In the Land of MMUs: Multiarchitecture OS-Agnostic Virtual Memory Forensics

Andrea Oliveri, Davide Balzarotti

In the Land of MMUs: Multiarchitecture OS-Agnostic Virtual Memory Forensics

ANDREA OLIVERI, Eurecom, France
DAVIDE BALZAROTTI, Eurecom, France

The first step required to perform any analysis of a physical memory image is the reconstruction of the virtual address spaces, which allows translating virtual addresses to their corresponding physical offsets. However, this phase is often overlooked and the challenges related to it are rarely discussed in the literature. Practical tools solve the problem by using a set of custom heuristics tailored on a very small number of well-known operating systems running on few architectures.

In this paper, we look for the first time at all the different ways the virtual to physical translation can be operated in 10 different CPU architectures. In each case, we study the inviolable constraints imposed by the MMU that can be used to build signatures to recover the required data structures from memory without any knowledge about the running operating system. We build a proof-of-concept tool to experiment with the extraction of virtual address spaces showing the challenges of performing an OS-agnostic virtual to physical address translation in real-world scenarios. We conduct experiments on a large set of 26 different OSs and a use case on a real hardware device. Finally, we show a possible usage of our technique to retrieve information about user space processes running on an unknown OS without any knowledge of its internals.

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

Additional Key Words and Phrases: memory forensics, OS-agnostic forensics, virtual memory, MMU

ACM Reference Format:
Andrea Oliveri and Davide Balzarotti. 2022. In the Land of MMUs: Multiarchitecture OS-Agnostic Virtual Memory Forensics. 1, 1 (April 2022), 33 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

The problem of recovering semantic information from low-level data is common to many areas of computer security. In particular, this is the main obstacle when performing a physical memory analysis—a task that is key for both memory forensics and virtual machine introspection. The problem, often called the *semantic gap*, captures the challenge of "interpreting low level bits and bytes into a high level semantic state of an in-guest operating system" [35]. However, at a closer look, the semantic gap can be further divided into two different aspects: the reconstruction of the virtual address spaces (which deal with translating pointers expressed as virtual addresses to their physical position in the memory) and the recovery and identification of key operating system (OS) kernel data structures (e.g. those related to running processes, memory management, and kernel modules).

In practice, most tools and existing techniques neglect the first aspect. For instance, Libvmi, a popular virtual machine introspection library, explicitly mentions that the virtual-to-physical translation is only available on live VMs as it would otherwise require OS-specific heuristics [67]. Similarly, Fu and Lin [35] acknowledge that to analyze the content of the memory, the first step requires "to perform the MMU level virtual to physical (V2P) address translation" but the authors again avoid the problem by inspecting the registers of a live target. Other papers focusing on narrowing the semantic gap, such as Virtuoso [27], relied instead on pre-existing frameworks (e.g., Volatility[64] and Rekall[23]) for the V2P translation. These memory forensics frameworks, as well as products developed by BlackBag[2] and Volexity[63], rely,

Authors' addresses: Andrea Oliveri, Eurecom, Sophia-Antipolis, France, andrea.oliveri@eurecom.fr; Davide Balzarotti, Eurecom, Sophia-Antipolis, France, davide.balzarotti@eurecom.fr.

2022. Manuscript submitted to ACM

in turn, on a set of manually curated and very specific OS and architecture heuristics to locate kernel data structures that are used to recover the mapping between the virtual and physical address spaces. However, these heuristics are based on a deep knowledge of the internals of the OS under analysis, thus precluding their generalization to different operating systems or different CPU architectures. In other words, every memory analysis study to date has "avoided" the virtual memory translation step either by focusing only on live systems or by considering only a small subset of OSs (in practice, Linux, Windows and OSX) running only on the x86 architecture and, partially, on ARM (in particular Android and recent Apple M1 devices). However, as recently shown by Cozzi et al. [25], IoT malware authors started to target also architectures less common in desktop environments, such as PowerPC and MIPS. This is troublesome from both an academic and a practical aspect. In the academic community the virtual-to-physical address translation, which is a fundamental step of the semantic gap reconstruction, is considered a solved problem. However, except for a few OSs running on x86 and ARM, this is far from being true. At the same time, from a practical point of view, the lack of cross-architecture OS-independent analysis techniques to perform V2P translation poses a serious challenge for the future of memory analysis. In fact, the rapid increase in IoT devices and cloud-hosted VMs translates into a more variegated number of architectures and operating systems.

1.1 Contribution

In this work, for the first time, we systematically study how to bridge the semantic gap in virtual-to-physical translations in 10 different CPU architectures in an OS-agnostic way. We show how the traditional page tables, which most researchers are familiar with, is only one of many possible ways to perform the V2P translation and how the memory management unit (MMU) affect the reconstruction of V2P mappings. In each case, we study the inviolable constraints imposed by the MMU and use them to build signatures to recover the required data structures from memory without *any* knowledge about the running OS. We also introduce a static code analysis step to recover the state of the MMU registers configured by the OS at boot time. We implement these techniques in a tool that, in contrast to the existing forensics tools available, uses only parameters derived from the CPU ISA to recover V2P mappings. We test our tool against memory dumps collected from 26 different operating systems and a physical device, proving the usefulness and accuracy of the described techniques in the reconstruction of the V2P mapping in real-world scenarios. Finally, we show a possible practical usage of our technique to recover and analyze the user space processes running on an unknown device. This permits an analyst to start a forensics analysis on the system without any knowledge of the internals of the OS.

2 VIRTUAL MEMORY: BASIC CONCEPTS

The term *Virtual Memory* refers to an abstraction of the memory resources available on a given machine. In virtual memory mode, each program performs memory accesses using an isolated and private address space called virtual address space (VAS). The combined work of the MMU and the OS permits to translate virtual addresses of a particular process into physical addresses that specify where the data is physically located in memory.

The virtual memory abstraction provides numerous benefits, allowing programs to be written as if they had complete and unrestricted access to the entire physical memory, regardless of its usage and the presence of other programs. This permits to run multiple processes at the same time without worrying about the actual physical memory configuration, which can change across different types of machines. Virtual memory provides also isolation and protection between processes by preventing malicious, unauthorized, or poorly implemented programs to access the memory of other running programs. Furthermore, it allows a program to allocate memory beyond the limits of the physical address space (PAS) by mapping part of the VAS of a process onto secondary storage.

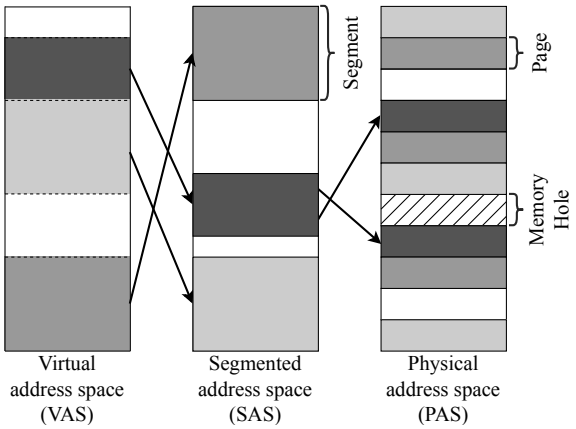


Fig. 1. Virtual, segmented and physical address spaces. Memory holes are physical memory regions that are not usable as storage memory regions (e.g MMIO, device reserved regions, ROMs, not assigned regions etc.)

The virtual memory abstraction is implemented in different architectures by relying on two different techniques: *segmentation* and *paging*, as illustrated in Figure 1. Segmentation, the oldest of the two methods, was originally developed to organize and protect running programs. Its goal is to divide the VAS of a process into one or more logical units, called *segments*, with different sizes and access permissions, by mapping them into another address space called segmented address space. In general, segments follow the internal structure of the program that they represent: the memory of a process can be divided, for example, into two different segments containing, respectively, the code and the data. Segmentation, however, does not permit optimal use of the available memory: if we map segments of various programs directly on the physical memory, the PAS starts to fragment and at some point it is impossible to allocate new chunks of sufficient size to accommodate new segments.

To solve this problem we first need to introduce the concept of *page*, which is a contiguous block of memory of fixed (and typically small) size. Then we divide the segmented address space (SAS) into a set of pages (called virtual pages), and we define a way to map them to pages in the physical address space (called physical pages) as shown in Figure 1. This technique, called paging, drastically reduces the fragmentation of the physical memory and contributes to maximizing the use of the available resources. In the context of our study, it is important to understand that both segmentation and paging require some in-memory data structures, or dedicated CPU registers, that need to be configured by the OS, and used by the MMU.

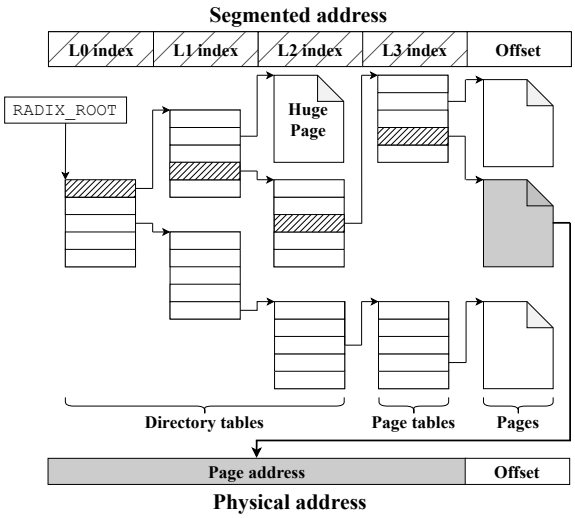


Fig. 2. Radix tree address resolution process.

2.1 MMU

The MMU is the hardware device that converts the virtual addresses used by the processor to physical addresses. To accomplish this task, the MMU needs to be configured by using special system registers, while in-memory structures that maintain the virtual-to-physical mapping have to be defined and continuously updated by the operating system. When the MMU fails to resolve a requested virtual address, it raises an interrupt to signal the OS to update the in-memory related structures.

It is important to note that the MMU demands strict conformity of the shape and topology of the in-memory structure to the ISA and MMU configuration requirements. Otherwise, it raises an exception and aborts the address translation.

The translation process can involve up to two separate translations, both accomplished by the MMU: segmentation, which converts virtual to segmented addresses, and paging, which converts segmented addresses to physical ones. Some architectures use either one or the other, while others use both.

In general, when the system boots, the MMU is virtually disabled and all the virtual addresses are identically transformed to physical ones. This allows the OS to start in a simplified memory environment and gives it time to properly configure and enable the MMU before spawning other processes.

Since address translation is a performance bottleneck, the latest translated addresses are cached in a few but low-latency hardware structures called Translation Lookaside Buffers (TLBs). Before starting a translation, the MMU checks if TLBs contain an already resolved virtual address and, if so, it returns directly the corresponding physical addresses.

Our analysis of the most common CPU architectures shows that two main approaches are used for paging: radix trees, and inverted page tables.

2.2 Radix Tree

Radix trees maintain a hierarchical representation of the SAS. Each tree is composed (see Figure 2) by N-1 levels of *directory tables*, each containing entries that either point to tables of the lower level or to a physical memory page (*huge pages*), whose size depends on the level itself. The final level is composed of *page tables* that point only to same-size physical memory pages. The tree root is the physical address of the directory table of Level 0 and it identifies uniquely the SAS and, consequently, the process to which it is assigned.[1] This address is stored in a special system register (here generically called `RADIX_ROOT`) by the operating system and it is used by the MMU to locate the radix tree when it starts a new translation. The translation performed by the MMU starts from the root table pointed by the address contained in `RADIX_ROOT`: the MMU then divides the segmented address into two parts: a prefix and a page offset. The prefix part is divided into a series of N chunks that represent the hierarchical sequence of indexes to be used to locate the entry inside a table of the corresponding level. This process ends when an entry points to a physical page. At this point, the MMU returns the concatenation of the page offset extracted by the segmented address to the physical page address found in the last page table entry.

2.3 Inverted page tables

Inverted page tables maintain one entry for every physical page in the system along with the associated virtual page address in its segmented space. As such, instead of having one radix tree per segmented space, a single inverted page table is maintained for all the processes of the system. Inverted page tables are often implemented by using a *hash table*, whose address is stored in a special system register. The OS then needs to use the same hash function used by the MMU, as defined in the processor ISA, to index an entry in the hash table when allocating memory for a process. When the system needs to resolve an address, (see the right side of Figure 4) the segmented address is split into the page number and a page offset. Starting from the page number and a unique segment identifier (VSID), the MMU generates a hash value which is retrieved from the hash table and, in the case of a positive match, the physical address is formed by concatenating the physical page address, indexed by the hash, with the page offset extracted from the segmented address.

3 APPROACH

The goal of our work is the automatically recover the virtual address space (VAS) of the kernel and all user space processes contained in a memory dump, independently from the OS and the applications that are running on the machine. The only input to our analysis is the CPU architecture and a copy of the raw memory. To reconstruct the set of VASs we need to extract the configuration of the MMU and also locate and interpret the data structures that are used by the MMU to translate virtual to physical addresses. While the actual techniques we employ are architecture-specific, and therefore we will present them in detail in the following sections, we can summarize them in three broad categories.

3.1 Structural Signatures

The use of structural signatures computed over the values of individual fields (sometimes called *invariants*) has been used in several studies to locate particular data structures in memory dumps or binary blobs. This technique has been successfully applied to retrieve OS kernel structures [28, 47], application data [46], user space stack layouts [1], internal representations of malicious processes [26], and hypervisors/VMs configurations [36]. We follow a similar approach by

[1] In certain architectures, the kernel does not have its own radix tree but is instead mapped inside every process segmented space at a fixed continuous block of addresses.

manually studying the different MMU data structures and compiling a list of structural constraints that can be used to build pattern-matching signatures.

