Lossless Instruction-to-Object Memory Tracing in the Linux Kernel

Nick Roessler
University of Pennsylvania
nroess@seas.upenn.edu

Yi Chien
Rice University
yc120@rice.edu

Lucas Atayde
Rice University
lsa4@rice.edu

Peiru Yang
Rice University
ypr17@mails.tsinghua.edu.cn

Imani Palmer
Null Hat Security
inp2@protonmail.com

Lily Gray
Rice University
lmg11@rice.edu

Nathan Dautenhahn
Rice University
ndd@rice.edu

Abstract

The lack of visibility into Linux's behavior makes it hard to refactor and maintain. To peer *inside the box*, we present Memorizer, a self-contained, low-level tracing framework that tracks (most) object allocations, data accesses, and function calls within the kernel. The core insight is a low-level object-centric representation that records detailed lifetime information while linking each operation (call/read/write) with its intended target. We evaluate Memorizer using extensive input programs and demonstrate its value by showing how Memorizer can (1) aid in refactoring, (2) extend code coverage with *object coverage* to improve testing and analysis, and (3) identify leaky abstractions. We also release a large data set, visualization tools, and Memorizer's source. This generic, object-centric approach is the first to provide lossless instruction-to-object tracing, adding an essential software engineering capability to the overly complex Linux kernel.

ACM Reference Format:
Nick Roessler, Yi Chien, Lucas Atayde, Peiru Yang, Imani Palmer, Lily Gray, and Nathan Dautenhahn. 2021. : **Lossless Instruction-to-Object Memory Tracing in the Linux Kernel**. In *The 14th ACM International Systems and Storage Conference (SYSTOR '21), June 14–16, 2021, Haifa, Israel.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3456727.3463767

1 Introduction

An abstraction is "leaky" when it fails to completely shield a user from the need to understand its implementation [32]. Leaky abstractions can be particularly problematic when software circumvents encapsulation by directly manipulating objects—a pervasive problem in the Linux kernel [36, 38]. These leaks can lead to security vulnerabilities [33] and poor interoperability. While leaky abstractions are easy to describe, they are difficult to identify and quantify. As a result, leaky abstractions are often only discovered when an abstraction has failed, or when it's time to refactor.

Leaky abstractions hint at a broader set of problems developers face when updating the kernel. In Linux's large multi-module environment, designer *intents for encapsulation* are often difficult to communicate and enforce. Communication difficulties arise not only from the large volume of knowledge required to understand it, but also the lack of tooling to observe its concrete inner-workings. These issues lead to portions of the kernel that are incredibly difficult to refactor without breaking prior code's incorrect usage of internal interfaces. The issue is further complicated by a kernel developer's unrestrained access to the entire address space, with few mechanisms in place to check for inconsistent or improper use of subsystems. Together, these issues result in a kernel that is often difficult to understand, refactor, and maintain.

Despite these challenges, developers are not without help. There are over 150 tools for performance observability, benchmarking, tuning [11, 12], debugging, vulnerability discovery [10, 16, 17, 40, 42], and automated refactoring [6]. Yet none of these tools expose a *global* picture of the kernel's dynamic behavior. Debuggers and tools like eBPF [37] enable manually driven analysis of the kernel, but narrowly focus on a specific context. Other approaches, like memory sanitization and allocation analysis, provide global coverage but only support a few specialized analysis and miss entire object classes (*e.g.,* early boot memory). These tools fail to expose detailed information about the lifetime interactions between objects, leading developers to a slow, iterative style involving reading about a kernel component, modifying it, and then waiting for bugs to arrive to fully understand its footprint.

The objective of this work is to *open the box* and expose an accessible and detailed view of the Linux kernel's memory access patterns to enable new debugging, refactoring and analysis methodologies. Our key hypothesis is that fine-grained object lifetime tracing—*i.e.,* tracing object creation, destruction, and access patterns—can provide a high fidelity representation of the underlying concrete system while being implementable in practice. To that end we present Memorizer, which instruments all allocations, frees, memory accesses, and function calls to build a fine-grained picture of dynamic Linux behavior. To do this we develop an in-kernel component that automatically builds a live object tree and stores access information in shadow objects. The core insight is a low-level, universal representation that allows for simple instrumentation and dynamic processing.

Building Memorizer required solving several challenges. First, tracing all kernel allocators is complex: prior work [16, 17, 42] provides a foundation but neglects early boot objects, special virtual memory mapped regions, and stack frame objects. The complexity was so high that we developed a methodology, *unidentified foreign objects (UFOs)*, to trace allocators we failed to find during early development. Second, losslessly recording lifetime object interactions and storing it for analysis—instruction-level access—stresses space and time, introducing the trade-off of storing just enough information for meaningful analysis. Third, though prior work has built out-of-VM based monitors [4, 5], Memorizer aims to be maintainable, preferably by the Linux community; as such, we built it inside of Linux. This led to several issues which demanded custom concurrency and memory management solutions. Last, efficiently extracting meaningful data is hard; Memorizer's approach is to trace low-level events at runtime and constructing a C-level view in post-processing.

To demonstrate the value of the tool and collected data, we build on and extend three common analysis tasks. First, we show how Memorizer can streamline refactoring projects by automatically exposing data abstractions with its access graph. We validate our dataset and methodology by comparing Memorizer's results to a recent security refactoring. Second, we show how Memorizer's access graph can be used to analyze leaky abstractions. Last, we show how Memorizer can extend traditional code coverage metrics with *object coverage* to open up a new dimension of kernel test case development and analysis. Overall, these examples only scratch the surface of what is possible with detailed object accesses and lifetime information. Our core contributions include:

- The design and implementation of Memorizer, which provides the first lossless object lifetime and access tracing in the Linux kernel (Section 3).
- A large data set for analyzing Linux (Section 5).
- A demonstration of Memorizer's utility through three use cases: streamlining kernel refactoring and maintenance, leaky abstractions analysis, and extending code coverage with *object coverage* (Section 6).
- Memorizer's source code, visualizations, and statistics from kernel tracing available at https://fierce-lab.gitlab.io/memorizer.

2 Background and Related Work

Linux has a plethora of tools to help with profiling, debugging, and analysis. `ftrace` [21] provides significant transparency into the kernel's runtime behavior by tracing the kernel's dynamic control flow; this feature has been invaluable to the Linux community for enabling efficient debugging and allowing users to profile their operating system's behavior to both diagnose and optimize performance. BPF [11] and eBPF [37] are useful for performance profiling and performing specialized analyses by attaching programs to tracepoints and monitoring kernel events in a running system. While these tools are powerful, Memorizer traces all memory operations and integrates with all of the kernel's allocation systems for more fine-grained and complete tracing than eBPF is capable of. `strace` [41] profiles system calls to provide transparency into the interactions between applications and the kernel. Other tools provide similar functionality for querying, tracing, and logging specific subsystems [12].

