  MOV       RSI, qword ptr [RSP]
FFFFFFFF81071310  MOV       EBP, EAX
FFFFFFFF81071312  CALL      _raw_spin_unlock_irqrestore
```

```
FFFFFFFF81121EF0  do_send_sig_info
FFFFFFFF81121F30  MOV       RDI, qword ptr [RBX + DAT_000009f8] ❷
FFFFFFFF81121F37  MOV       RSI, R12

FFFFFFFF81121F3A  CALL      _raw_spin_unlock_irqrestore
```

Figure 9: *BinDiff* fails to map the access at ❶ to the equivalent access at ❷, instead claiming the instruction was removed.

	Ref.	1	2	3	4	5	6	7	8	9	10	R1	R2
5.8.14													
Number of members	57560	13053	15139	17831	17206	26504	37480	38341	59736	55701	40570	56651	56651
Profile coverage (%)	100.0	86.2	82.6	74.5	73.2	57.2	61.9	48.4	46.3	49.3	50.9	99.5	99.5
Correct / Wrong (%)													
– Overall	84.8 / 9.5	73.5 / 11.2	74.3 / 11.6	65.4 / 20.4	66.2 / 18.6	68.2 / 18.8	72.3 / 19.0	70.3 / 18.9	73.3 / 19.7	74.1 / 19.4	72.7 / 18.1	86.1 / 8.1	86.1 / 8.1
– Required by Volatility	80.9 / 6.4	77.3 / 8.5	78.0 / 8.5	75.9 / 9.9	70.3 / 9.4	75.2 / 7.3	78.7 / 9.2	71.5 / 8.8	77.3 / 9.2	81.6 / 7.8	76.6 / 9.2	79.0 / 9.4	81.9 / 7.2
Katana (analyses)	43	**38**	**38**	**36**	**40**	**41**	**38**	**38**	**38**	**38**	**38**	**25**	**34**
Volatility (analyses)	**52**	15	15	10	9	10	10	9	10	10	10	15	15
5.4.70													
Number of members	55783	12534	15088	18593	18986	25118	34954	28084	33007	68099	73932	55912	55912
Profile coverage (%)	100.0	99.9	92.7	90.2	84.8	82.8	61.8	66.8	65.4	50.4	45.7	99.5	99.5
Correct / Wrong (%)													
– Overall	84.9 / 9.6	74.7 / 8.4	72.8 / 11.5	77.4 / 8.5	70.2 / 15.3	79.3 / 9.4	73.3 / 16.4	72.9 / 14.3	72.5 / 16.0	71.8 / 20.8	73.6 / 17.6	86.2 / 8.0	86.1 / 8.1
– Required by Volatility	79.5 / 7.5	75.3 / 8.9	76.0 / 8.2	76.7 / 7.5	77.4 / 6.2	72.9 / 7.1	76.4 / 7.9	78.3 / 7.7	73.3 / 5.5	81.5 / 7.5	70.7 / 7.9	79.7 / 9.1	80.4 / 7.7
Katana (analyses)	42	**41**	**41**	**42**	**43**	**44**	**41**	**41**	**42**	**33**	**42**	**25**	**36**
Volatility (analyses)	**52**	15	14	15	10	9	10	10	10	10	9	15	15
4.19.150													
Number of members	53305	11907	13662	15591	19105	21505	34953	19766	20933	38619	57069	64239	64239
Profile coverage (%)	100.0	99.9	96.2	85.4	85.7	76.6	65.8	83.0	81.9	57.4	53.7	100.0	100.0
Correct / Wrong (%)													
– Overall	84.4 / 9.1	76.3 / 8.3	66.9 / 18.3	76.9 / 8.3	70.4 / 16.0	77.4 / 11.1	73.5 / 16.8	70.0 / 17.1	71.3 / 15.6	70.6 / 19.3	74.5 / 18.0	86.5 / 7.0	86.5 / 7.0
– Required by Volatility	86.3 / 4.1	83.6 / 7.5	80.8 / 10.3	79.3 / 8.1	81.5 / 6.8	87.0 / 5.5	80.8 / 10.3	78.1 / 7.3	77.4 / 7.5	76.7 / 9.6	84.2 / 8.2	73.3 / 6.2	73.3 / 6.2
Katana (analyses)	35	**35**	**28**	**41**	**29**	**41**	**29**	**30**	**33**	**29**	**30**	**34**	**34**
Volatility (analyses)	**52**	15	10	15	10	15	10	13	10	9	10	15	15
4.14.200													
Number of members	50852	11604	14345	14053	17953	14116	30092	26208	44598	32069	71410	60379	60379