For instance, a set of constraints to match Intel x86_64 ISA directory tables require each entry pointing to an inferior table to have the `P` bit set (`entry[P]==1`), some other bits unset (`entry[MAXPHYADDR:51]==0` $\wedge$ `entry[7]==0`) and the pointer containing a paging table to be in the PAS range (`entry[addr]` $\ll$ `12` $\in$ `RAM`). These constraints are then translated to patterns that can be used to verify whether the bits of a given physical page satisfy the requirements. The complete list of constraints we use in this paper is reported in Appendix B. It is important to stress that these signatures are derived solely by analyzing the inviolable constraints imposed by the MMU and they are completely independent of the OS.

3.2 Validation Rules

Structural signatures are used to match all the possible regions of memory that *could* contain a given data structure. However, the patterns are not very unique and therefore result in a large number of false positives. Therefore, the second type of technique we use is a set of rules to filter out the noise and reduce the number of candidates. However, since we cannot make assumptions on the underlying OS and we already use all the information from the MMU to distil the signatures in the first place, possible validation routines are scarce and difficult to construct.

The problem is further complicated by the great variability of OS behaviours in managing the system resources. For instance, one might think to discard tables that contain pointers to physical pages outside the size of the physical memory itself. However, even this simple rule fails in practice as some OSs create a complete radix tree that can address all the possible physical memory pages of the RAM, whether such memory is installed or not on the machine. Furthermore, all OSs maintain a mapping to MMIO ranges that are often allocated far outside the PAS. Another simple strategy would be to set a minimum threshold on the number of pages (but this fails as the kernel can map the process VAS using a single physical page) or to look at how the physical pages of a process are distributed in the physical memory and discard the structure if not even two pages are consecutive. However, even this simple idea did not work as we encountered operating systems that spread physical pages all over the entire PAS and, in general, is false for all the OSs in presence of high memory pressure.

To avoid these problems, our strategy to create validation rules is based on the use of other inviolable constraints imposed by other CPU subsystems. In particular, we aim at identifying pages that *must* always be mapped in the VAS to have a functional system (for instance, the ones containing the Interrupt Address Table) for certain architectures and use them to discard those VASs that do not map their physical addresses.

3.3 Binary Code Analysis

Some architectures use MMU-related CPU registers that contain values that do not refer to any in-memory data structure. Thus, to recover these values, we cannot rely on signatures. However, we can still take advantage of how the boot process works: when a machine boots up the bootloader loads the kernel code into physical memory and runs it. In this early boot stage, the MMU is disabled and all the virtual addresses are mapped in a fixed way to the physical ones. At this point, the kernel sets up the MMU-related registers based on the physical configuration of the machine (e.g. the CPU model or the size of the RAM). When we dump the physical memory of the system, this contains all the allocated kernel code[2] and this allows us to try to recover the content of the MMU registers by analyzing the assembly

[2]It is possible that the MMU setup is done by the bootloader and the kernel can delete or rewrite the physical memory pages containing it. However, all the OS we encountered in our study leave the task of the configuration of the MMU to the kernel itself.

instructions used by the kernel code to load their values. In particular, for architectures that have aligned and fixed-size opcodes, we can search for the byte patterns corresponding to the instructions that load values inside the MMU-related registers. Then we can identify the functions containing these opcodes and finally, we can use data-flow analysis to derive which values are loaded inside them. To accomplish this task, default values of some CPU registers at the system power-on are required because the kernel could use them to decide how to configure the MMU.

In the next three sections, we discuss ten of the most popular CPU architectures, grouped into three separate groups based on the different MMU modes available on them. The first group includes ARM, RISC-V and x86 in their 32 and 64-bit implementations. These architectures use exclusively radix tree-based MMU modes. The second group contains PowerPC (32-bit), which uses the inverted page table as main MMU mode but, at the same time, also has another peculiar mode that distinguishes it from other architectures. Finally, the last group contains MIPS in its 32 and 64-bit flavours. These architectures show unique and very flexible management of the virtual memory, which requires a separate category. The last architecture, Power, is treated instead in Appendix A.

4 GROUP I: RADIX TREES

This group includes three architectures that use a radix tree to translate virtual to physical addresses. We present them here ordered from the easiest to the hardest to analyze.

4.1 RISC-V (32 and 64-bit)

4.1.1 MMU Internals. RISC-V is an open-source ISA first published by UC Berkeley in 2010. It has become increasingly popular as several large companies[21][66][44] have announced its use in their future products, or they are implementing their custom versions. The last ratified ISA [65] features both 32-bit and 64-bit little-endian CPUs that support three different MMU modes: `SV32`, `SV39` and `SV48` (the last two available only for 64-bit CPUs). As there is no segmentation unit in RISC-V, virtual addresses are identically mapped to segmented ones. The MMU uses the `SATP` register as `RADIX_ROOT` register, which also permits to select the MMU mode. The radix tree is composed of two, three, or four table levels (respectively for `SV32`, `SV39` and `SV48`) with a predefined size and fixed layout of the entries specified by the ISA. Every mode supports both 4KiB pages and huge pages of different sizes.

4.1.2 Analysis. In RISC-V the shape of the entries permits to distinguish among tables of different levels, thus allowing the reconstruction of a radix tree in a very simple way. The algorithm starts by parsing the memory dump and identifying directory tables of all levels, page tables, and data pages by using the set of constraint signatures presented in Appendix B. Starting from data pages and empty tables we then look for all tables which point to them and recursively list all tables of level N-1 pointing to the ones of level N in a forward search. By using this technique we can completely reconstruct the radix tree, find the top-level table (and its address contained in the `SATP` register), and derive the VAS associated with it. Furthermore, the privilege separation and access permission bits, present in the table entries, permit to distinguish user pages from kernel ones, and pages containing data from those containing executable code.

4.2 Intel x86 (32 and 64-bit)

4.2.1 MMU Internals. First introduced in 1985 by Intel, the x86 ISA has been extended over the years by adding new functionalities and new instructions, maintaining however full backward compatibility. This fact has resulted in the existence of a segmentation unit that must be enabled to use the paging one[42]. Before the existence of the paging unit, OSs running on x86 ISA used segmentation as a form of isolation between the processes running on the system.

However, after the introduction of the paging unit, segmentation was quickly replaced by paging by all OSs. As we could not find any case in which the two were used in combination, for the 32-bit version of the architecture we assume that the operating system defines segments that identically map virtual addresses to segmented ones.

When AMD introduced the x86_64 architecture extension (also known as AMD64), aware of the fact that the segmentation unit was not used by any OSs, it virtually disabled it: when the MMU is configured for using the 64-bit specific paging mode it continues to operate with the segmentation unit enabled but it uses an automatically defined set of segments, which identically map all the virtual addresses to segmented ones.

The current implementation of the x86 architecture supports three major MMU modes with paging enabled: 32-bit mode, PAE mode, and 4-level mode, the last one available only in 64-bit. The physical address of the root of the radix tree is stored in the `CR3` register and the radix tree is composed, respectively, by two, three, or four table levels with a predefined size and fixed layout of the entries. Every mode supports 4KiB pages, as well as huge pages of different sizes.

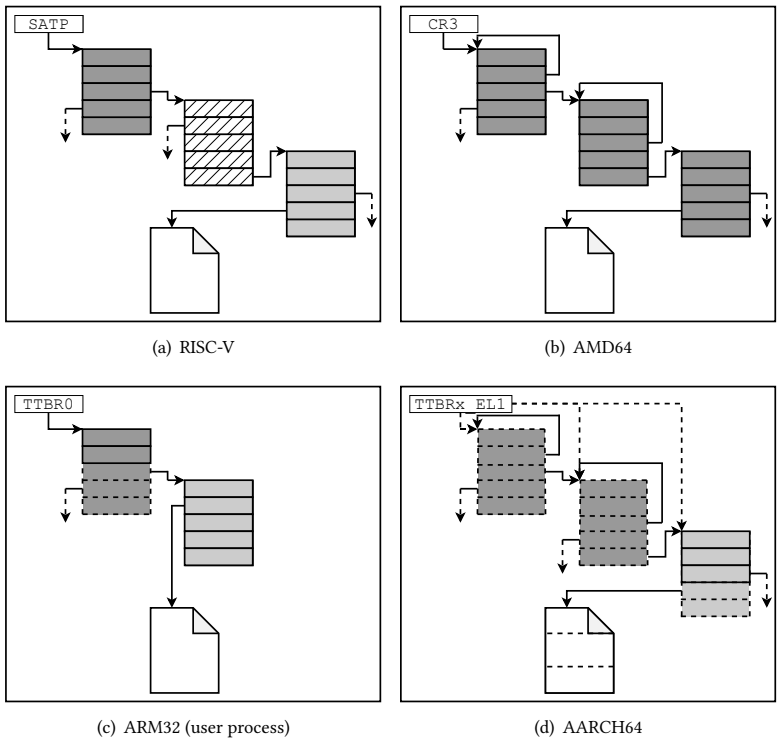
4.2.2 Analysis. In contrast to the RISC-V ISA, the x86 directory table and page tables are, in general, not distinguishable. This peculiarity is exploited by some OSs (for example the Windows family) to allocate a reduced number of tables by inserting self-references entries in them (see Figure 3). This poses a problem for our analysis as it is difficult to tell whether a parsed table is the root of a radix tree, an intermediate level, or a leaf one.

To solve this problem we use another structure that must be initialized by any operating system and that has a predefined format: the interrupt table. In the x86 architecture, the interrupt descriptor table (IDT) has a predefined length and is composed of entries of fixed format. Each entry contains the virtual address of an interrupt handler to which the CPU jumps when it receives an interrupt. Every process must have mapped the IDT in its PAS because an interrupt can occur at every moment also when the system is not running kernel code. This allows us to discriminate between root directory tables and lower-level tables: true top directory tables are starting points of radix trees that are able to resolve all the virtual interrupt handler addresses while trees with false-positive top directory tables, in general, are not able to do so. As for the RISC-V architecture, it is possible to infer which ranges of the VAS of a process contain code or data and its privilege level thanks to the flags available in the tables entries.

4.3 ARM (32-bit)

4.3.1 MMU Internals. ARM is a RISC (Reduced Instruction Set Computer) architecture developed by ARM Holdings and widely adopted in mobile and IoT devices. ARM-based processors are produced in different implementations and versions by many companies. In our study, we consider only the Application Profile ARM processors (ARM-A) which is the only one implementing an MMU. ARM32 CPUs are bi-endian and support two MMU modes[39]: short and long descriptor modes for which the radix tree have, respectively, two and three levels and map 4KiB pages as well as huge pages of different sizes. Each VAS is divided into two parts. The high-end one is resolved by using a radix tree pointed by the `TTBR1` register, contains the kernel code/data, and is accessible only in that mode. The lower end is instead resolved by using the `TTBR0` register, which points to a different radix tree, containing the data structures of the running process.[3] At every change of the execution context, the scheduler loads the value of `TTBR0` register accordingly to the scheduled process while the value of `TTBR1` is left unchanged. The `TTBCR` register controls which MMU mode is used and the size of the kernel and user VAS, from which it derives the size of the user space top-level tables in short descriptor mode.

[3]It is also possible for the OS to use only `TTBR0` as `RADIX_ROOT` register for both the kernel and the user parts of the virtual address space.



(a) RISC-V (b) AMD64

(c) ARM32 (user process) (d) AARCH64

Fig. 3. Radix trees used by RISC-V, AMD64, ARM32, and AArch64. Tables with different layouts are represented with different colours. Tables and pages with partially dashed borders can have different sizes. Tables with full dashed borders shall not exist in the tree. Note the possibility for x86 and AArch64 tables to have self-referencing entries.

4.3.2 Analysis. Because of the peculiarities of the ARM architecture, an approach based solely on signatures (like the ones used for RISC-V and Intel) is not sufficient. In fact, we first need to recover the value of the `TTBCR` register to identify the virtual address space layout. For this reason, ARM32 requires to first perform our code analysis phase described in Section 3. If the procedure succeeds, we then continue our analysis based on the retrieved information.

As it was for RISC-V, also in ARM short descriptor mode the tables can be easily classified as page or directory tables. We can then parse the memory looking for kernel directory tables and page tables, which have both a fixed size, and user process directory tables of sizes compatible with the recovered value of `TTBCR`.

If the MMU is configured in long descriptor mode, things get more complicated. In this mode, it is not possible to distinguish between directory tables of different levels but only between directory and page tables. Moreover, in ARM we cannot rely on the IDT to filter out false radix trees because its entries do not have a fixed format, which makes them difficult to locate in memory. To solve the problem, we look for pages containing code that must always be present to have a fully functional paginated system and which must be reachable in kernel mode. Our technique is based on the fact that in ARM32 CPUs when the MMU is unable to resolve an address it raises an interrupt and it inserts information about the cause of the faulty translation inside the `IFSR` / `DFSR` registers. These are privileged registers accessible only when the CPU is in kernel mode and must be read by the MMU related interrupt handlers to correctly manage the fault. Since this code snippet needs to always be accessible in memory, we can look for opcodes that read the content of

IFSR / DFSR registers to identify physical pages which must be indexed from a valid kernel radix tree and filter out false positive ones.