Other tools try to profile and track *memory*. Kmemleak [28] tracks dynamic heap objects and can be used to diagnose memory leaks inside the kernel. Kernel Address Sanitizer (KASAN) [16] has been used to detect and diagnose memory errors; KASAN works by maintaining a shadow space of valid allocations and checking runtime accesses against the shadow space. Valgrind [31, 39], Pin [25], DynamoRio [1], and DIOTA [26] monitor and trace memory access. However, these tools are primarily designed for userspace applications and focus on detecting errors or providing specialized analysis, while Memorizer's focus is on complete memory tracing for the kernel. Some work has also been done on extracting the structure of Linux using static analysis [22]. However, static analysis suffers from imprecision and does not expose actual dynamic runtime counts for analysis. Table 1 summarizes core differences between Memorizer and related efforts.

3 Memorizer Design and Implementation

Memorizer's goal is to trace all dynamic memory accesses to kernel objects as well as control operations. To do so, it maintains a shadow object for each runtime object and records memory dependencies through read/write monitoring and control dependencies through call monitoring. Tracing kernel operations at this granularity is challenging because of the number of operations, the complexity of kernel's allocators, and the concurrent nature of the kernel.

3.1 Design Principles

Complete Allocator Coverage: Associate *every* memory access with its target object, statically or dynamically allocated.

Lossless: Existing tools commonly use ring buffers which lose data. Memorizer seeks to log every access.

Tool	Capability	Object Tracing	Allocator Coverage	Call Tracing	Use Cases
ftrace [21]	Function tracing	-	-	Yes	Profiling, performance and behavioral analysis
eBPF [37]	Attach monitor	Alloc	Slab, Page, Stack	Probe	Tracing, packet filtering, dynamic programmability
kmemtrace [30]	Slab statistics tracing	Alloc	Slab	No	Slab and memory analysis
KASAN [16]	Memory error detection	Alloc + access	Slab, Page, Stack, Globals	No	Memory error detection
Memorizer	Access/call tracing	Alloc + access	Slab, Page, Stack, Globals, Memblock	Yes, log call	Analyze access patterns, object coverage, refactoring

Table 1: A summary of Memorizer's features compared to other tracing and analysis tools.

Reusable: Many analysis tools provide fixed-functionality, instead enable flexible post-processing for analysis.

Maintainable: Build directly into the Linux kernel with minimal hooks into other subsystems.

Meaningful: Minimize complexity while providing meaningful data: use virtual addresses and map to language level.

3.2 Overview

Memorizer's system architecture is shown in Figure 1. To capture events, Memorizer (1) instruments the Linux source code to hook object allocations, and (2) adds compile-time instrumentation for every read, write, and call. When an object is allocated, Memorizer allocates a new shadow object for it and installs it into the live object map. When a memory access occurs, Memorizer locates the associated shadow object and updates its access counts. Similarly, Memorizer tracks function calls by updating its accounting on each call. When an object is free'd, Memorizer removes it from the object map and places the shadow object into a queue awaiting serialization. Data is serialized through a debugfs interface for easy collection. Memorizer is controlled through the debugfs interface, *i.e.,* enabling and disabling logging and configuring options. Memorizer includes a stats module for analyzing events, such as the number of allocations (broken down by allocator), accesses, calls, and internal memory usage. Collected logs are mapped to higher-level semantics and visualized using a suite of graphing and analysis tools.

3.3 Object and Memory Management

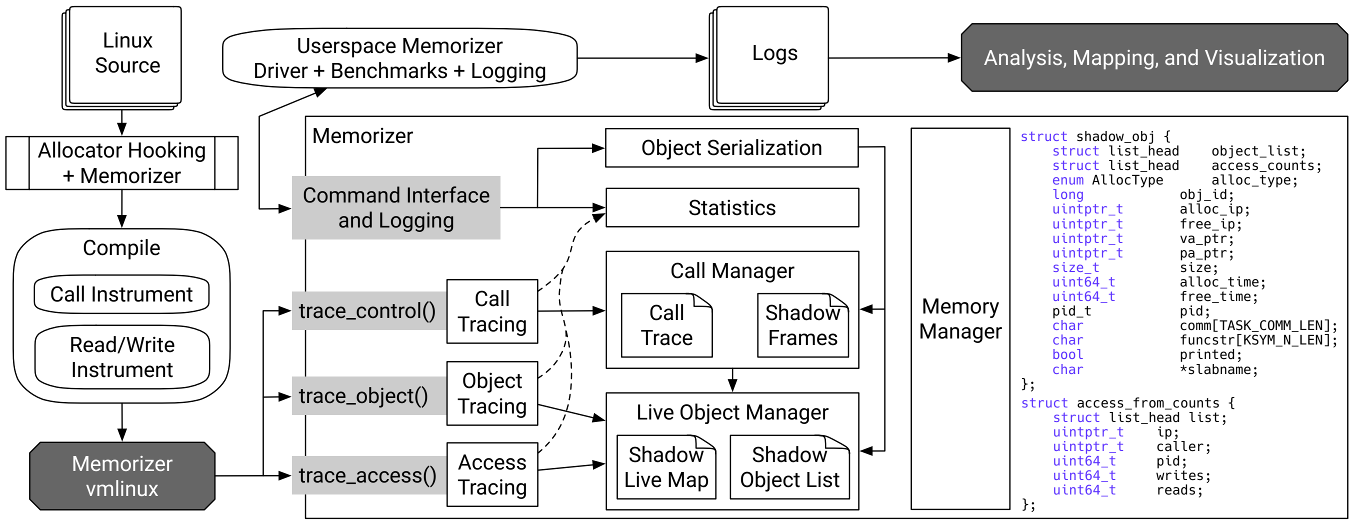
The heart and soul of Memorizer is the object management and tracing system, which maintains a *shadow object* for each kernel object as well as the live object map that associates each address with its shadow object.

3.3.1 Universal Representation The two design principles of *lossless* and *meaningful* contend with each other: the more information stored the harder it is to remain lossless. We have found the standard object model [18, 32] to be an informative way to analyze systems. That is, some *subject* performs *operations* on some *object*. Thus, our low-level representation traces events as *(subject, operation, object)* triples. The core data structure for object-centric tracing is `struct shadow_obj`, as shown in Figure 1. A key feature is that all objects, regardless of allocator, have the same shape. This opaque representation simplifies tracing and analysis, and can represent objects of diverse types, from individual fields to large block regions.

Memorizer traces read, write, call, return, alloc and free operations. The subject for each is minimally the virtual address that invoked the operation. The subject for allocation is richer, *i.e.,* both multidimensional and extensible, as we not only track the allocation site but also other valuable contextual data, such as the size, alloc/free time, current process, and slabname if available. Rather than keeping a log of each access, which requires too much memory, Memorizer compresses all accesses from the same program address into a single structure that sums all accesses, `struct access_from_counts`. Access contexts could be extended, but each dimension generates more list items, which overruns available memory.