Profile coverage (%)	100.0	99.9	94.2	92.8	76.5	86.7	67.5	69.3	61.6	63.0	43.2	100.0	100.0
Correct / Wrong (%)													
– Overall	85.0 / 9.8	77.2 / 8.2	78.0 / 8.7	78.1 / 8.1	77.2 / 9.1	67.8 / 17.0	74.6 / 16.0	73.5 / 17.2	75.2 / 17.8	70.0 / 19.6	72.2 / 19.7	87.1 / 7.7	87.1 / 7.7
– Required by Volatility	84.2 / 7.9	80.4 / 8.8	80.4 / 8.8	81.8 / 8.1	81.8 / 5.4	72.9 / 11.8	78.9 / 10.6	80.3 / 10.6	80.9 / 13.2	72.3 / 12.8	77.0 / 12.5	77.6 / 9.9	77.6 / 9.9
Katana (analyses)	35	**37**	**37**	**43**	**38**	**32**	**30**	**23**	**28**	**20**	**24**	**34**	**34**
Volatility (analyses)	**52**	15	15	15	15	14	9	10	10	10	10	15	15
4.9.238													
Number of members	48029	11278	11715	15081	16326	19487	26984	26324	29047	41700	42290		
Profile coverage (%)	100.0	99.9	99.3	90.1	87.7	82.4	72.8	68.9	60.8	57.5	58.0		
Correct / Wrong (%)													
– Overall	85.4 / 9.8	76.6 / 9.4	76.1 / 9.8	77.8 / 9.9	72.3 / 16.4	72.9 / 16.3	72.9 / 17.8	71.6 / 18.4	73.3 / 17.2	73.9 / 18.9	76.4 / 15.1		
– Required by Volatility	87.6 / 5.2	83.2 / 6.0	83.9 / 5.4	83.2 / 6.0	78.5 / 10.7	82.0 / 8.6	81.2 / 9.4	80.5 / 10.7	82.8 / 9.0	79.7 / 12.6	81.9 / 9.4		
Katana (analyses)	37	**36**	**36**	**36**	**28**	**28**	**28**	**24**	**33**	**19**	**27**		
Volatility (analyses)	**52**	15	15	15	10	10	10	10	14	10	10		
4.4.238													
Number of members	45633	10842	12416	12708	12091	13093	21366	24967	46264	47216	30369		
Profile coverage (%)	100.0	99.9	99.3	91.9	94.9	92.1	80.9	65.1	46.8	53.7	60.9		
Correct / Wrong (%)													
– Overall	86.0 / 9.8	77.2 / 9.9	78.2 / 11.4	77.4 / 10.1	70.6 / 16.7	71.4 / 16.6	73.4 / 17.8	73.6 / 17.8	73.6 / 18.9	76.0 / 17.6	72.0 / 19.7		
– Required by Volatility	88.4 / 4.5	84.1 / 5.3	85.4 / 5.3	85.4 / 4.6	73.8 / 9.7	75.2 / 8.3	78.8 / 9.9	77.5 / 13.2	78.6 / 11.0	77.9 / 10.3	80.7 / 12.4		
Katana (analyses)	38	**37**	**36**	**38**	**26**	**30**	**26**	**23**	**26**	**27**	**25**		
Volatility (analyses)	**52**	15	15	15	10	10	10	10	9	10	10		
3.10.108													
Number of members	39518	8285	9323	10123	13084	13953	18697	22406	28845	18267	21546		
Profile coverage (%)	100.0	99.9	93.5	89.2	87.2	78.1	69.4	69.8	59.8	62.7	53.6		
Correct / Wrong (%)													
– Overall	85.7 / 9.7	75.5 / 10.8	76.1 / 10.4	76.1 / 11.1	76.0 / 13.4	73.2 / 16.6	71.9 / 18.2	74.0 / 18.7	74.6 / 17.7	72.3 / 16.9	73.5 / 16.3		
– Required by Volatility	91.4 / 4.9	86.1 / 5.1	86.1 / 5.1	86.7 / 5.1	88.6 / 4.4	80.9 / 9.2	82.3 / 8.9	84.6 / 9.9	87.7 / 8.0	86.1 / 8.9	80.3 / 9.2		
Katana (analyses)	34	**37**	**37**	**37**	**34**	**28**	**27**	**25**	**29**	**33**	**27**		
Volatility (analyses)	**52**	15	15	15	14	10	13	10	13	10	10		

Structure layout randomization not supported

Table 3: Performance of Volatility and Katana in the presence of kernel configuration variations