4.4 ARM (64-bit)

4.4.1 MMU Internals. Introduced in 2011, ARM 64-bit (also known as AArch64) is a new instruction set that shows retro compatibility with ARM32 and, at the same time, drastically changes the internal organization of the CPU by introducing new operating modes, new MMU modes, and by defining different sets of system registers for different CPU modes. The ARM32 TTBR0, TTBR1 and TTBCR registers are replaced by TTBR0_EL1, TTBR1_EL1 and TCR_EL1: the EL1 suffix indicates that they are writable when the CPU is in kernel mode and are used in kernel and less privilege modes as user mode (suffixes EL2 and EL3 are used to indicate registers that are part of the register sets of the higher privileged hypervisor and monitor modes). Note that also the MMU configuration register (TCR) is duplicated between CPU modes: the MMU can be configured to have different behaviours when the CPU runs user and kernel code, hypervisor, or monitor one. AArch64 introduces two new MMU modes: the long mode and LPA long mode, the latter using a slightly modified version of table entries derived from the former and supporting a PAS up to 52 bits. These two modes are based on the long mode of ARM32 and they inherit the impossibility to distinguish directory tables of different levels, thus permitting the existence of loops in the radix tree. Important changes are introduced by AArch64 on the supported physical page sizes and the number of levels of the radix trees. AArch64 support three physical page size (4KiB, 16KiB, and 64KiB, plus huge pages) which can be configured independently for the kernel and the user processes. Furthermore, the sizes of the kernel and user process parts of the VAS and the sizes chosen for the physical pages determine the size of the tables, their alignment in memory, and the number of levels in the radix trees.

4.4.2 Analysis. To deal with the AArch64 MMU specificity, we adapted the parsing technique used for ARM32, but replacing the IFSR / DFSR registers with their corresponding ESR_EL1, FAR_EL1 and ELR_EL1. We also modified the algorithm to reconstruct radix trees compatible with the new constraints based on the shape imposed by TCR_EL1. The impossibility, also for ARM64, to recover the IDT does not permit to resolve the ambiguity due to the presence of loops in trees, supposedly leaving more false positives compared to ARM32 short descriptor mode. This affects in particular user processes that do not contain any specific instructions or code snippets we can use to filter out false positives.

5 GROUP II: INVERTED PAGE TABLES

5.1 PowerPC (32-bits)

5.1.1 MMU Internals. PowerPC is a 32-bit big-endian architecture used mainly in desktop and medium-class servers. It supports only one MMU operating mode based on Block Address Translation + Inverted Hash Table (BAT+SDR1 mode). In BAT+SDR1 mode the MMU uses two concurrent resolution processes [34], one based on block translation and one on the inverted page table: if the block one succeeds the inverted page resolution is interrupted.

The block translation mode relies on a set of CPU registers (the BAT set) to define fixed mappings between contiguous blocks of VAS to same-size blocks in the PAS. If the virtual address to be translated belongs to one of the ranges defined by the BAT registers the MMU returns the associated physical address. Each BAT register describes the characteristics of one of the contiguous blocks: its physical start address, its length, the start address of the associated VAS, the permissions flags and CPU modes that are allowed to access the memory.

The inverted page table resolution walk starts, instead, by selecting one of the 16 segment registers (SR, in Figure 4) according to the upper four bits of the virtual address. Segment registers contain access permission bits in addition to a

virtual segment identifier (VSID), which is combined with the remaining part of the virtual address to form the Virtual Page Number (VPN). The VPN is then passed as input to the hash function to get the key for the hash table (whose starting address and size are stored in `SDR1` register). The extracted value, the real page number, is the physical address of the page associated with the initial virtual address.

5.1.2 Analysis. As in the case of ARM and AArch64, a combination of parsing and data-flow analysis is required to recover the information stored in the CPU registers. Our analysis starts by looking for the hash tables of the minimum size allowed by the ISA and then we progressively aggregate them to form bigger ones. We also perform a code analysis phase to try to recover the `SDR1` value which indicates, without ambiguity, the physical address and the size of the table. The same code analysis also tries to recover the content of the `SR` and `BAT` registers. However, some of the `BAT` registers could be used on a system-wide base, for example by defining fixed blocks of physical memory dedicated to the kernel data and code, or on a per-process base, thus changing at every context switch. In the second case, the code analysis phase cannot succeed because of the custom storage format used by the kernel to save the registers in its private context switch data structures. In the same way, the `SR` register set is used to define segments inside the SAS and, in general, the majority of them change during a context switch. This fact does not permit to recover the segment definitions and cannot permit to assign them, unambiguously, to each different process.

This complexity in the use of the `BAT` and `SR` registers affects the ability to recover the VAS of the kernel/user processes in two different ways. First, even when we can recover the content of `BAT` registers, they do not contain any information about which process they refer to. Furthermore, the MMU uses the upper four bits of a virtual address resolved through the inverted page table to choose the segment to use during the translation. Therefore, if we are not able to extract and associate the `SR` registers to a process, we can only partially reconstruct the V2P mapping. In Section 8.4 we will see the consequences of these on the results.

6 GROUP III: SOFTWARE-DEFINED ADDRESS TRANSLATION

6.1 MIPS (32 and 64-bit)

6.1.1 MMU Internals. MIPS is a modular RISC architecture developed by MIPS Technology and largely used in embedded devices. The current ISA release (revision 6) features both 32-bit and 64-bit bi-endian CPU flavours and it supports a range of both mandatory and optional MMU modes. The segmentation unit is always active, also during boot, and the VAS is partitioned into multiple segments with different access permissions and translation policies. Segments can be mainly classified in *unmapped* and *mapped*: virtual addresses which belong to unmapped segments are translated directly into physical addresses. On the contrary, mapped segments are paginated and the virtual addresses which belong to them are translated by using the Translation Lookaside Buffers.

The MIPS ISA defines a default set of segments which contains at least one unmapped segment contain the kernel code (`kseg0` and `kseg1` in 32-bit and `xkphys` in 64-bit implementations). The ISA also defines a default mapped segment which contains virtual addresses associated with code and data of user processes (`useg` / `xuseg`), and at least one mapped segment which contains data structures of the kernel (`kseg2` and `kseg3` in 32-bit and `xkseg` in 64-bit implementations).

At boot, the segments use the ISA predefined layout (shown in Figure 5) but some MIPS CPU models permit, by modifying the content of the MMU `SegCtl` registers, to redefine segments by changing their starting virtual address, size, access permissions and translation mode. They also allow to completely disable the unmapped kernel segment (EVA mode), thus using paging also for the addresses of the kernel code.

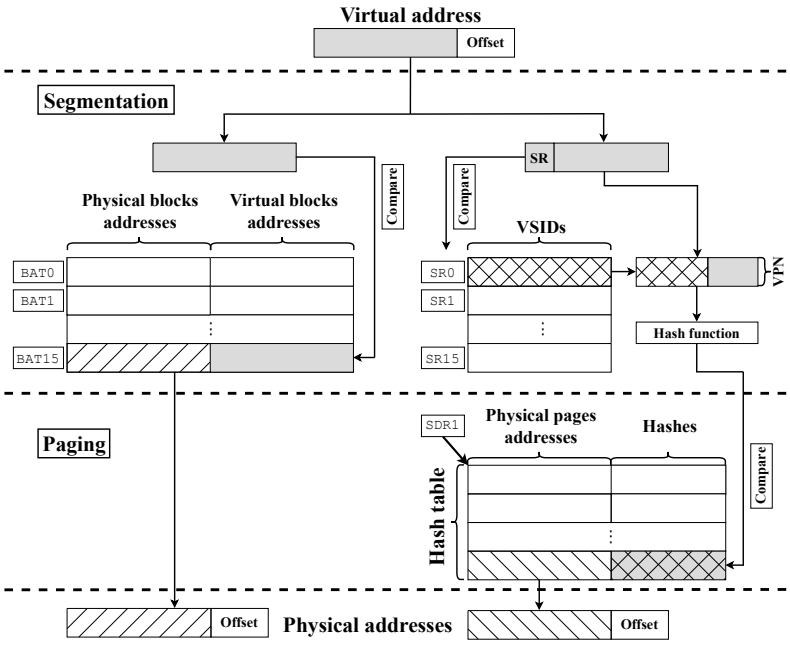


Fig. 4. Address resolution in PowerPC.

All the architectures we discussed so far automatically manage the TLBs content by using the MMU hardware TLB refiller which, thanks to the in-memory structures defined by the OS, can resolve segmented addresses never solved before and refill the TLBs. However, MIPS CPUs by default do not have a hardware TLB refiller and all the complexity of the paging and TLB refilling is therefore delegated to the operating system. This approach offers great flexibility as the OS can choose which data structure to use to manage the memory allocation and segmented addresses translations, e.g. by using a complete software implementation of radix trees or huge arrays of page table entries [4]. When the MMU does not find a valid TLB entry able to translate a segmented address, it raises an exception. In response, the OS needs to look into its software paging data structures and refills the TLB with the missing physical page.

Finally, the MIPS ISA defines also an optional MMU mode based on a hardware radix tree TLB refiller, which is much more flexible than the equivalent modes provided by other architectures. In fact, the number of the radix tree levels, the size of huge pages, and the bits of a segmented address that need to be translated are all completely configurable via MMU registers. Furthermore, the format of the page table entries is not completely defined by the ISA, which only lists the essential fields and leave their absolute positions inside the page table entries to be configured via MMU registers.

6.1.2 Analysis. The lack of in-memory translation structures (in the case of default MIPS MMU mode), and the great complexity and variability of the page tables (in the case of MMU supporting TLBs hardware refiller), makes it very difficult to implement a fully automated recovery technique. However, we can still use our code analysis phase to recover the configuration of the segments (`SegCtl0, SegCtl1, SegCtl2` registers) and, if the CPU supports it, also

[4]The MIPS architecture uses a dedicated `Context` register to support this last technique as illustrated in [50]. This register is used by the OS to maintain the address of an array of physical pages allocated for the current process, changing its value at each context switch. However, in contrast to other architectures, the resolution of segmented addresses and the refill of the TLBs continue to remain a software task.

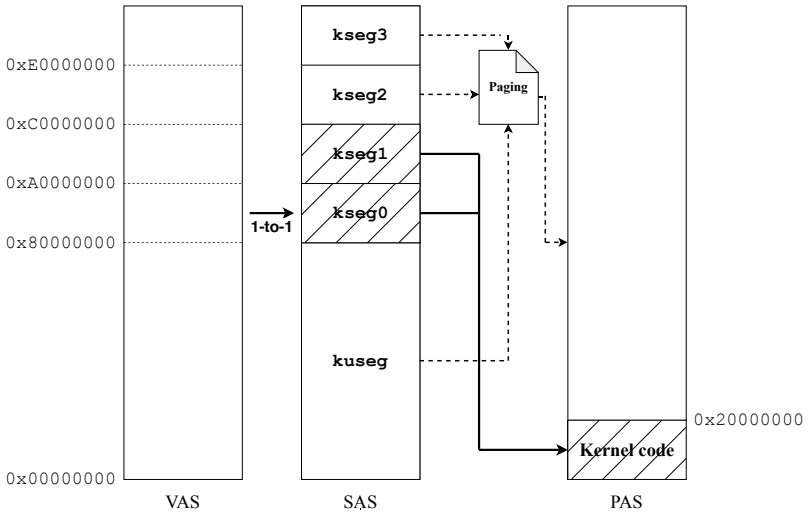


Fig. 5. Segmentation and paging in MIPS32 using default segments layout.

the configuration values of the TLBs hardware refiller (`Config, Config5, ContextConfig, PageGrain, PageMask, PWBase, PWCtl, PWField, PWSize, Wired` registers). Unfortunately, even when all register values can be recovered, they can only be used to reconstruct the segment configuration and the code part of the VAS of the kernel, which is loaded inside unmapped segments. In EVA mode and for user process VASs in systems without a hardware TLBs refiller, a manual analysis of the TLB interrupt handler (that can be easily located in a memory dump by looking for opcodes that load the TLBs) could allow an analyst to reverse engineer the custom algorithm used by the kernel to map virtual to physical pages. Instead, in presence of a hardware TLBs refiller, by combining the MMU configuration retrieved by the code analysis phase with a custom parser for page tables, it could be possible to manually recover the radix trees used to resolve the VASs of kernel data and user processes.

7 IMPLEMENTATION

In the previous sections, we presented the internal details of how the MMU translates virtual to physical addresses on different architectures. We also discussed how the different structures can be identified and reconstructed from a memory image, and which class of techniques is required to retrieve the different pieces of information. Table 1 presents a summary of the various architecture and MMU modes and shows what could be recovered when operating *in a perfect scenario*. In other words, this ignores possible false positives and assumes a perfect code analysis phase that always succeeds in reconstructing the value of the MMU registers. Thus, the results presented in the table are a useful upper bound of what could be achieved in theory. In reality, results can be much worse.

Therefore, to test and validate the techniques discussed in Section 3 in real-world scenarios, we implemented them in a proof-of-concept tool written in Python: MMUShell. MMUShell takes as input a memory dump and a YAML file that describes the hardware machine on which the image was collected. This includes information about the CPU architecture, the MMU mode (if known), and the default hardware values of the machine at the reset (e.g. the layout of

Table 1. Data structure/register recovery for all the studied architectures *in a perfect scenario*.