3.3.2 Representing Time We originally attempted to use Linux `jiffies` [7, 29] but they demonstrated atomicity discrepancies, so we settled on using our own logical time. While logical time does not capture global system time, it does make analysis for data structure lifetimes and critical optimizations [29, 44] possible. An alternative option is to use a hardware clock (*e.g.,* TSC).

3.3.3 Live Object Map: Lookup Table Memorizer maps each byte of virtual address space to a shadow object using a three-level hash table, where the key is the virtual address of the live object—analogous to software page tables. The internal interface is: `init`, `alloc`, `lookup`, and `free`. `init` performs initial allocation of the lookup tables. `alloc` takes the virtual address and size of an object and updates each leaf entry with the address of the shadow object. `free` zeros out the entries for the free'd object region and sets both the free time and freeing code pointer in the shadow object. `Lookup` returns the shadow object pointed to by the given virtual address. A key aspect of the map was table sizes, where the address space is represented in three page levels. Poor choices in sizes resulted in L3 thrashing in early prototypes, which we manually tuned. Tables are allocated on demand.



```
struct shadow_obj {
    struct list_head    object_list;
    struct list_head    access_counts;
    enum AllocType      alloc_type;
    long                obj_id;
    uintptr_t           alloc_ip;
    uintptr_t           free_ip;
    uintptr_t           va_ptr;
    uintptr_t           pa_ptr;
    size_t              size;
    uint64_t            alloc_time;
    uint64_t            free_time;
    pid_t               pid;
    char                comm[TASK_COMM_LEN];
    char                funcstr[KSYM_N_LEN];
    bool                printed;
    char                *slabname;
};
struct access_from_counts {
    struct list_head list;
    uintptr_t    ip;
    uintptr_t    caller;
    uint64_t     pid;
    uint64_t     writes;
    uint64_t     reads;
};
```

Figure 1: System Architecture and Flow. Memorizer uses a combination of source code instrumentation and compile-time hooks to capture all object allocation, memory access, and call operations that occur in the kernel. When an object is allocated, Memorizer allocates a shadow object for it; when a memory access occurs, Memorizer locates the appropriate shadow object and updates its access counts. The `structs` show the shadow object and access list, which together track all accesses to each object. A debugfs interface allows for easy serialization and collection of the captured data and can be used to control Memorizer, *i.e.,* enable or disable its various tracers. Lastly, the log files are parsed by a suite of analysis and graphing tools for offline analysis.

3.3.4 Allocator Promotion Linux's allocators are structured in a hierarchy; for example, the slab allocator allocates its memory by requesting large page-aligned memory blocks from the page allocator. When an object is allocated by a higher-level allocator, the containing region is *promoted*; Memorizer only associates each address with the single active object from the highest-level allocation. We describe some complexities related to promotion (such as accesses from constructor functions that run before allocations) in Sect. 3.4.1.

3.3.5 Internal Allocator Using Linux allocators for internal memory created stability issues. Instead, Memorizer maintains its own memory region that is disjoint from the rest of kernel memory; this design is simple and reduces Memorizer's impact on system behavior. We implement a simple bump allocator that serves blocks from this region for Memorizer's allocation requirements, and for SMP we create per core memory pools but have yet to implement a free interface.

3.4 Complete Object Tracing

One of the most complex and time-consuming aspects of Memorizer was identifying, hooking, and labeling Linux allocators and special memory regions. Enabling complete object tracing required significant manual exploration of Linux kernel memory management subsystems—we even used Memorizer itself to learn about and expose allocators.

[1]Early boot (memblock) memory permits overlapping regions, which challenges Memorizer's object tracking. Memorizer currently ignores frees and considers all memblock allocations as a single object.

3.4.1 Hooking Linux Allocators This section details the kernel's allocators and how Memorizer hooks them—coincidentally, it provides an introductory overview of various Linux allocators and memory regions. Memorizer exports a specialized interface for each such allocator so that it can track type information about them. In this way, Memorizer tracks general information about each allocation which is a core component of tracing the access graph.

Globals and per-CPU variables Memorizer allocates shadow objects for static data, including the global variables from the `.data`, `.rodata`, and `.bss` sections, as well as per-CPU variables, on system initialization. It hooks the KASAN global variable initialization logic to process these objects, assigning each to a single fixed range of virtual memory based on the object's location and size. Static data are never freed, thus there are no deallocation operations.

The Page Allocator The page allocator [2] is a low-level, page-granular dynamic allocation subsystem in the kernel. It is used by the kernel in contexts in which large, contiguous, page-aligned regions of memory are appropriate. We hook each allocation interface to the page allocator, which includes `__alloc_pages_nodemask`

	Allocator	Allocation Interfaces	Deallocation Interfaces
Dynamic	Page Allocator	`__alloc_pages_nodemask`, `get_free_pages`, `alloc_pages_exact`, `get_zeroed_page`	`free_page_prepare`
	Slab Allocator	`kmalloc`, `kmem_cache_alloc`	`kfree`, `kmem_cache_free`
	Vmalloc	`vmalloc`	`vfree`
	Memblock (early boot)	`memblock_insert_region`	`memblock_remove_region`^{1}
	Stack Memory	Prologue instrumention	Epilogue instrumentation

	Region	Allocation and Description
Static	Globals	Each global and per-CPU variable is hooked from `kasan_register_global` during system initialization
	`FIXADDR`	Addresses in range (`FIXADDR_START`, `FIXADDR_START + FIXADDR_SIZE`) are treated as a single object
	`VMEMMAP`	Addresses in range (`VMEMMAP_BASE, VMEMMAP_END`) are treated as a single object

Table 2: Overview of hooked allocators, including both dynamic memory (top) and static memory (bottom).

and `__get_free_pages`. Additionally, there are specialized interfaces such as `alloc_pages_exact` and `get_zeroed_page` that internally call `__get_free_pages`. Memorizer hooks these as well. For deallocations, we hook `free_page_prepare`, a function called by all the free routines, to capture page freeing operations.

The Slab Allocator While the page allocator provides a primitive for dynamic allocations, its page-granularity makes it unsuitable for the (likely common) cases in which objects are *smaller* than a page, and would thus lead to excessive internal fragmentation. As a result, the kernel is equipped with a general-purpose slab allocator [3], built on top of the page allocator, for handling most dynamic memory use cases. Memorizer supports SLUB, the default slab allocator in most modern kernel distributions.