Architecture	MMU mode	Mode structure						Recoverability				
		Segmented	Paginated	Paging Mode	In-memory structs	In-register structs	Conf. registers	Kernel MMU structs	User MMU structs	Kernel VA space	User VA space	Access perms
AArch64	Long	○	●	◇	●	○	●	●	●	●	●	◐[1]
	Long LPA	○	●	◇	●	○	●	●	●	●	●	◐[1]
AMD64	4-level	◐[2]	●	◇	●	○	○	●	●	●	●	●
ARM32	Short	○	●	◇	●	○	●	●	●	●	●	◐[1]
	Long	○	●	◇	●	○	●	●	●	●	●	◐[1]
MIPS32	TLB	●	◐[3]	-[4]	◐[4]	○	●	◐[4]	◐[4]	◐[5]	◐[5]	●
	Radix	●	◐[3]	◇	●	○	●	●[6]	●[6]	●[6]	●[6]	●
MIPS64	TLB	●	◐[3]	-[4]	◐[4]	○	●	◐[4]	◐[4]	◐[5]	◐[5]	●
	Radix	●	◐[3]	◇	●	○	●	●[6]	●[6]	●[6]	●[6]	●
PowerPC	BAT+SDR1	●	●	□	●	●	○	◐[7]	◐[7]	◐[8]	◐[8]	◐[9]
RISC-V32	SV32	○	●	◇	●	○	○	●	●	●	●	●
RISC-V64	SV39	○	●	◇	●	○	○	●	●	●	●	●
	SV48	○	●	◇	●	○	○	●	●	●	●	●
x86	IA32	●	●	◇	●	○	○	●	●	●	●	●
	PAE	●	●	◇	●	○	○	●	●	●	●	●

Paging modes: ◇ Radix tree, □ Inverted page table.

[1] Some permission bits depend on SCTLR register value.

[2] Segmentation unit is active but virtually disabled.

[3] Paging is used for kernel data and user address space.

[4] Software dependent.

[5] ISA defines specific address ranges for kernel and user address spaces, internal allocations are software dependent.

[6] Not in an automatic way.

[7] SR and BAT registers can be associated with the running process making them not recoverable using data-flow analysis.

[8] The VAS mapping is only partially recoverable.

[9] Permissions for paginated SAS are stored in SR registers, which in general are not recoverable.

the PAS and the initial content of some CPU registers). All these are public information, that can be extracted from the hardware specifications, and are completely independent of the operating system that is running on the device.

MMUShell implements all parsing rules listed in Appendix and uses them to create patterns that can locate candidate MMU data structures. Whenever the system needs to retrieve a value contained in a MMU register, MMUShell identifies all the locations of instructions that operate on the corresponding register and retrieves all previous instructions until it finds an unconditional jump, a long jump, an instruction signalling a function epilogue (e.g RET instruction for AArch64 architecture), special instructions (e.g instructions which permit the return from an interrupt handler), or an invalid

opcode. This backward analysis is possible because all the architectures for which we need to recover MMU registers have aligned and fixed-size opcodes, thus allowing for a backward linear disassembly.

Results could be improved by extracting the entire function that contains the target instruction (and then also to compute a callgraph to reason inter-procedurally). However, the functions executed during the OS initialization, including those that configure the MMU, are often hand-written in assembly and do not have an easily recognizable structure as those produced by a traditional compiler toolchain. In our experiments, this fact prevented automated disassembly and function recognition tools to identify the required functions.

After extracting the code as described above, MMUShell analyzes it to retrieve the values that were written by the OS in the MMU registers. To implement this analysis we evaluated different popular options, which however failed to support all the architectures in our study. Eventually, we decided to extend the Miasm framework [31]. Miasm supports symbolic analysis by first lifting binaries of different architectures to a custom intermediate representation. However, Miasm was developed to operate on user space binaries that do not contain all the special and privileged instructions we encountered in our experiments. So, we had to extend the framework to include both the MMU registers and the main operands used to load and retrieve their values.

MMUShell implements 11 out of 13 MMU modes described in Section 3. We did not implement ARM Long (which is almost identical to AArch64 Long) as we could not find OS that supports it and RISC-V SV48 (which is an extension of SV39 and, currently, is not supported by any CPU available on the market). The code analysis phase for MIPS64 is currently not supported by Miasm (however, the functioning of MIPS64 MMU is exactly the same as MIPS32, which is supported by our tool).

8 EXPERIMENTS

To evaluate the accuracy and performance of our approach we tested MMUShell on memory dumps running 26 different operating systems with 10 different MMU modes. Each system was configured with 4GB of RAM, unless the OS required a different amount (see Tables 3-6 for more details). We have chosen to limit the amount of RAM to 4GB because many 32bit architectures cannot address more memory and because 4GB also represents a typical memory size that can be found in IoT devices.

To avoid possible inconsistencies [53] and establish a ground truth for our experiments, we run each OS in a virtual machine hosted by a custom version of QEMU [30]. In particular, we modified the emulator to record, for each write operation on MMU registers, the timestamp and the written value. The use of QEMU also allowed us to acquire a copy of the physical memory layout of the emulated machine. For our tests, we installed each OS, booted the machine, and manually used the system by issuing a number of basic commands before acquiring the physical memory.

The operating systems were selected to be as varied as possible in terms of kernel architecture, (including monolithic, microkernel, nanokernel, hybrids, multikernel, and real-time), public availability of the source code, programming languages used to implement the kernel (assembly, C, Rust), year of release (from 1993 to 2020) and purpose (spanning embedded devices, general-purpose, hobbyists project, research, and teaching). The list also deliberately includes some very old (and obsolete) OSs to better prove the generality of our approach. Table 2 lists all the OSs along with the architectures in which we were able to run them. Note that Linux is present twice, first as a minimal distribution (Buildroot [3]) similar to what one might expect to find in embedded devices, and then with a full-fledged Debian installation. The reason is that we were not able to find a popular desktop distribution that runs on all our architectures and MMU configurations. While Debian is supported only by a subset of the Linux Buildroot configurations, it has one order of magnitude more processes (and therefore more MMU structures) to recover.

Table 2. Summary of the operating system analyzed.

OS			Architectures MMU modes									
	Kernel type[1]	Open-source	AArch64 Long	AMD64	ARM32 Short	MIPS32 TLBs	MIPS32 Radix	PowerPC	RISC-V SV32	RISC-V SV48	x86 IA32	x86 PAE
9Front[24]	H	●	○	○	○	○	○	○	○	○	●	○
Barrelfish[17]	U	●	●	●	●	○	○	○	○	○	○	○
Darwin[4]	H	●	○	●	○	○	○	○	○	○	○	○
Embox[5]	R	●	●	○	●	●	○	○	○	○	●	○
FreeBSD	M	●	●	●	○	○	○	○	○	●	●	○
GenodeOS[6]	m	●	○	●	○	○	○	○	○	○	○	○
HaikuOS[7]	H	●	○	●	○	○	○	○	○	○	●	○
HelenOS[8]	m	●	●	●	●	●	○	●	○	○	○	○
Linux Buildroot[3]	M	●	●	●	●	●	●	●	●	●	●	●
Linux Debian	M	●	●	●	●	○	○	●	○	●	●	○
MacOS 9	n	○	○	○	○	○	○	●	○	○	○	○
MacOS X	H	○	○	○	○	○	○	●	○	○	○	○
Minix3[9]	m	●	○	○	○	○	○	○	○	○	●	○
MorphOS[10]	m	○	○	○	○	○	○	●	○	○	○	○
NetBSD	M	●	●	●	●	○	○	●	○	○	●	○
Illumos[29]	M	●	○	●	○	○	○	○	○	○	○	○
QNX[11]	R	○	○	○	○	○	○	○	○	○	●	○
rCore[13]	M	●	●	●	○	○	○	○	●	●	○	○
ReactOS[14]	m	●	○	○	○	○	○	○	○	○	●	○
RedoxOS[15]	m	●	○	●	○	○	○	○	○	○	○	○
vxWorks[19]	R	○	○	●	○	○	○	○	○	○	○	○
Windows 10	H	○	●	●	○	○	○	○	○	○	○	●
Windows 95	M	○	○	○	○	○	○	○	○	○	●	○
Windows NT	H	○	○	○	○	○	○	○	○	○	●	○
Windows XP	H	○	○	●	○	○	○	○	○	○	●	●
XV6[20]	M	●	○	○	○	○	○	○	○	●	○	○

[1] H: hybridkernel, m: microkernel, M: monolithic kernel, U: multikernel, n: nanokernel, R: real-time kernel

The MMU registers values collected by the emulator are used as ground truth. However, it is necessary to filter out those obsolete entries that reference data structures not present anymore in RAM at the time we acquired the memory dump (e.g. radix trees of defunct processes freed by the kernel). While it might still be possible to recover these data structures from memory, their bytes might have been already partially overwritten by other processes, thus making it difficult to locate and validate their correctness. For this reason, we performed a self-consistency check on radix trees and remove those that contained at least one table that is no more valid. We also verified the last time the kernel wrote a value into each MMU register to identify only the last configuration used.

In our proof of concept implementation, we did not pay particular attention to the performance, also because our analysis is a one-time operation that needs to be performed only once. According to our experiments, the slowest operation (the search of IDT tables) required approximately 1 hour for a 4GB dump on a 4 core x86 CPU.

Table 3. RISC-V (32 and 64 bit)

OS	MMU mode	VAS	FN	FP
FreeBSD	SV39	9	-	-
Buildroot[1]	SV32	7	-	-
Buildroot	SV39	7	-	-
Debian	SV39	18	-	-
rCore	SV32	2	-	-
rCore	SV39	2	-	-
XV6	SV39	3	-	-

[1] Does not support 4GiB of RAM.

In the rest of the section, we discuss in detail the results for the three groups of architectures.

8.1 Group I: RISC-V 32 and 64-bit

Table 3 shows the results for RISC-V 32 and 64 bit. In particular, for the various operating systems and MMU modes, the table reports the total number of virtual address spaces (VAS) retrieved from the modified QEMU (i.e., the ground truth). The last two columns show the number of false negatives (i.e., the address spaces that could not be correctly reconstructed by our tool) and false positives (i.e., spurious data structures that were erroneously reported as address spaces).

In this case, MMUShell was able to successfully retrieve all the correct radix trees associated with kernel and user space processes without any false positives. Therefore, our system could fully reconstruct the virtual-to-physical memory mapping for the kernel and each running process.

8.2 Group I: Intel x86 and AMD64

The results for Intel x86 and AMD64 are reported in Table 4. In this case, the tables include two additional columns to report whether the correct Interrupt Descriptor Table (IDT) has been located in the dump and the number of additional candidates IDTs. While the extraction of the IDT is not the goal of our analysis, it plays a very important role in the results because, as we explained in Section 4.2 MMUShell uses this information to validate the candidate page tables. In fact, when MMUShell was not able to retrieve a valid IDT, the number of false positives rose considerably.

MMUShell was able to find all the VAS in all the tested configurations with the exception of rCore running on AMD64, for which it wrongly identifies the IDT. Therefore, even if the correct radix trees were found by the tree reconstruction part of the algorithm, they were later discarded as possible FP because they could not resolve the virtual addresses of the false interrupt handlers.

8.3 Group I: ARM32 and AArch64

For ARM32 and AArch64 our tool needed to retrieve the part of the `TTBCR/TCR_EL1` register (`N` and `EAE` for ARM32, `T0SZ, T1SZ, TG0, TG1` for AArch64) that controls the shape of the processes radix trees. In Table 5 we report the statistics of the retrieved register fields values for each OS and then report the radix trees with a shape compatible with the true positive values of `TTBCR/TCR_EL1` fields. For ARM32 the data flow analysis permitted, for all the OSs, to retrieve both the values of fields of `TTBCR` register as well as all the kernel radix trees and, when `TTBR1` is used, also all the user process ones.

Table 4. INTEL x86 and AMD64

Intel x86 IA32 / PAE MMU modes						
OS	**MMU mode**	**IDT**		**VAS**	**FN**	**FP**
		Found	FP			
9Front	IA32	●	3	61	-	8
Embox	IA32	○	-	1	-	54
FreeBSD	IA32	●	1	21	-	-
HaikuOS	IA32	●	1	18	-	12
Buildroot	IA32	●	1	9	-	1
Buildroot	PAE	●	1	10	-	1
Debian	IA32	●	1	68	-	2
Minix3	IA32	●	-	209	-	2
NetBSD	IA32	●	-	18	-	-
QNX	IA32	●	2	40	-	1
ReactOS	IA32	●	1	17	-	-
Windows 10	PAE	●	5	137	-	34
Windows 95[1]	IA32	●	-	1	-	-
Windows NT	IA32	●	-	14	-	-
Windows XP	IA32	●	-	19	-	-
Windows XP	PAE	●	-	19	-	1

AMD64 4-level MMU mode					
OS	**IDT**		**VAS**	**FN**	**FP**
	Found	FP			
Barrelfish	●	-	31	-	-
Darwin	●	-	16	-	11
FreeBSD	●	2	48	-	3
GenodeOS	○	-	61	-	68
HaikuOS	●	-	17	-	1
HelenOS	○	-	63	-	63
Buildroot	●	-	22	-	2
Debian	●	-	71	-	2
NetBSD	●	-	18	-	-
OmniOS	●	6	4	-	-
rCore	○	1	3	3	-
vxWorks	○	-	2	-	-
RedoxOS	○	-	62	-	33
Windows 10	●	1	127	-	4
Windows XP	●	2	21	-	-

[1] Does not support more than 512MiB of RAM.