The Slab allocator provides a general-purpose interface, `kmalloc`, which behaves similarly to the userspace `malloc`; Memorizer hooks this interface, in addition to its `kfree` analog for deallocation. In addition, for cases in which many objects of the same type are expected to be allocated, the slab allocator allows for the creation of entire *slabs* of objects of the same type. A fresh, free object from a particular slab (sometimes called a cache) can be requested with `kmem_cache_alloc`, then subsequently released with `kmem_cache_free`; Memorizer also hooks these interfaces.

Slab allocation is complex in that whenever it initializes or extends the cache it allocates a large region to hold many objects, effectively preallocating them. Moreover, caches with a constructor may modify these objects in bulk before they are used to satisfy allocations, meaning they can get accesses before their real use. Consequently, Memorizer allocates shadow objects for individual cache objects as they are created and initialized; this way accesses from the constructors can be properly attributed to each of the objects. Subsequently, when a cache object is used to satisfy a kernel allocation request, the allocation address in the shadow object is updated to the location of the call that performed the request, and no fresh shadow object is created. This design means cache objects have accesses from both (1) their constructors, and (2) the accesses that result from their use by the kernel.

The `vmalloc` Interface Memorizer hooks the `vmalloc` / `vfree` interface, another dynamic memory allocation system built on top of the page allocator. Its primary use is for creating large, virtually contiguous regions of memory, such as creating space for loading dynamic kernel modules.

The Memblock Subsystem The `memblock` [23] system is a primitive memory subsystem in the kernel, primarily used during boot before the other allocation systems have been initialized. Memorizer currently treats this type of memory in a coarse-grained fashion, with all allocations sharing a single shadow object. Memorizer hooks the subsystem from `memblock_insert_region`, a function that is called by its other interfaces.

Special Memory Regions In addition to the range of static and dynamic objects in the kernel, there are several special regions of the memory that Memorizer handles. One is for `fixed-map` [24] memory, which contains pages whose physical addresses do not depend on `__START_KERNEL_map`. Fixed-map memory is used for several low-level purposes, such as storing the Interrupt Descriptor Table (IDT) and during boot inside `early_ioremap_setup`, for example. The kernel also uses a sparse virtual memory map for fast address translation, which Memorizer treats as a single object.

Stack Stack tracing, next to concurrency and managing the complete set of allocators, is the most technically challenging element of Memorizer because stack objects are implicitly allocated by the compiler, which we wanted to avoid modifying. Memorizer provides two modes for tracking stack objects. The first is a coarse-grained approach that minimally attributes all accesses to the stack data type. It works by allocating a single stack shadow object, and then at the time of a fork, it maps the entire allocated stack (*e.g.,* two-page region) to the stack shadow object.

Additionally, Memorizer provides a frame-level tracing option for stack data. In this mode, the kernel is recompiled and GCC's `-finstrument-function` is used to insert tracing code in each prologue and epilogue. This instrumentation uses the `RBP` and `RSP` registers to identify the virtual address range of the active stack frame. Memorizer then creates a single shadow object for each caller-callee edge (as opposed to one per function), which makes the frame tracing context-sensitive. On each call, Memorizer updates the lookup table for the allocated frame, which is a costly operation and one reason for the slowdown incurred by this mode.

While this approach works for the majority of functions, it is possible for the compiler to allocate locals after the prologue, causing them to be allocated outside of the traced frame. For this reason, we enable the coarse-grained tracing as well to account for those accesses. In future work, one might either modify the compiler or use debugging metadata to compute the frame's size to better handle this situation.

Lastly, we must free the allocated frame. Because frames are frequently created and destroyed, updating the lookup table on each epilogue would be slow. Instead, Memorizer ignores deallocating on returns and relies on allocation promotion to ensure that the active object is always the correct one. The major drawback of this approach is that if a call edge is missed, then accesses would be attributed to the most recent frame in that location. We find that GCC's `-finstrument-function` doesn't instrument all the function calls in the kernel; in post-processing, accesses from those that it misses can be either be removed or attributed to the generic stack data type.

3.5 Access Tracing

Memorizer traces memory accesses by hooking KASAN's access tracing. KASAN uses a GCC compiler pass that instruments reads and writes as well as functions like `memcpy`. We modify KASAN with calls into Memorizer, which requires a handful of hook points for various operations that pass the address of the accessing instruction and size of the access. Access counts are stored in the `struct access_from_counts` (Figure 1) within a linked list pointed to by the shadow object. On an access, Memorizer (1) queries the live object map to get the shadow object, (2) searches the list for the access site, and (3) updates the counts (and allocates and inserts it if missing). For performance reasons, Memorizer tracks only the number of accesses and does not include their temporal sequencing.

3.5.1 Missing Shadow Objects: UFOs Memorizer aims for complete coverage, but in early prototypes, many accesses did not have an associated shadow object—the allocation was missed. To aid in debugging and still produce complete tracing, we created a class of objects called Unidentified Foreign Objects (UFOs), which are implicitly created on an access to an address on a memory page for which Memorizer does not have an allocated Shadow Object. The UFO implicit allocator does two things. First, it interprets KASAN's labels to infer the type of object being accessed: Heap, Stack, Global, etc., which can indicate valuable type information for missed allocations. Second, it creates and adds a page-sized shadow object for the UFO in the live map. The number of UFOs diminished as we used this methodology to identify classes of allocators we missed. For example, the UFOs and their analysis helped us identify early boot allocators. Although the number of UFOs has now reached zero in our implementation, the UFO allocator is still valuable in that it can aid in the maintainability of Memorizer by automatically exposing any new/custom allocators.

3.6 Call Tracing

Memorizer traces calls by using GCC's function instrumentation (`-finstrument-function`) that inserts a call to `__cyg_profile_func_{enter,exit}` at every function entry and exit. Memorizer traces call operations as (caller,callee,shadow_frame,count) tuples and uses a bucket hash table using the caller virtual address as the key. Memorizer does not trace return edges but could easily be extended to do so. The first Memorizer prototype used `ftrace` for call tracing, but we found it lacked complete function coverage.

3.7 Concurrency Control

One of the most unique aspects of Memorizer is that it monitors three separate but highly intertwined operations (call, access, and alloc), while executing in every low-level context. As such, we must protect from reentrance throughout interleaved tracing contexts, interrupts, shadow object writes, and lookup table updates. By design, Memorizer has only a few global objects requiring protection. To prevent reentrance—and thus preventing infinite loops—we use a monitor pattern allowing only one entrance at a time. Memorizer disables interrupts while operating on critical sections: currently, these sections are coarse and can likely be optimized. This design will not trace NMI handlers as they cannot be masked. This means that if an NMI is delivered while Memorizer is running, Memorizer may lose a small number of accesses. NMIs are rare *e.g.,* for non-recoverable errors. Our monitor uses atomic instructions to avoid relying on Linux functions, which would instantly cause reentrance. Memorizer protects shadow objects with a reader-writer lock since multiple cores could modify the same object close in time. The lookup tables are shared, however, Memorizer uses a lockless algorithm because the kernel should have proper concurrency control on object allocation and free.

3.8 Control and Serialization

Memorizer is controlled through a debugfs interface. Key features include the ability to enable/disable logging of the different events (access,call,alloc), show stats, clean up runtime, and export the access-graph. Each of these interfaces is quite simple and self-explanatory. The more complex interface is serialization, which is provided via the Linux `seq_file` abstraction [20] because it allows for fine-grained locking and internal control. The major challenge with serializing Memorizer's data for output is how often it should be performed. Writing out the data can take a long time, and the system cannot trace while the serialization is being performed. To address this, we chose to run full test suites and do a single serialization afterward. This puts enormous pressure on the runtime memory and usability. Memorizer also provides boot-based options that can enable/disable tracing and select the amount of memory for the internal allocator.

4 Experimental Methodology

One aim of our work is to produce a large, meaningful data set and analysis. However, since Memorizer is based on dynamic analysis, we must be careful what claims can be made. We address this problem in a few ways. First, we introduce the *access coverage* metric for measuring coverage as the number of distinct access edges—we explore analysis in Section 6.3. Second, we use access coverage to measure the total observed behavior and the difference per each new experiment. This allows us to know when we've exposed as much as possible from a given input program because the number of new edges stabilizes. Third, we focus our analysis on observable claims as indicated by the data instead of making claims that depend on possibly undiscovered execution. For example, a leaky abstraction is an access that violates an appropriate interface, and can only be claimed by observing evidence in the graph. We cannot claim that leaky abstractions do not exist.

Collection Methodology We collect data using Qemu to boot a Memorizer 4.10.0 kernel while running inside Ubuntu 16.04. We modify the default Ubuntu kernel configuration to: use a single core, disable preemption, and disable KASLR so that addresses match the `vmlinux` and are consistent across runs for analysis. We reserve enough memory so that Memorizer can trace a full test before serializing, which is typically between 16 and 50 GB. Incremental serialization is possible but slows down progress significantly and can miss certain behavior (interrupts). To drive the kernel, we run the Linux Test Project (LTP) (26 test suites), as well as the `system` and `kernel` Phoronix benchmarks (28 benchmarks). Each test is run on a fresh Qemu instance; at the end of execution, the access graph is written to disk for offline analysis. All tests are run twenty times to improve and analyze coverage.

The first run of LTP adds approximately 300k unique edges into the access graph. The second pass adds around one thousand, and by the 15th trial only around 30 or so unique edges are added for each pass of LTP (0.01% increase per complete pass). This leads us to conclude that there is a small degree of non-determinism but that coverage largely stabilizes. When adding Phoronix data to the access graph derived from LTP, we observe 1000 edges on the first first pass of the Phoronix benchmarks, and between 0 to 10 unique edges on subsequent passes (with 5 passes adding a total of 0), for a total of 1,200 additional edges. This indicates that coverage is stable and that our method exposes most of the accesses that can be observed from LTP and Phoronix benchmarks.

Data Details and Limitations Data is stored as an adjacency matrix in ASCII containing the full count of all logged interactions. Raw trace files are 2.0GB on average. We produce compressed forms of these files by combining all dynamic allocations from the same allocation site into a single object, which speeds up our analysis for the cases in which we treat objects as types; compressed trace files are 5.9MB on average. We trace all allocations during boot, but do not trace accesses or calls; access and call tracing is enabled immediately before running a trial and disabled afterward.

5 Performance Evaluation

We evaluate four configurations: baseline Linux, Linux with KASAN enabled, Linux with KASAN + Memorizer, and finally Linux with KASAN + Memorizer + frame tracing. All experiments are done using KVM with a single core (except for the SMP evaluation), 64GB memory (40GB memory is assigned to Memorizer if enabled). The bare metal machine is equipped with an Intel(R) Xeon(R) Gold 6126 CPU @ 2.60GHz, 188G memory, and 3.5T storage.

Microbenchmarks: We run the LMBench [27] latency and bandwidth microbenchmarks on our various kernel configurations and show the results in Table 3. Because Memorizer logs every call and memory access, the latency overheads are substantial for some metrics, *e.g.,* the latency of a `execve` (`exec proc`) is about 300X compared to the baseline kernel configuration. For other microbenchmark metrics such as bandwidth, the cost is less severe (*e.g.,* `Bcopy` is around 72%). In general, the overhead introduced by Memorizer is in direct proportion to the number of allocation, free, call, return, and memory access operations that are performed. We find that both call tracing and memory access tracing contribute to overhead in about even proportion.

Microbenchmarks for SMP LMBenchmark results with 1, 2, 4, and 8 cores are shown in Figure 2. Memorizer scales as anticipated with core count on most metrics, but some (*e.g.,* `exec proc`) degrade substantially. Since, Memorizer uses a read-write spinlock for shadow object access this

LMBench Processes Times Overhead for Different Configurations

	null call	null I/O	stat	open/clos	slct TCP	sig inst	sig hndl	fork proc	exec proc	sh proc
KASAN	1.3	2.2	4.6	4.2	4.1	2.1	7.7	7.7	8	4.5
Memorizer	23	71	253	141	1824	73	14	263	296	177
Stack trace	5464	5997	8015	5778	44041	9200	1155	4802	5525	3068

LMBench Local Communication Bandwidth (MB/s) for Different Configurations

	Pipe	AF UNIX	TCP	File reread	Mmap reread	Bcopy (libc)	Bcopy (hand)	Mem read	Mem write
Linux	9206	13000	7326	11900	25900	13300	10400	18K	12.6K
KASAN	4021	2054	1808	7710	25000	13200	9892	17K	11.9K
Memorizer	22	17	11	226	17300	9320	7584	13K	10.6K
Stack Trace	0.2	0.3	0.2	0.8	9218	5801	5184	9608	7188

Table 3: LMBench Evaluation

	Linux	KASAN	Memorizer	Stack Trace
encode-mp3 (secs)	10.8	10.9	19.8	42.3
openssl (signs/sec)	260.3	260.8	217.9	29.8
compress-7zip (MIPS)	4143	4230	1050	528
pybench (msecs)	1472	1479	2070	8293
ffmpeg (secs)	25.1	25.2	61.7	103.7
gnupg (secs)	14	15	112	136

Table 4: Phoronix Benchmark Evaluation

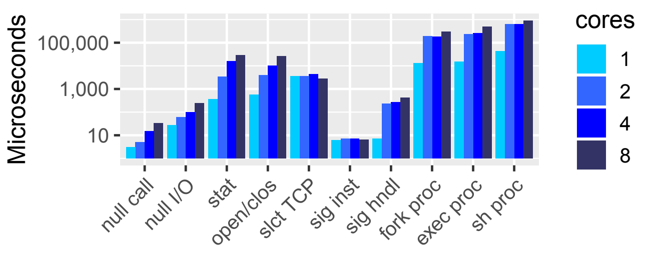


Figure 2: LMBench SMP Evaluation

trend is expected as contention increases. Some metrics *e.g.,* `slct TCP` are largely independent of core count.