Table 5. ARM and AArch64

ARM32 Short MMU mode								
OS	**TTBCR fields**		**Kernel VASs**			**User processes VASs**		
	TP	FP	VAS	FN	FP	VAS	FN	FP
Barrelfish	2/2	1	1	-	6	26	-	6
Embox	2/2	-	1	-	2	-[2]		
HelenOS	2/2	-	33	-	6	-[2]		
Buildroot[1]	2/2	-	14	-	1	-[2]		
Debian[1]	2/2	-	77	-	1	-[2]		
NetBSD	2/2	1	1	-	-	15	-	-

[1] `vexpress-a9` machine supports a maximum of 1GiB of RAM.
[2] Uses only `TTBR0` due to `TTBCR.N=0`[39]

AArch64 Long MMU mode								
OS	**TCR_EL1 fields**		**Kernel VASs**			**User processes VASs**		
	TP	FP	VAS	FN	FP	VAS	FN	FP
Barrelfish	3/4	1	1	-	1	10	-	-
Embox	0/4	-	1	-	1	-	-	-
FreeBSD	4/4	-	1	-	3	14	-	-
HelenOS[1]	4/4	-	2	-	2	49	-	-
Buildroot	4/4	2	1	-	1	5	-	-
Debian	4/4	2	1	-	1	15	-	-
NetBSD	4/4	1	1	-	1	16	-	-
rCore[2]	4/4	-	2	-	-	1	-	-
Windows 10	0/4	-	107	-	1	97	-	2

[1] Does not support more than 1GiB of RAM.
[2] `raspi3` machine supports a maximum of 1GiB of RAM.

For AArch64 instead, in two cases (Embox and Windows 10) it was not possible to recover the four `TCR_EL1` fields necessary to reconstruct the radix tree and VAS shape. In another case (Barrelfish) we failed to retrieve only one of these fields. For all the tested configurations MMUShell recovered all the radix trees for kernel and user space processes.

Table 6. PowerPC and MIPS32

PowerPC BAT+SDR1 MMU mode						
OS	**BAT registers**		**Hash Table**			
	Subregs.	BAT	Location	Size	FP	SDR1
HelenOS[1]	8/16	0/8	●	●	-	○
Buildroot[2]	16/16	8/8	●	●	-	●
Debian[2]	16/16	8/8	●	●	-	●
MacOS 9[3]	12/16	4/8	●	●	-	○
MacOS X[2]	-[4]	-	●	●	-	●
MorphOS[2]	6/16	3/8	●	●	-	○
NetBSD	7/16	2/8	●	●	6	○

[1] `g3beige` machine supports a maximum of 1GiB of RAM.
[2] `mac99` machine supports a maximum of 2GiB of RAM.
[3] Does not support more than 1GiB of RAM.
[4] See text for more details.

MIPS32 TLBs / Radix Tree MMU modes				
OS	**MMU mode**	**MMU registers**		
		TP	FN	FP
Embox	TLBs	13	-	-
HelenOS[1]	TLBs	13	-	2
Buildroot[2]	Radix	12	1	6
Debian[2]	TLBs	13	-	2

[1] `malta` machine supports a maximum of 2GiB of RAM.
[2] Does not support more than 256MiB of RAM on `malta` machine.

8.4 Group II: PowerPC 32-bit

In the PowerPC architecture, the address resolution process involves the use of the BAT registers and an inverted page table pointed by the SDR1 register. Each BAT register is physically implemented as a couple of 32-bit sub-registers (BATU and BATL) which are loaded separately by the CPU but used as an atomic unit by the MMU. The BATU subregister contains the address of the translated virtual address block and its size, while the companion BATL contains the physical address of the correspondent block plus its access permissions. Thus, even if only one of the two subregisters is automatically retrieved from a memory dump, it still provides important information to the human analyst about the layout of the virtual or the PAS. For this reason, Table 6 reports the number of physical subregisters recovered by MMUShell and the number of complete couples, corresponding to the entire BAT register.

Our code analysis phase was able to completely recover the content of the BAT set for Linux, partially for Morphos MacOS 9 and NetBSD, and only isolated subregisters for HelenOS. This is due to the lack of interprocedural analysis and the incomplete support of PowerPC MMU instructions in the Miasm framework. It is important to note that MacOS X defines different address blocks for every process, thus changing the content of the BAT registers at every context switch. As a result, the values cannot be recovered without an in-depth knowledge of the data structures used by the OS kernel. On the contrary, MMUShell can retrieve, for every tested OS, the physical address and size of the hash table and, in some cases, also the content of the SDR1 register. Thus, MMUShell could fully reconstruct, for each OS, the mapping between segmented and physical address spaces, but was not able to completely reconstruct the mapping between the virtual and the segmented spaces. Furthermore, in the case of Linux and partially of MorphOS, MacOS 9 and NetBSD, by retrieving also BAT registers, MMUShell could also list the blocks of VAS directly mapped to the physical ones.

8.5 Group III: MIPS 32-bit

As discussed in Section 6.1 for MIPS32 we need to retrieve the partial content of 13 different MMU registers to identify uniquely the MMU configuration of the machine. While MMUShell was very accurate in retrieving those values, as explained in Section 6.1, these registers provide only half of the picture and a human analyst still need to manually reverse engineer the algorithm used by the kernel to map segments to physical pages.

9 APPLICATION TO REAL HARDWARE

In order to demonstrate the usefulness and validate the functionality of MMUShell in a real-world scenario and demonstrate how our solution can help in a real investigation, we have conducted a forensic analysis on a real hardware running an uncommon OS. For this experiment, we chose a Raspberry PI 3B+[12], an AArch64 board that also supports ARM32 and for which we were able to acquire the raw memory content by using the integrated JTAG controller. On the board, we have installed RISC OS[16], a partially-open operating system used digital TV decoders, factory automation systems [16]. This is an excellent example of a popular but not well-known OS that runs on embedded devices but that is not supported by QEMU (and this is why we could not include it as part of our controlled experiments).

By using only the board documentation, we created the necessary YAML file that describes the physical memory layout and the initial default value of the CPU registers. Moreover, through the JTAG interface, we are also able to dump the value of the MMU registers at runtime, and later use their content as ground truth to validate the results automatically extracted from the memory by MMUShell.

MMUShell was able to correctly recover all MMU registers values (as confirmed by our ground truth values collected during the dump phase) and reconstruct the kernel/user radix tree without any false positive (RISC OS, as other real-time OSs analyzed in Section 8, uses only one radix tree for all the processes running in the system).

As a result, the analyst now can easily reconstruct the mapping between the PAS and the VASes of the kernel and the user processes. This permits to correctly resolve the pointers contained in the dump and allows to start the analysis of the data structures contained in it.

It is important to stress again that MMUShell results derive only from the memory dump and the public information on the hardware architecture. No OS-specific rules or information is necessary for our analysis.

10 TOWARDS OS AGNOSTIC MEMORY FORENSICS

To conclude the paper, in this section, we want to show a possible use of our technique to perform a preliminary memory analysis of a dump acquired from an unknown operating system. It is important to stress that there are no tools or techniques available today that can operate in these settings. While our work is only the first step towards an OS-agnostic analysis, by using our virtual memory reconstruction we can already automatically extract all running processes and derive information about them without any knowledge of the OS internals, executable file format, or mechanism used to interact with the kernel.

In fact, MMUShell can export each virtual address space as a self-contained ELF Core dump file, which maps each physical memory page to its corresponding virtual one inside a `LOAD` segment – along with right the page access permissions. This allows an analyst to obtain a faithful representation of the process VAS. In addition, each ELF can also be easily opened and analyzed by using popular static analysis tools – such as IDA Pro. Each ELF generated by MMUShell corresponds to a process running in the system (either kernel, privileged services, user space programs, or dead processes). This can already be used by a security analyst to retrieve the process running in any IoT device, independently from the architecture or the running OS. The list of running processes in a compromised device could be easily compared with that acquired on a clean system to detect anomalous processes that might need to be manually reverse engineered.

By using the extracted information, in the rest of the section, we discuss which other properties could be identified for either a single or an ensemble of processes. Unlike the other part of our work, to perform this task we rely on a set

of heuristics based on the usage of the MMU by a generic OS, which are reasonable in multiprocess systems with an MMU.

- **Kernel and privileged services identification** – The access permissions of the pages in the process VAS allows to discriminate whether a process is a privileged one (which runs at kernel-level and does not permit, in general, any write access to its space to user mode processes or does not map any user space accessible pages) or a user space one (which can have kernel pages mapped in its address space but without write access).
- **Executable file format identification** – It is possible to use tools, like `binwalk`[38], which permit to identify a known executable file format headers (e.g., ELF, PE, COFF, etc.) in the VAS of a process, thus recovering additional information about the structure of the executable file loaded supported by the unknown OS.
- **Core shared libraries and data pages** – Modern OSs that support multiple processes and virtual memory usually rely on a special library to support the executable loading process before starting its execution (e.g. `ld-linux.so` in Linux) or core libraries and data pages (e.g. `libc.so`) to easily interface the user space process with the kernel through syscalls. To reduce memory consumption, physical pages that contain core shared libraries are present in a unique copy that is mapped, without write permissions, into the majority of the user space VAS. We can find these shared resources by looking for regions in each user process that point to the same physical pages and which are present in at least 90% of the user processes.
- **Minor shared libraries, shared code pages, and interprocess shared memory regions** – By using the same technique without the 90% threshold we identify less common shared libraries, pages of code shared among processes (due to syscall similar to the UNIX `fork`) and, including also pages with permission `RW-`: shared regions used by multiple processes to efficiently share data among them.
- **Special regions** – By using our data, we can identify a set of virtual address ranges which are always mapped at the same physical addresses in all processes. These (e.g., the `vsyscall` pages in Linux). could play a particular role in the target OS architecture.
- **Stack identification** – The stack pages of a process must have at least `RW-` permissions (in general, for security reasons, executable permission is missing but in some architecture, as x86-32, this is not possible) and they must be contiguous. Furthermore, if we assume that the stack frame pointer register is used to track old previous stack frame start, stack pages contain adjacent couples of values composed by an instruction address and a stack address. This pattern is directly related to the functioning of stack-based architectures: a `CALL` instruction to a routine pushes the address of the immediately following instruction on the stack (the return address), followed by the address of the previous stack frame contained in the stack frame pointer register. Stack pages can be identified by looking for pages with the correct permissions set and which contains the biggest number of valid couples of return addresses and stack frame pointer addresses. It is possible to identify return addresses because they are virtual addresses that belong to code pages (`R-X`) and tools like `radare2`[32], which can load our VAS due to its representation as ELF core file, permit, also in case of non-fixed length opcode architecture like x86, to perform backward decompilation and check if the previous instruction pointed by the return address is a real `CALL` instruction removing a great number of false positives. At the same time, stack frame pointer addresses are supposed to belong to the same page on which they reside (supposing the stack frame is not bigger than a page size or astride them).
- **Approximated entry point** – Directly related to stack identification, we can derive an approximated position of the entry point of the executable: the bottom of the stack must contain the address of an instruction belonging

to the first function executed by the program or by the loader library, permitting to identify approximately which part of code could contain the real entry point.

- **Processes relations** – A manual analysis of which processes share writable memory areas and code areas permit to identify relations among processes. A multiplicity of stack candidates (if we suppose that they are not false positives) may indicate the presence of different threads associated with the process: they will share the same code pages but each will have its own private stack. Finally, it is possible to calculate a TLSH[18] fuzzy hash on the content of the single physical pages of two or more processes to check similarities in code and cluster them also in case of code relocations as shown in [52].

10.1 Experiment

For this experiment, we have chosen an x86-32 installation of the QNX operating system. The x86 architecture has been chosen because, as discussed in previous sections, radix-tree based architectures (such as INTEL, ARM, RISC-V and partially Power) permit a complete reconstruction of the VASs. This partially complicates our analysis, like the MMU mode supported by x86-32 which does not provide any bit to set a page as executable, so all the accessible pages in user mode are at least readable and executable.

We choose the QNX OS because it is a closed-source operating system that does not have a lot of public information about its internals and it was poorly studied by the memory forensics community.

As ground truth, we have extracted, through `/proc/PID/as` interface of QNX, information about the virtual memory space layout of all processes running in the system and their associated threads. MMUShell, as shown in Table 4 correctly identified all the valid VASs present in the memory dump (40), including three VASs belonging to processes that already terminated (at dump time only 37 processes were running on the system).

By using the heuristics described above, our tool was able to automatically distinguish all processes in user processes (36) from the kernel itself (1) and determine that QNX uses the ELF format. It also identified two core libraries mapped by most processes (which correspond to `ldqnx.so.2` and `libc.so.3`), as well as many other libraries used by a subset of the user space programs and writable memory regions shared among different processes. For 34 processes, our system correctly identified the stack and the instructions belonging to the first function executed by the programs. Furthermore, the presence of multiple stacks on some processes confirmed their multithread nature.

In summary, the analysis and experiment presented in this section show a possible practical use of our tool to perform a preliminary memory analysis for any operating system currently unsupported by existing forensic tools.