Macrobenchmarks In Figure 4 we show the results of the various kernel configurations on a range of Phoronix [34] benchmarks. Because these benchmarks spend a large portion of their time in userspace, the overhead is lower than in LMBench. Qualitatively, while some overheads are large (*e.g.,* 3.9X in the case of `compress-7zip`), the system is still quite usable even when tracing; we are able to SSH into the machine and perform tasks without issue.

Runtime Memory Consumption After booting the system (in which allocations are traced but accesses are not), Memorizer uses a total of 5634MB of memory (5287MB for the lookup table and 347MB for the shadow objects). After tracing `lmbench`, Memorizer uses a total of 14237MB of memory (11036MB for the lookup table, 1021MB for the shadow objects, and 2180MB of access edges). For the same workload, the kernel itself uses about 50MB of memory including its code and data. Measuring memory consumption in this way helps us quantify Memorizer's memory pressure and determine how much should be allocated for tracing runs.

6 Uses

In this section, we show how Memorizer can: aid in refactoring, maintaining, or adding new kernel subsystems; be used to analyze and measure the leakiness of kernel abstractions; and extend traditional coverage metrics with *object coverage*.

6.1 Refactoring and Maintenance

Refactoring and maintaining kernel code is a daunting task. Understanding the underlying data abstractions is among the most central of concepts: what are the interfaces to *this* particular object? How well-encapsulated is it? How many other subsystems interact with this object? Currently, a typical developer workflow is *iterative*: a developer reads code, makes a guess about a refactoring, finds what they missed, and repeats the process. Such a workflow is haphazard and expensive in human engineering time. In this section, we show how Memorizer can streamline refactoring by providing developers with automatically-generated access graphs to quickly understand kernel data abstractions.

6.1.1 Case Study: Privilege Separation xMP [35] is a recent security work that uses nested paging hardware to enforce in-kernel abstractions at runtime. Although xMP provides memory protection services, *using* it requires significant effort: an engineer selects an object (or group of objects) to protect, defines what code can access it, transforms the code, and instruments kernel operations to switch contexts.

The authors used xMP to isolate `struct cred`, which is used for almost all runtime access control checks. To isolate it, the authors first modified the slab allocator to place `cred_jar` objects into separated nested pages. Then, through reading kernel source and observing access faults, they iteratively identified where write accesses came from and wrapped each location with context switches, `xmp_unprotect`, and `xmp_protect`, for access. In other words, the process involved repeated trial and error, which we have confirmed with the xMP authors.

We contrast this manual, time consuming method with using memorizer, where all allocations are tracked by default and all that is needed is to query the access graph. We first query to find the set of write instructions to `cred` objects. We then compare these with the manually instrumented code locations performed by the xMP authors, and found a nearly perfect match: only 4/35 accesses were missed because the xMP kernel used SELinux while ours used AppArmor. Furthermore, we observe that `cred` objects are written only 9 times on average but are read 236 times—they are checked on every privileged operation but rarely modified—indicating an optimization that allows global read-only access and only changes contexts for the less frequent write operations. These compelling results demonstrate that (1) Memorizer can save large amounts of engineering effort through its object tracing by providing developers with fine-grained object access graphs, and that (2) Memorizer can be used to estimate the difficulty of refactoring by directly quantifying and exposing the locations in the kernel that access a particular object type.

6.1.2 Memorizer to Implement Memorizer Perhaps the biggest testament to the value of Memorizer's approach is in the development of Memorizer itself. None of the authors began this work with deep knowledge of the kernel's allocation systems. The trifecta of object, call, and access tracing allowed us to quickly triangulate aspects of the system we didn't understand in order to complete our implementation.

In particular, Memorizer's unidentified object allocator (UFOs—Sec. 3.5.1) allowed us to quickly *learn which allocators we were missing*: in a system with a vast number of allocators, this approach gave us clear direction in reaching nearly complete object coverage (an improvement on all prior object tracing monitors). By inspecting our data we found easily observable patterns, such as *high-degree* objects, that we used to quickly and automatically discover the hierarchical relationships among allocators: for example, in our first iteration of tracing the page allocator, we hooked only the `alloc_pages` interface; however, `alloc_pages_exact` calls `alloc_pages`: we quickly saw this is a common and closely-located call site, which we then hooked as an allocator. This process allowed us to understand the complex relationships within the allocators.

6.2 Leaky Abstractions Analysis

In this section we introduce the *Object Encapsulation* model and method for measuring abstraction leakiness. In the Object Encapsulation model, each object, o, and instruction, i, is assigned to exactly one *encapsulating partition*, p. An encapsulating partition is a grouping of code and data that is treated as single compsite object for the purpose of analyzing encapsulation boundaries. The *Encapsulation Ratio (ER)* is a new metric that measures the *leakiness* of each object, and is defined as the ratio of access *sites* not in the partition (X external or public sites) to the total access sites (T): X/T. Intuitively, an external access *crosses* the encapsulation boundary, implicitly making the data public. This model can be instantiated with diverse assignments: directory, groups of functions, or even using algorithms based on frequency and types of access.

To make our analysis concrete, we assign each i and o to a p based on the file where the instruction or object allocation site resides. This means that each C file is considered a singleton class. Table 5 presents the number of object allocation sites that have exactly $\{0,1\}$ external read (X_r) and write (X_w) sites, as well as sites with an ER of 1. Our results show a bimodal distribution of ERs, implying that in general, objects are either well encapsulated, *i.e.,* written mostly by the file that allocated the object, or not at all, *i.e.,* all accesses are from outside the allocating file. In general, writes are much better encapsulated than reads. Surprisingly, 284 object sites had no external writes and 53 had exactly one external write, meaning that 52% of the objects were only written by at most one instruction not in the same file as the allocation site. This indicates a high degree of write encapsulation for over half of the observed objects. Interestingly, 91 of the 645 objects (14%) had an ER of 1, indicating that *all* writes were external. We corroborated that these writes were to objects allocated by shared libraries and memory management code, confirming that certain objects are legitimate public kernel data.

	Allocation Sites	% of Total Sites
$X_w = 0$	284	44%
$X_w = 1$	53	8%
$ER_w = 1$	91	14%
$X_r = 0$	173	27%
$X_r = 1$	48	7%
$ER_r = 1$	136	21%

Table 5: File Level Encapsulation Analysis

While this technique exposes the latent connections between components in a general way, it becomes particularly useful if the encapsulating assignments are done by a developer, as they can precisely assign partitions to detect violations. We perform a non-expert variant in the next section, but we envision that developers could make an assignment for their code and then a CI system could automatically report encapsulation analysis of how their abstractions are (ab)used.

6.2.1 Linux Cryptography API We use the encapsulation analysis to investigate Linux's Cryptography API by assigning encapsulating objects based on our manual analysis of the crypto library. We observed that all private keys or certificates are well encapsulated with a write ER of 0, indicating high integrity and that external components do not directly modify them. These properties inspire confidence not only in the engineering of the API but Memorizer's ability to identify code that has strongly defined encapsulation boundaries.

6.3 Testing: Coverage Metric

Code coverage is the traditional way to measure dynamic analysis [9, 13, 14, 43]. However, we believe data coverage is a critical dimension for understanding program behavior—oftentimes it is impossible to tell where faults lie in a program through code coverage alone, as the same code may run on a wide range of data inputs at runtime. We propose Memorizer's access graph for augmenting traditional code coverage with fined-grained object coverage. To handle dynamic objects in this analysis, we combine all objects by allocation site, so that an "object" is closer to the notion of a type, and a unit of object coverage is an instruction-to-object pair.

Test Suite Effectiveness The Linux Test Project (LTP) [15] is the most commonly used test suite for the Linux kernel. Prior work has evaluated its code coverage using `gcov` [8], and used that data to improve test quality [19]. We show how to use object coverage to further understand and improve LTP efficacy. Figure 3 (top) shows the number of unique accesses that are covered by each test that are never covered in *any other test*, indicating which tests contribute the most unique coverage and thus should be prioritized in testing. For example, (`fs`, `syscall`) contains thousands of unique edges, whereas other tests (`ipc`, `nptl`, `pipes`) are interestingly entirely covered by other tests. Figure 3 (bottom) shows the percent of the accesses covered by each test's runs compared to the entire LTP suite, indicating how incomplete each test is and can be used to know when to add more tests. Interestingly, *ltplite* (a lightweight version of the complete LTP) covers only 60.3% of accesses compared to the complete testing suite.

We also trace the `kernel` and `systems` benchmarks from the Phoronix [34] test suite. The Phoronix benchmarks encounter a total of 664 accesses that were not covered by LTP. Memorizer can map these low-level traces back to high-level models, which would enable us to extend LTP's test cases accordingly. As can be seen, Memorizer is a powerful tool for measuring and improving test coverage.

Future Explorations These analyses show how using a large collection of diverse training data can allow valuable relative comparisons leading to concrete recommendations (*e.g.,* which tests to use). However, the ideal way to measure coverage would be to compare with a ground truth. But, no prior art explores access coverage. We believe such an approach would start with a points-to analysis; however, that has obvious shortcomings as static analysis is by definition an over-approximation. Our results suggest combining static

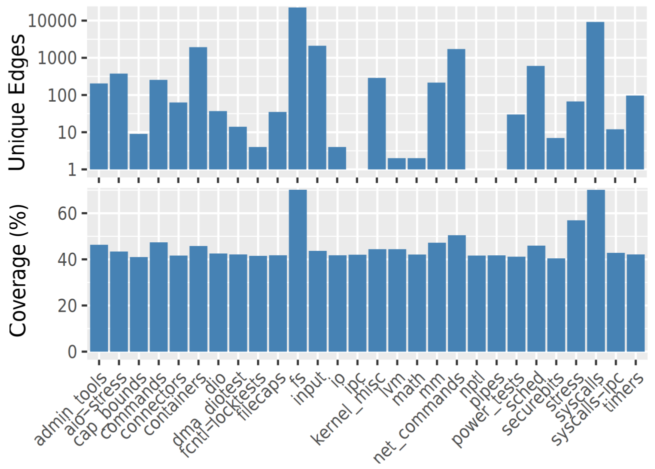


Figure 3: The relative object coverage of each LTP test (bottom) as well as the number of unique accesses not covered by any other test (top).

and dynamic access coverage for improving coverage metrics would be an impactful future direction. Furthermore, improving coverage-guided fuzzers like `syzkaller` [10], improving the precision and performance of symbolic execution, and performing optimizations like cache prefetching would be interesting extensions to our work.

7 Conclusion

In this paper, we present Memorizer, a tool to trace the Linux kernel access patterns. The primary objective is to completely log all control, memory accesses, and object allocation events. Our results indicate that it is possible to trace the complete access patterns and demonstrate powerful use cases, which we believe lay the foundation for future work in maintenance, refactoring, optimization, fuzzing, and security.

Availability

All Memorizer resources are linked at https://fierce-lab.gitlab.io/memorizer. The resources include: source code, Docker LinuxKit custom kernel builder for easy use, visualizations for dynamic exploration including flame graphs, force-directed graphs, heat maps, and data set.

Acknowledgments

We would like to thank André DeHon for his in depth analysis and enhancements to the ideas and system in this paper, as well as members of the Penn IC lab, Penn DSL lab, and Rice FIERCE lab. We would like to thank our anonymous reviewers and our shepherd, Orna Agmon Ben-Yehuda, for their precise direction that greatly improved the paper. We would like to thank AuNoLeZeZo for their unwaivering support. This research was supported by the Office of Naval Research grant BAA N00014-17-S-B010 and NSF grant CNS 2008867.

References

[1] Derek L. Bruening and Saman Amarasinghe. "Efficient, Transparent, and Comprehensive Runtime Code Manipulation". AAI0807735. PhD thesis. USA, 2004.

[2] *Chapter 6: Physical Page Allocation*. https://www.kernel.org/doc/gorman/html/understand/understand009.html.

[3] *Chapter 8: Slab Allocation*. https://www.kernel.org/doc/gorman/html/understand/understand011.html.

[4] Austin Clements. *Mtrace*. 2014. URL: https://github.com/aclements/mtrace.

[5] Austin T. Clements, M. Frans Kaashoek, Nickolai Zeldovich, Robert T. Morris, and Eddie Kohler. "The Scalable Commutativity Rule: Designing Scalable Software for Multicore Processors". In: *ACM Transactions on Computer Systems* 32.4 (Jan. 20, 2015), 10:1–10:47. URL: https://doi.org/10.1145/2699681.

[6] *Coccinelle*. https://www.kernel.org/doc/html/latest/dev-tools/coccinelle.html.

[7] eLinux. *Kernel Timer Systems - eLinux.Org*. Oct. 2, 2017. URL: https://elinux.org/Kernel_Timer_Systems.

[8] *Gcov*. https://gcc.gnu.org/onlinedocs/gcc/Gcov.html.

[9] Milos Gligoric, Alex Groce, Chaoqiang Zhang, Rohan Sharma, Mohammad Amin Alipour, and Darko Marinov. "Comparing Non-Adequate Test Suites Using Coverage Criteria". In: *Proceedings of the 2013 International Symposium on Software Testing and Analysis*. ISSTA 2013. Lugano, Switzerland: Association for Computing Machinery, 2013, pp. 302–313. URL: https://doi.org/10.1145/2483760.2483769.