11 LIMITATIONS AND FUTURE EXTENSIONS

We now discuss some limitations of our approach, either due to engineering problems or to the OS-agnostic nature of our work. We believe this list can also provide some ideas for possible extensions and future research in the field:

- Even after our extensions, Miasm still lacks support for several opcodes, in particular for PowerPC and MIPS. This further emphasizes the need of our community for a binary analysis framework able to operate in a symbolic way on privileged code for non-x86 architectures.
- On some architectures, like Intel x86/AMD64 and ARM32/AArch64, when a page table entry has the present bit unset, the MMU simply ignores the content of the remaining bits and raises an interrupt if the CPU tries to access virtual memory addresses potentially resolved by it. However, as extensively shown in numerous works, OSs like Windows, Linux and OSX fill ignored bits of invalid page table entries with values that efficiently permit

them to manage extensions to the traditional model of virtual address space. Examples of this behaviour are the classical swap to secondary storage, or the compressed swap stores and transition pages implemented in Windows [43, 55], the `PROT_NONE` pages in Linux [45], OSX memory queues [22], or the possibility for Linux and OSX to compress at runtime part of the RAM content [54] to increase the memory available to the system. All these techniques save a reference to the original page and additional metadata (e.g. an offset within a swap file) into the invalid page table entry relative to the page itself. When the CPU tries to access a virtual address that belongs to those pages, the MMU raises an exception and the OS checks the invalid bits in the entry. If the OS determines, for example, that the page is part of a compressed memory region, it decompresses the page, restores the association between the page table entry and the page, set the valid bit, and restart the execution of the process that has generated the previous illegal access.

Our technique is completely blind to these types of behaviour, which is strongly OS-dependent. However, these extensions do not break the inviolable MMU constraints since they are based on the use of ignored bits in invalid page table entries. To access to this type of pages a partially OS-independent approach could consist in the identification of the page fault handler in the interrupt table and its emulation by using the faulting page table entry as a parameter (as specified by the CPU ISA). This automated approach may allow to gain some information about pages marked as invalid but instead treated in special ways by the OS. As a last resort, a human analyst could perform a manual analysis of the page fault handler to understand how it works and how to manage invalid page entries. Furthermore, in the last few years, the memory forensics community has deeply studied the memory management subsystems of Windows, Linux and OSX and has implemented inside tools as Volatility and Rekall[62] modules able to treat correctly invalid page table entries, thus drastically increasing the analysis capabilities of these systems.

- We found very little information that can be used to build validation rules to reduce the number of false-positives. However, in our experience, human experts can often tell the difference between real VASs and false-positive ones, suggesting that maybe a ML classifier can be trained to reduce the FP. This is an interesting research direction that we leave as future work.
- Even though we performed all our experiments in a 'benign' scenario, as we have specified in various parts of the paper the MMU requires that the OS strictly respects a number of inviolable constraints. A rootkit (or other forms of a compromised system) is no exception to this rule and cannot bypass our MMU constraints.

  However, even though we did not encounter this case in any of the 26 OSs we tested, it is possible that the code responsible to configure the MMU is intentionally deleted from memory after the setup is complete. In this (possibly adversarial) setting, bridging the semantic gap may become very difficult, if even possible at all.
- In our work, we assumed that the OS under analysis is running on a bare-metal machine or inside a VM for which we can get a memory dump containing only the VM memory. However, it is also possible to consider the possibility in which the OS is running as a hypervisor for other VMs. In this case, a memory dump of the full system will contain also the VMs memory. All the treated architectures in this work (but PowerPC and MIPS32) support virtualization in various forms by using, in general, an extension of the already presented MMU modes. In particular, radix-tree architectures extend the existing MMU modes introducing radix-trees composed of a different type of page tables associated with each VMs. Each of these new radix-trees maps the VM physical page addresses, valid inside the associated VM, to physical pages exiting on the bare-metal machine. An interesting development of our technique could support also the reconstruction of the MMU hypervisor in-memory data structure associated with each single VM which only requires to define new structure signatures to match this

new type of data. The special case of Intel x86, memory forensics of hypervisor has also been covered by Graziano et al [36].

12 DISCUSSION

In this study, we discussed the broad range of strategies adopted by the MMU of different CPU architectures to translate virtual to physical addresses. We have presented the first complete study of the problem, and our solution to bridge the semantic gap of virtual-to-physical address translation in a *zero-knowledge scenario*. We have also highlighted that, due to the peculiarities of some architectures (PowerPC, Power and MIPS), a full automatic complete reconstruction of the virtual address spaces without a deep knowledge of the OS internals is, in the general case, impossible. This fact is not a limitation of our technique but an intrinsic limitation imposed by those ISAs.

We have shown that our solution, based on a combination of OS-agnostic MMU derived constraints (to identify candidate data structures) and static code analysis (to retrieve the setup of system registers) can extract the maximum amount of information, permitted by the CPU ISA, about the virtual address spaces of the running processes. The technique does not use *any* knowledge on the running OS or tailor-made heuristics.

MMUShell, the proof-of-concept tool implementing our technique, was able to recover all the VAS in the vast majority of the experiments and also on a dump extracted from a physical hardware device, proving the feasibility of our approach in real-world scenarios. A simple OS-agnostic memory forensics analysis was conducted on a close-sources OS to show the potential of our tool in real-world scenarios.

The amount of time required to add the support for a new architecture would largely depend on the type of virtual-to-physical address translation strategy implemented by its MMU, the analyst knowledge of its internal details, and the quality of the available documentation. For instance, it took us only a few days to support RISC-V (to read the documentation, compile a list of constraints, create a valid strategy to reconstruct radix-trees, and perform some tests) while MIPS and ARM64 required roughly a few weeks each, due to the variability of the available options, complexity and peculiarity of their MMU modes.

13 RELATED WORK

The more general problem of extracting data structures from memory dumps has already been treated in several previous studies, even if with different goals. For instance, Dolan-Gavitt et al.[28] have proposed a signature-based approach to define invariants of kernel data structures and generate signatures for their recovery in memory dumps. A similar approach, but based also on probabilistic inferences, was developed in[46]. Another signature-based work is "Multi-Aspect, Robust, and Memory Exclusive Guest OS Fingerprinting"[37]. In this work *Yufei et al.* use invariants, derived from kernel code and data structures extracted from a memory dump of a system running Linux, to generate signatures to identify the kernel version. Other techniques use signatures derived from the topology of data structures (SigGraph[47], SigPath[61] and Mace[33]) or neural networks to label data structures in memory dumps (DeepMem[59]). Other works have focused instead on the reconstruction of the shape, content and primitive types of the data-structures [48][49][57][58][60]. However, all these techniques work on data structures that contain virtual addresses, thus assuming the problem of address spaces translation was already solved by other means, or they require multiple dumps of the same version of the operating system to train an algorithm. Saur and Grizzard [56] are the first to propose an approach to reconstruct radix trees for Intel x86-32 from both Windows XP and Linux memory dumps. Their parsing technique is based on rules derived from the ISA specifications and tailor-made heuristics derived from the OSs internals, which allows the authors to retrieve virtual and physical address spaces of hidden processes running on a sampled system.

Their approach, however, is limited by the non-universality of their parsing rules, which contains heuristics based on how Linux and Windows kernels use page tables, making it impossible to use on dumps of different OSs. Finally, a work more similar to ours for its focus on CPU ISA but with a different goal is [36]. In this work, the authors use OS-agnostic parsing rules to detect, in memory dumps, virtual machine control structures, used by the Intel CPUs to maintain the state of virtual machines.

14 CODE AVAILABILITY

We will release MMUShell as an open-source project[51], together with the part of the dataset not covered by particular OS license restrictions.

15 ACKNOWLEDGMENTS

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

REFERENCES

[1] Ali Reza Arasteh and Mourad Debbabi. 2007. Forensic memory analysis: From stack and code to execution history. *Digital Investigation* 4 (2007), 114–125.
[2] Various authors. 2022. *BlackBag Technologies.* BlackBag Technologies. https://www.blackbagtech.com/
[3] Various authors. 2022. *Buildroot.* Buildroot Association. https://buildroot.org/
[4] Various authors. 2022. *Darwin OS.* Apple Inc. https://github.com/apple/darwin-xnu
[5] Various authors. 2022. *Embox OS.* Embox Ltd. https://www.embox.rocks/
[6] Various authors. 2022. *Genode OS.* Genode Labs. https://genode.org/
[7] Various authors. 2022. *Haiku OS.* Haiku Inc. https://www.haiku-os.org/
[8] Various authors. 2022. *HelenOS.* HelenOS Community. http://www.helenos.org/
[9] Various authors. 2022. *Minix3 OS.* VU University. https://www.minix3.org/
[10] Various authors. 2022. *MorphOS.* MorphOS Development Team. https://www.morphos-team.net/
[11] Various authors. 2022. *QNX.* BlackBerry Limited. https://www.qnx.com
[12] Various authors. 2022. *Raspberry PI.* RaspberryPI Foundation. https://www.raspberrypi.org/
[13] Various authors. 2022. *rCore.* rCore Developers. https://github.com/rcore-os/rCore
[14] Various authors. 2022. *ReactOS.* ReactOS Team and Contributors. https://reactos.org/
[15] Various authors. 2022. *Redox OS.* Redox Developers. https://www.redox-os.org/
[16] Various authors. 2022. *RISC OS Open.* RISC OS Open Limited. https://www.riscosopen.org
[17] Various authors. 2022. *The Barrelfish OS.* ETH Zurich. http://www.barrelfish.org/
[18] Various authors. 2022. *TLSH - Trend Micro Locality Sensitive Hash.* Trend Micro. https://github.com/trendmicro/tlsh
[19] Various authors. 2022. *vxWorks.* Wind River Systems. https://www.windriver.com/products/vxworks/
[20] Various authors. 2022. *XV6.* Massachusetts Institute of Technology. https://github.com/mit-pdos/xv6-riscv
[21] Jeffrey Burt. 2020. *Alibaba On The Bleeding Edge Of RISC-V With XT910.* The Next Platform. https://www.nextplatform.com/2020/08/21/alibaba-on-the-bleeding-edge-of-risc-v-with-xt910/
[22] Andrew Case, Ryan D Maggio, Modhuparna Manna, and Golden G Richard III. 2020. Memory Analysis of macOS Page Queues. *Forensic Science International: Digital Investigation* 33 (2020), 301004.
[23] Michael Cohen. 2014. Rekall Memory Forensic Framework.
[24] 9Front community. 2022. *9Front OS.* 9Front community. http://9front.org/
[25] Emanuele Cozzi, Mariano Graziano, Yanick Fratantonio, and Davide Balzarotti. 2018. Understanding linux malware. In *2018 IEEE symposium on security and privacy (SP).* IEEE, IEEE Computer Society, Los Alamitos, CA, USA, 161–175.
[26] Anthony Cozzie, Frank Stratton, Hui Xue, and Samuel T. King. 2008. Digging for Data Structures. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation* (San Diego, California) *(OSDI'08).* USENIX Association, USA, 255–266.
[27] Brendan Dolan-Gavitt, Tim Leek, Michael Zhivich, Jonathon Giffin, and Wenke Lee. 2011. Virtuoso: Narrowing the Semantic Gap in Virtual Machine Introspection. In *Proceedings of the 2011 IEEE Symposium on Security and Privacy (SP '11).* IEEE Computer Society, USA, 297–312. https://doi.org/10.1109/SP.2011.11

[28] 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* (Chicago, Illinois, USA) *(CCS '09)*. Association for Computing Machinery, New York, NY, USA, 566–577. https://doi.org/10.1145/1653662.1653730

[29] OmniOS Community Edition. 2022. *OmniOS*. Various authors. https://omniosce.org/

[30] Fabrice Bellard et al. 2022. *QEMU - Generic and open source machine and userspace emulator and virtualizer*. QEMU Community. https://www.qemu.org/

[31] Fabrice Desclaux et al. 2022. *Miasm - Reverse engineering framework*. CEA. https://github.com/cea-sec/miasm

[32] Sergi Alvarez et al. 2022. *Radare2 - Libre and Portable Reverse Engineering Framework*. Radare2 community. https://rada.re/n/

[33] Qian Feng, Aravind Prakash, Heng Yin, and Zhiqiang Lin. 2014. Mace: High-coverage and robust memory analysis for commodity operating systems. In *Proceedings of the 30th annual computer security applications conference*. Association for Computing Machinery, USA, 196–205.

[34] FreeScale. 2005. *Programming Environments Manual for 32-Bit Implementations of the PowerPC Architecture*. FreeScale.

[35] 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 *IEEE Symposium on Security and Privacy, SP 2012, 21-23 May 2012, San Francisco, California, USA*. IEEE Computer Society, USA, 586–600. https://doi.org/10.1109/SP.2012.40

[36] Mariano Graziano, Andrea Lanzi, and Davide Balzarotti. 2013. Hypervisor Memory Forensics. In *Research in Attacks, Intrusions, and Defenses*, Salvatore J Stolfo, Angelos Stavrou, and Charles V Wright (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 21–40.

[37] Yufei Gu, Yangchun Fu, Aravind Prakash, Zhiqiang Lin, and Heng Yin. 2014. Multi-aspect, robust, and memory exclusive guest os fingerprinting. *IEEE Transactions on Cloud Computing* 2, 4 (2014), 380–394.

[38] Craig Heffner. 2022. *Binwalk*. ReFirm Labs. https://github.com/ReFirmLabs/binwalk

[39] ARM Holdings. 2018. *ARM Architecture Reference Manual, ARMv7-A and ARMv7-R edition*. ARM Holding.

[40] ARM Holdings. 2020. *ARM Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile*. ARM Holdings.

[41] IBM. 2017. *Power ISA. Version 3.0B*. IBM.