[10] Google. *Syzkaller - Kernel Fuzzer*. Google, May 25, 2020. URL: https://github.com/google/syzkaller.

[11] Brendan Gregg. *BPF Performance Tools*. Addison-Wesley Professional, 2019.

[12] Brendan Gregg. *Linux Performance*. 2020. URL: http://www.brendangregg.com/linuxperf.html.

[13] Neelam Gupta and Zachary V Heidepriem. "A new structural coverage criterion for dynamic detection of program invariants". In: *18th IEEE International Conference on Automated Software Engineering, 2003. Proceedings*. IEEE. 2003, pp. 49–58.

[14] Mohammad Mahdi Hassan and James H. Andrews. "Comparing Multi-Point Stride Coverage and Dataflow Coverage". In: *Proceedings of the 2013 International Conference on Software Engineering*. ICSE '13. San Francisco, CA, USA: IEEE Press, 2013, pp. 172–181.

[15] Manoj Iyer. "Analysis of Linux test project's kernel code coverage". In: *Austin, TX: IBM Corporation* (2002).

[16] *Kernel address sanitizer*. https://www.kernel.org/doc/html/v3.13/dev-tools/kasan.html. 2018.

[17] *Kernel Memory Leak Detector — The Linux Kernel Documentation*. https://www.kernel.org/doc/html/v4.10/dev-tools/kmemleak.html.

[18] Butler W. Lampson. "Protection". In: *SIGOPS Oper. Syst. Rev.* 8.1 (Jan. 1974), pp. 18–24. URL: http://doi.acm.org/10.1145/775265.775268.

[19] Paul Larson, Nigel Hinds, Rajan Ravindran, and Hubertus Franke. "Improving the Linux Test Project with kernel code coverage analysis". In: *Proceedings of the 2003 Ottawa Linux Symposium*. Citeseer. 2003, pp. 260–275.

[20] Linux Foundation. *The Seq_fil Interface*. 2020. URL: https://github.com/torvalds/linux.

[21] *Linux ftrace*. https://www.elinux.org/Ftrace. 2018.

[22] *Linux Kernel Map*. https://makelinux.github.io/kernel/map/.

[23] *Linux kernel memory management Part 1: Introduction*. https://0xax.gitbooks.io/linux-insides/content/MM/linux-mm-1.html.

[24] *Linux kernel memory management Part 2: Fix-Mapped Addresses and ioremap*. https://0xax.gitbooks.io/linux-insides/content/MM/linux-mm-2.html.

[25] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. "Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation". In: *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '05. Chicago, IL, USA: Association for Computing Machinery, 2005, pp. 190–200. URL: https://doi.org/10.1145/1065010.1065034.

[26] Jonas Maebe, Michiel Ronsse, and Koen De Bosschere. "DIOTA: Dynamic Instrumentation, Optimization and Transformation of Applications". eng. In: *Compendium of Workshops and Tutorials Held in conjunction with PACT'02: International Conference on Parallel Architectures and Compilation Techniques / Charney, M.; Kaeli, D. (eds.), Charlottesville, Va, 2002*. 2002.

[27] Larry McVoy and Carl Staelin. "lmbench: Portable Tools for Performance Analysis". In: Jan. 1996, pp. 279–294.

[28] *memleak-bpfcc(8) — bpfcc-tools — Debian unstable — Debian Manpages*. URL: https://manpages.debian.org/unstable/bpfcc-tools/memleak-bpfcc.8.en.html.

[29] Ingo Molnar. *LKML: Ingo Molnar: Kernel/Timer.c Design (Was: Re: Ktimers Subsystem)*. 2005. URL: https://lkml.org/lkml/2005/10/19/46.

[30] Eduard Gabriel Munteanu. *Documentation/vm/kemtrace.txt*. https://lwn.net/Articles/327579/.

[31] Nicholas Nethercote and Julian Seward. "Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation". In: *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '07. San Diego, California, USA: ACM, 2007, pp. 89–100. URL: http://doi.acm.org/10.1145/1250734.1250746.

[32] D. L. Parnas. "On the Criteria to Be Used in Decomposing Systems into Modules". In: *Commun. ACM* 15.12 (Dec. 1972), pp. 1053–1058. URL: http://doi.acm.org/10.1145/361598.361623.

[33] Manfred Paul. *CVE-2020-8835: Linux Kernel Privilege Escalation via Improper eBPF Program Verification*. 2020. URL: https://www.thezdi.com/blog/2020/4/8/cve-2020-8835-linux-kernel-privilege-escalation-via-improper-ebpf-program-verification.

[34] *Phoronix Test Suite*. https://www.phoronix-test-suite.com. 2018.

[35] Sergej Proskurin, Marius Momeu, Seyedhamed Ghavamnia, Vasileios P Kemerlis, and Michalis Polychronakis. "xMP: selective memory protection for kernel and user space". In: *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2020, pp. 563–577.

[36] S.R. Schach, B. Jin, D.R. Wright, G.Z. Heller, and A.J. Offutt. "Maintainability of the Linux Kernel". In: *IEE Proceedings - Software* 149.1 (Feb. 2002), pp. 18–23.

[37] D. Scholz, D. Raumer, P. Emmerich, A. Kurtz, K. Lesiak, and G. Carle. "Performance Implications of Packet Filtering with Linux eBPF". In: *2018 30th International Teletraffic Congress (ITC 30)*. Vol. 01. Sept. 2018, pp. 209–217.

[38] Ben Schoon. *Google calls out Samsung for 'unnecessary' changes to Android's kernel*. 2020. URL: https://9to5google.com/2020/02/14/google-samsung-android-kernel-changes-security/.

[39] Julian Seward and Nicholas Nethercote. "Using Valgrind to Detect Undefined Value Errors with Bit-Precision". In: *Proceedings of the Annual Conference on USENIX Annual Technical Conference*. ATEC '05. Anaheim, CA: USENIX Association, 2005, p. 2.

[40] *Sparse*. https://www.kernel.org/doc/html/latest/dev-tools/sparse.html.

[41] *strace(1) - Linux manual page*. URL: http://man7.org/linux/man-pages/man1/strace.1.html.

[42] *The Undefined Behavior Sanitizer - UBSAN*. https://www.kernel.org/doc/html/v4.11/dev-tools/ubsan.html.

[43] Michael Whalen, Gregory Gay, Dongjiang You, Mats Heimdahl, and Matt Staats. "Observable Modified Condition/Decision Coverage". In: May 2013.

[44] John Whaley and Martin Rinard. "Compositional Pointer and Escape Analysis for Java Programs". In: *ACM SIGPLAN Notices* 34.10 (Oct. 1, 1999), pp. 187–206. URL: https://doi.org/10.1145/320385.320400.