[42] Intel. 2020. *Intel 64 and IA-32 Architectures–Software Developer's Manual–Volume 3 (3A, 3B, 3C & 3D): System Programming Guide*. Intel Corporation.

[43] Jesse D Kornblum. 2007. Using every part of the buffalo in Windows memory analysis. *Digital Investigation* 4, 1 (2007), 24–29.

[44] Kevin Krewell. 2017. *Western Digital Gives A Billion Unit Boost To Open Source RISC-V CPU*. Forbes. https://www.forbes.com/sites/tiriasresearch/2017/12/06/western-digital-gives-a-billion-unit-boost-to-open-source-risc-v-cpu/

[45] Jamie Levy. 2015. *Using PROT_NONE on Linux*. Volatility Labs. https://volatility-labs.blogspot.com/2015/05/using-mprotect-protnone-on-linux.html

[46] Zhiqiang Lin, Junghwan Rhee, Chao Wu, Xiangyu Zhang, and Dongyan Xu. 2012. Discovering semantic data of interest from un-mappable with confidence. In *Proceedings of the 19th Network and Distributed System Security Symposium (NDSS)*. The Internet Society, USA.

[47] Zhiqiang Lin, Junghwan Rhee, Xiangyu Zhang, Dongyan Xu, and Xuxian Jiang. 2011. SigGraph: Brute Force Scanning of Kernel Data Structure Instances Using Graph-based Signatures. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*. The Internet Society, San Diego, California, USA. https://www.ndss-symposium.org/ndss2011/siggraph-brute-force-scanning-of-kernel-data-structure-instances-using-graph-based-signatures

[48] Zhiqiang Lin, Xiangyu Zhang, and Dongyan Xu. 2010. Automatic reverse engineering of data structures from binary execution. In *Proceedings of the 11th Annual Information Security Symposium*. CERIAS - Purdue University, USA, 1–18.

[49] Daniel Mercier, Aziem Chawdhary, and Richard Jones. 2017. dynStruct: An automatic reverse engineering tool for structure recovery and memory use analysis. In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE Computer Society, IEEE Computer Society, USA, 497–501.

[50] Mips. 2015. *MIPS Architecture For Programmers Vol. III: MIPS32 / microMIPS32 Privileged Resource Architecture*. Imagination Technologies.

[51] Andrea Oliveri. 2022. *MMUShell*. Eurecom. https://github.com/eurecom-s3/mmushell

[52] Fabio Pagani, Matteo Dell'Amico, and Davide Balzarotti. 2018. Beyond Precision and Recall: Understanding Uses (and Misuses) of Similarity Hashes in Binary Analysis. In *Proceedings of the Eighth ACM Conference on Data and Application Security and Privacy* (Tempe, AZ, USA) *(CODASPY '18)*. Association for Computing Machinery, New York, NY, USA, 354–365. https://doi.org/10.1145/3176258.3176306

[53] Fabio Pagani, Oleksii Fedorov, and Davide Balzarotti. 2019. Introducing the temporal dimension to memory forensics. *ACM Transactions on Privacy and Security (TOPS)* 22, 2 (2019), 1–21.

[54] Golden G Richard III and Andrew Case. 2014. In lieu of swap: Analyzing compressed RAM in Mac OS X and Linux. *Digital Investigation* 11 (2014), S3–S12.

[55] O. Sardar and D. Andonov. 2019. *White paper: Finding Evil in Windows 10 Compressed Memory*. Technical Report. FireEye, 601 McCarthy Blvd. Milpitas, CA 95035. 11 pages. https://www.fireeye.com/content/dam/fireeye-www/blog/pdfs/finding-evil-in-windows-10-compressed-memory-wp.pdf

[56] Karla Saur and Julian B. Grizzard. 2010. Locating x86 paging structures in memory images. *Digital Investigation* 7, 1-2 (oct 2010), 28–37. https://doi.org/10.1016/j.diin.2010.08.002

[57] Asia Slowinska, Traian Stancescu, and Herbert Bos. 2010. DDE: dynamic data structure excavation. In *Proceedings of the first ACM asia-pacific workshop on Workshop on systems*. USENIX Association, New Delhi, India, 13–18.

[58] Asia Slowinska, Traian Stancescu, and Herbert Bos. 2011. Howard: A Dynamic Excavator for Reverse Engineering Data Structures. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*. The Internet Society, San Diego, California, USA.

[59] Wei Song, Heng Yin, Chang Liu, and Dawn Song. 2018. DeepMem: Learning Graph Neural Network Models for Fast and Robust Memory Forensic Analysis. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (Toronto, Canada) *(CCS '18)*. Association for

Computing Machinery, New York, NY, USA, 606–618. https://doi.org/10.1145/3243734.3243813

[60] Katerina Troshina, Yegor Derevenets, and Alexander Chernov. 2010. Reconstruction of composite types for decompilation. In *2010 10th IEEE Working Conference on Source Code Analysis and Manipulation*. IEEE Computer Society, IEEE Computer Society, USA, 179–188.

[61] David Urbina, Yufei Gu, Juan Caballero, and Zhiqiang Lin. 2014. Sigpath: A memory graph based approach for program data introspection and modification. In *European Symposium on Research in Computer Security*. Springer, Springer International Publishing, Cham, 237–256.

[62] Sebastian Vogl and Blaine Stancill. 2019. *Rekall support for Windows 10 memory compression*. FireEye. https://github.com/mandiant/win10_rekall/blob/win10_compressed_memory/rekall-core/rekall/plugins/windows/win10_memcompression.py

[63] Volexity. 2022. *Volexity*. Volexity. https://www.volexity.com/

[64] Aaron Walker. 2017. Volatility framework: Volatile memory artifact extraction utility framework.

[65] Asanovic K. Waterman A. (Ed.). 2019. *The RISC-V Instruction Set Manual, Volume II: Privileged Architecture, Document Version 20190608-Priv-MSU-Ratified*. RISC-V Foundation, USA.

[66] William G. Wong. 2020. *Ada and RISC-V Secure Nvidia's Future*. Endeavour Business Media. https://www.electronicdesign.com/markets/automotive/article/21121197/ada-and-riscv-secure-nvidias-future

[67] Haiquan Xiong, Zhiyong Liu, Weizhi Xu, and Shuai Jiao. 2012. Libvmi: A Library for Bridging the Semantic Gap between Guest OS and VMM. In *Proceedings of the 2012 IEEE 12th International Conference on Computer and Information Technology (CIT '12)*. IEEE Computer Society, USA, 549–556. https://doi.org/10.1109/CIT.2012.119

A POWER ISA (64-BIT)

The Power architecture is a RISC architecture introduced by IBM in the early '90s. Starting from the original POWER ISA, many architectural variants have been developed during the last 30 years, including variants for gaming consoles. However, only two of those architectures are still currently developed and supported: PowerPC, already described in Section 5.1, and Power ISA.

A.1 MMU Internals

The Power ISA is a server-class 64-bit bi-endian architecture focused on para-virtualization: a hypervisor OS manages all the resources of the system, including MMU registers and in-memory tables, partitioning them between the para-virtualized machines (called Logical Partitions or LPARs). An OS running inside an LPAR needs specific support for this environment in order to interface with the hypervisor. CPUs based on the latest Power ISA revision [41], like the IBM POWER9 family, use the `PTCR` register to point to an in-memory hypervisor-reserved table called Partition Table, which contains pointers to relevant memory management structures of the hypervisor and every LPAR running on the system. Each entry of this table points to one of two different types of structures automatically used by the MMU to translate virtual addresses of the LPAR (or the hypervisor): an Inverted Page Table (along with an optional segment table used instead of the segment registers) or the root of the radix tree of the kernel (with a related process table containing roots of process radix trees).

The Inverted Page Resolution mode works in the same way as for PowerPC (without `BAT` registers) with, in addition, the support for 64-bit VASs and multiple page sizes. In this mode, all the LPAR share the hash table with the hypervisor calling it to fill the hash table and the segment registers (the optional segment table is filled directly by the OS).

In radix tree mode, the translation of a virtual address to a physical one goes through a double translation: a radix-tree, allocated by the guest OS, translates the virtual address to a pseudo-physical address. Then, a radix tree associated with the LPAR and maintained by the hypervisor, translates the pseudo-physical address to a real physical one. Radix trees have a predefined number of levels, page sizes and table entry format which permits to distinguish between directory and page tables.

A.2 Analysis

On a Power ISA machine we can perform a physical memory dump in two different ways: a dump of the entire machine memory or a dump of a single LPAR. In the first case, we can recover the value contained in the `PTCR` register using code analysis or parsing for the Partition Table using rules derived from the ISA. If an LPAR or the hypervisor uses the radix tree MMU or the hash table with the segment table it is possible to recover the entire V2P address mapping starting from the Partition Table. Instead, if they use the hash table MMU mode with the segment registers set, we can only reconstruct the segmented to physical translation, in the same ways as we described for PowerPC.

Finally, if we work with an LPAR memory dump we can recover the structure of the guest radix trees but if the LPAR uses the hash table mode, we cannot recover any information about the V2P address translation because the hash table resides in the hypervisor memory.

A.3 Limitations

We have decided to not include Power ISA in Section 3 for four reasons: first of all, this architecture is strongly oriented to virtualization and the MMU operating modes are structured with this perspective, opening the problem of the privilege level to whom the memory dump is taken, which influences, in turn, the features recoverability. Second, QEMU does not support the emulation of a complete Power machine running an LPAR—making it impossible to test the reconstruction techniques in the case of a hypervisor memory dump. Third, Miasm does not support Power ISA, making it impossible to perform the code analysis phase. Last but not least, the only operating system we could find that runs in hypervisor mode is Linux and this lack of test cases would not allow to have any statistical significance on the obtained results.

B STRUCTURE SIGNATURES AND VALIDATION RULES

Table 7. Structure signatures (○) and validation rules (●) for each MMU mode implemented in MMUShell.
See ISA's documentation[65][42][39][40][34] for more details.

AArch64 (64 bit) Long MMU mode		
Object	**Type**	**Rule**
PTL0/PTL1/PTL2/PTL3 Entry	○	Size(Entry) = 8 Bytes
Empty PTL0/PTL1/PTL2/PTL3 Entry	○	Entry[0] = 0
PTL3 reserved entry 4/16/64Kib granule	○	Entry[0] = 1 ∧ Entry[1] = 0
PTL3 entry 4/64KiB granule	○	Entry[0,1] = 1 ∧ Entry[SH] ≠ 1
PTL3 entry 16KiB granule	○	Entry[0,1] = 1 ∧ Entry[SH] ≠ 1 ∧ Entry[12,13] = 0
Block PTL2 entry 4KiB granule	○	Entry[0] = 1 ∧ Entry[1] = 1 ∧ Entry[12...15] = 0 ∧ Entry[SH] ≠ 1 ∧ Entry[17...20] = 0
Block PTL2 entry 16KiB granule	○	Entry[0] = 1 ∧ Entry[1] = 1 ∧ Entry[12...15] = 0 ∧ Entry[SH] ≠ 1 ∧ Entry[17...24] = 0
Block PTL2 entry 64KiB granule	○	Entry[0] = 1 ∧ Entry[1] = 1 ∧ Entry[12...15] = 0 ∧ Entry[SH] ≠ 1 ∧ Entry[17...28] = 0
Block PTL1 entry 4KiB granule	○	Entry[0] = 1 ∧ Entry[1] = 1 ∧ Entry[12...15] = 0 ∧ Entry[SH] ≠ 1 ∧ Entry[17...29] = 0
PTL0/PTL1/PTL2 pointer 4KiB granule	○	Entry[0,1] = 1 ∧ Entry[Address] ∈ RAM
PTL0/PTL1/PTL2 pointer 16KiB granule	○	Entry[0,1] = 1 ∧ Entry[12,13] = 0 ∧ Entry[Address] ∈ RAM
PTL0/PTL1/PTL2 pointer 64KiB granule	○	Entry[0,1] = 1 ∧ Entry[12...15] = 0 ∧ Entry[Address] ∈ RAM
Kernel Radix tree	●	∃ DataPage | ReadOpcode(`ESR_EL1, FAR_EL1, ELR_EL1`) ∈ DataPage ∧ ∃ DataPage | WriteOpcode(`TTBR0_EL1, TCR_EL1`) ∈ DataPage ∧ ∃ DataPage | ExecOpcode(`ERET`) ∈ DataPage
Kernel Radix tree	●[1]	∃ Entry ∈ {Tables of RadixT} | Entry[AP[0,1]] = 0 ⇒ RadixT ∈ {Accepted Kernel radix trees}
User Radix tree	●	∃ DataPage | ExecOpcode(`RET`) ∈ DataPage ∧ ∃ DataPage | ExecOpcode(`BLR`) ∈ DataPage
User Radix tree	●[2]	∃ Entry ∈ {Tables of RadixT} | Entry[AP[1]] = 1 ⇒ RadixT ∈ {Accepted User radix trees}
PTL0/PTL1/PTL2/PTL3	○	Address(Table) alignment and Size(Table) compatible with `TCR_EL1` fields. See [40] for more details.
TCR_EL1	○	TCR_EL1[6, 35, 59...63] = 0

Size(X) = The size of the object X.
Object[W,X,Y...Z] = Bit W,X and all the bits in [Y,Z] of Object.
Object[NAME] = Field 'NAME' of Object. See documentation for the exact location of the field in the object.
RAM = Physical address used to access to a system memory location (no MMIO, ROMs, device memory etc.).
ReadOpcode(X,Y), WriteOpcode(X,Y) = Physical address of an opcode which reads/writes on register X or Y.
ExecOpcode(X) = Physical address of X opcode.
Address(X) = Physical address of the object X.
REGISTER[W,X,Y...Z] = Bits of the value contained in REGISTER.

[1] It exist at least a page writable in kernel mode and not writable in user mode.
[2] It exist at least a page readable or writable in in user mode.

AMD64 (64 bit) 4-level MMU mode		
Object	**Type**	**Rule**
IDT entry	●	Size(IDTEntry) = 16 Byte
IDT entry	●	Address(IDT) % 4 = 0
Empty IDT entry	●	IDTEntry[P] = 0
Used IDT entry	●	IDTEntry[P] = 1 ∧ IDTEntry[35...39,44,95:127] = 0 ∧ IDTEntry[TYPE] ∈ {14,15} ∧ IDTEntry[DPL] ∈ {0,3}
IDT table	●	Address(IDT) % 8 = 0 ∧ Size(IDT) = 4096 Byte
IDT table	●	∀ i ∈ {0...8,10...14,16...19} IDT[i][P] = 1 ∧ IDT[i][47...64] = 1
IDT table	●[1]	IDT[3][DPL] = 3 ∧ IDT[0,2,6,7,8,10...14][DPL] = 0
PT/PD/PDPT/PML4 Entry	○	Size(Entry) = 8 Bytes
Empty PT/PD/PDPT/PML4 Entry	○	Entry[P] = 0
PT entry	○	Entry[P] = 1 ∧ Entry[MAXPHYADDR:51] = 0
PD 2MiB entry	○	Entry[P] = 1 ∧ Entry[MAXPHYADDR:51] = 0 ∧ Entry[7] = 1
PDPT 1GiB entry	○	Entry[P] = 1 ∧ Entry[MAXPHYADDR:51] = 0 ∧ Entry[7] = 1 ∧ Entry[21:29] = 0
PT/PD/PDPT pointer	○	Entry[P] = 1 ∧ Entry[MAXPHYADDR:51] = 0 ∧ Entry[7] = 0 ∧ Entry[Address] ≪ 12 ∈ RAM
PD/PT/PDPT/PML4	○	Address(Table) % 4096 = 0 ∧ Size(Table) = 4096 Bytes
Radix Tree	●	∀ InterruptHandler ∈ {Found IDT} Resolve(RadixTree, InterruptHandler[Address]) = True

IDT[X][Y] = Field Y of the record X in interrupt table.
MAXPHYADDR = See documentation [42] for more details.
Resolve(R,V) = Resolve virtual address V using radix tree R returning True for success False otherwise.

[1] The DPL field of IDT entries defines the minimum CPU privilege mode which is allowed to execute the code pointed by the entry: 0 for kernel mode, 3 for user mode. We force the DPL field of some entries to be 0 because that entries are associated with interrupts which are managed by the kernel (e.g. NMI, invalid opcode, double fault etc.). Instead, we force the code pointed by the entry associated with breakpoint exception to have DPL = 3 because it is needed to permit to a user space process to debug other processes.

ARM (32 bits) Short MMU mode		
Object	**Type**	**Rule**
PTL1/PTL2 Entry	○	Size(Entry) = 4 Bytes
Empty PTL1/PTL2 Entry	○	Entry[0,1] = 0
Small PTL2 Page Entry	○	Entry[1] = 1 ∧ (Entry[TEX] ≠ 1 ∨ Entry[B] = 1 ∨ Entry[C] = 0)
Large PTL2 Page Entry	○	Entry[0] = 1 ∧ Entry[1] = 0 ∧ Entry[6...8] = 0 ∧ (Entry[TEX] ≠ 1 ∨ Entry[B] = 1 ∨ Entry[C] = 0)
Large PTL2 page entries group	○	if ∃ LPEntry in Table ⇒ ∃ $LPEntry_j$ | $Address(LPEntry_{j+1})$ = $Address(LPEntry_j + 4)$ ∧ $Address(LPEntry_0)$ % 16 = 0) j ∈ {0...15}
Section PTL1 Entry	○	Entry[1] = 1 ∧ Entry[9,18] = 0 ∧ (Entry[TEX] ≠ 1 ∨ Entry[B] = 1 ∨ Entry[C] = 0)
Supersection PTL1 Entry	○	Entry[1,18] = 1 ∧ Entry[9] = 0 ∧ (Entry[TEX] ≠ 1 ∨ Entry[B] = 1 ∨ Entry[C] = 0)
Supersection PTL1 entries group	○	if ∃ SSEntry in Table ⇒ ∃ $SSEntry_j$ | $Address(SSEntry_{j+1})$ = $Address(SSEntry_j + 4)$ ∧ $Address(SSEntry_0)$ % 16 = 0) j ∈ {0...15}
PTL2 pointer Entry	○	Entry[0] = 1 ∧ Entry[1] = 0 ∧ Entry[9] = 0
PTL1 Table	○	Address(Table) alignment and Size(Table) compatible with `TTBCR` fields. See [42] for more details.
PTL2 Table	○	Address(Table) % 1024 = 0 ∧ Size(Table) = 1024 Bytes
Kernel Radix tree	●	∃ DataPage | ReadOpcodes(`DFSR, IFSR`) ∈ DataPage ∧ ∃ DataPage | WriteOpcodes(`TTBR0, TTBCR`) ∈ DataPage
Kernel Radix tree	●[1]	∃ Entry ∈ {Tables of RadixT} | Entry[PNX] = 0 ⇒ RadixT ∈ {Accepted Kernel radix trees}
User Radix tree	●[1]	∃ Entry ∈ {Tables of RadixT} | Entry[NX] = 0 ⇒ RadixT ∈ {Accepted User radix trees}
User Radix tree	●[2]	∃ Entry ∈ {Tables of RadixT} | Entry[AP[1]] = 1 ⇒ RadixT ∈ {Accepted User radix trees}
TTBCR	○	TTBCR[3] = 1 ∧ TTBCR[6, 31] = 0

[1] It exist at least a table entry executable i.e. it exist at least a page which contains code.
[2] It exist at least a table entry writable i.e. it exist at least a page which contains data.

MIPS32 (32 bit) TLB and Radix tree MMU modes		
Object	**Type**	**Rule**
Config	○	Config[4...6] = 0 ∧ Config[31] = 1
Config5	○	Config5[1,12,14...26] = 0
PageGrain	○	PageGrain[5...7,13...25] = 0
PageMask	○	PageMask[0...10,29...31] = 0
PWCtl	○	PWCtl[8...30] = 0
PWField	○	PWField[30...31] = 0 ∧ (if Config[10...12] ≥ 2 ⇒ ∀ i ∈ {GDI, UDI, MDI, PTI}PWField[i] < 12)
PWSize	○	PWSize[30...31] = 0 ∧ (if Config[10...12] ≥ 2 ⇒ PWSize[PTW] ≠ 1)
Wired	○	Wired[0...15] < Wired[16...31]

PowerPC (32 bit) SDR1+BAT MMU mode		
Object	**Type**	**Rule**
Hash Table Entry	○	Size(Entry) = 8 Bytes
Empty Hash Table entry	○	Entry[V] = 0
Used Hash Table entry	○	Entry[V] = 1 ∧ Entry[RPN] ≪ 12 ∈ RAM ∧ (Entry[R] = 1 ∨ Entry[C] = 0)
Hash Table entries group	○	HT[i][j] ≠ HT[i][k] i ∈ {0. . .Size(HT)/64 - 1} j,k ∈ {0. . .7} ∧ j ≠ k
Hash Table	○	Address(HT) % 16 = 0 ∧ Size(HT) ∈ {64Kib 128Kib 256KiB 512KiB 1MiB 2MiB 4Mib 8MiB 16MiB 32MiB}
Hash Table	○	∀ Entry ∈ HT | Entry[V] = 1 ⟹ Address(Entry)[16. . .31] = Address(HT)[25. . .31] ‖ Hash(Entry[VSID][10. . .18] ⊕ Entry[API]) ∧ (1 ≪ (Log(Size(HT) - 16) -1) ∨ (Address(HT) ≫ 16)[0. . .8]
Hash Table	●[1]	∃ Entry ∈ HT | Entry[V] = 1
Hash Table	●[2]	∃ Entry_1 and Entry_2 ∈ HT | Entry_1[V] = Entry_2[V] = 1 ∧ Entry_1[VSID] ≠ Entry_2[VSID]
Hash Table	●[3]	∃ Entry_1 and Entry_2 ∈ HT Entry_1[V] = Entry_2[V] = 1 ∧ Entry_1[RPN] ≠ Entry_2[RPN]
Parsed Hash Tables	●[4]	∀ j ∈ {Parsed HTs}|$H_j = H(\text{HT}_j) = -\sum P(\text{RPN}_i)\log_2 P(\text{RPN}_i)$; $H_{\max}$ = Max(H_j); if $H_j > 0.8H_{\max} \Rightarrow$ j ∈ {Accepted HTs}

HT[X][Y] = Field Y of the record X in hash table.

[1] It exists at least one not empty entry in the table.

[2] It exists entries associated with at least two different segments.

[3] The hash table maps for at least two different physical pages.

[4] The kernel, spreading data among a wide range of physical pages in RAM, increase the entropy of the distribution of the physical addresses of the pages in the real hash table. We filter the set of hash tables candidates discarding tables which have entropy less than the 80% of the maximum of entropy in the set.

RISC-V SV32 and SV48 MMU modes		
Object	**Type**	**Rule**
PTL0/PTL1 Entry (SV32)	○	Size(Entry) = 4 Bytes
PTL0/PTL1/PTL2 Entry (SV48)	○	Size(Entry) = 8 Bytes
Empty PTL0/PTL1(/PTL2) Entry	○	Entry[V] = 0
PTL0 entry (SV32)	○	Entry[V] = 1
PTL0 entry (SV39)	○	Entry[V] = 1 ∧ Entry[54,63] = 0
PTL1 4MiB entry (SV32)	○	Entry[V] = 1 ∧ Entry[10:19] = 0
PTL1 2MiB entry (SV39)	○	Entry[V] = 1 ∧ Entry[54,63] = 0 ∧ Entry[10:18] = 0
PTL2 1GiB entry (SV39)	○	Entry[V] = 1 ∧ Entry[54,63] = 0 ∧ Entry[10:28] = 0
PTL0 pointer (SV32)	○	Entry[V] = 1 ∧ Entry[R,W,X,D,A,U] = 0 ∧ Entry[Address] ≪ 12 ∈ RAM
PTL0/PTL1 pointer (SV39)	○	Entry[V] = 1 ∧ Entry[R,W,X,D,A,U] = 0 ∧ Entry[54,63] = 0 ∧ Entry[Address] ≪ 12 ∈ RAM
PTL0/PTL1(/PTL2)	○	Address(Table) % 4096 = 0 ∧ Size(Table) = 4096 Bytes

Intel x86 IA32 and PAE MMU modes		
Object	**Type**	**Rule**
IDT entry	●	Size(IDTEntry) = 8 Bytes
Empty IDT entry	●	IDTEntry[P] = 0
Used IDT entry	●	IDTEntry[P] = 1 ∧ IDTEntry[32:39,44] = 0 ∧ IDTEntry[42] = 1 ∧ IDTEntry[DPL] ∈ {0,3}
IDT	●	Address(IDT) % 4 = 0 ∧ Size(IDT) = 2048 Bytes
IDT	●	IDT[0...8,10...14][P] = 1
IDT	●[1]	IDT[0,2,6...8,10...14][DPL] = 0 ∧ IDT[3][DPL] = 3
PT/PD Entry (IA32)	○	Size(Entry) = 4 Bytes
PT/PD/PDPT Entry (PAE)	○	Size(Entry) = 8 Bytes
Empty PT/PD(/PDPT) Entry	○	Entry[P] = 0
PT entry (IA32)	○	Entry[P] = 1
PT entry (PAE)	○	Entry[P] = 1 ∧ Entry[MAXPHYADDR:62] = 0 ∧ Entry[7] = 0
PD 4MiB entry (IA32)	○	Entry[P] = 1 ∧ Entry[7] = 1 ∧ Entry[MAXPHYADDR...19, 21] = 0
PD 2MiB entry (PAE)	○	Entry[P] = 1 ∧ Entry[MAXPHYADDR:62] = 0 ∧ Entry[7] = 1 ∧ Entry[13:20] = 0
PT pointer (IA32)	○	Entry[P] = 1 ∧ Entry[7] = 0 ∧ Entry[Address] ≪ 12 ∈ RAM
PT pointer (PAE)	○	Entry[P] = 1 ∧ Entry[MAXPHYADDR:62] = 0 ∧ Entry[7] = 0 ∧ newline Entry[Address] ≪ 12 ∈ RAM
PD pointer (PAE)	○	Entry[P] = 1 ∧ Entry[MAXPHYADDR:62] = 0 ∧ Entry[7] = 0 ∧ newline Entry[1,2,5,6,8,63] = 0 ∧ Entry[Address] ≪ 12 ∈ RAM
PD/PT	○	Address(Table) % 4096 = 0 ∧ Size(Table) = 4096 Bytes
PDPT (PAE)	○	Address(Table) % 32 = 0 ∧ Size(Table) = 32 Bytes
Radix Tree	●	∀ InterruptHandler ∈ {Found IDT} Resolve(RadixTree, Address(InterruptHandler))

[1] See Note 1 of AMD64 architecture.