Forensic Science International: Digital Investigation 56 (2026) 302045

Contents lists available at ScienceDirect

Forensic Science International: Digital Investigation

journal homepage: www.elsevier.com/locate/fsidi

DFRWS EU 2026 - Selected Papers from the 13th Annual Digital Forensics Research Conference Europe

LEMON: A universal eBPF-based volatile memory acquisition tool for modern android devices and hardened linux systems

Andrea Oliveri *, Marco Cavenati, Stefano De Rosa, Sudharsun Lakshmi Narasimhan, Davide Balzarotti

EURECOM, France

A B S T R A C T

Acquiring volatile memory (RAM) from modern Linux and Generic Kernel Image (GKI) Android systems has become increasingly difficult due to recent hardening features such as Secure Boot, Kernel Lockdown, and strict module signing and loading policies. As a result, traditional open-source tools such as LiME and AVML are no longer viable, preventing the acquisition and analysis of complete physical memory dumps.

To address this limitation, we introduce LEMON, the first eBPF-based universal memory acquisition tool for both hardened Linux systems and modern GKI Android devices, extending the range of devices on which volatile memory acquisition is possible. LEMON *requires neither kernel source code, code signing, nor prior deployment*. It is compatible with `x86_64` and `ARM64` architectures and supports acquisition either to local storage or over the network in standard forensic file formats.

In this paper, we provide a detailed description of LEMON's implementation, compare it with state-of-the-art open-source acquisition tools, evaluate the byte-level atomicity of its dumps and its acquisition time against existing alternatives, and show the deployment and use of LEMON on two real GKI-equipped Android phones. Furthermore, to enable a complete memory forensics analysis chain on modern Android phones, we adapt a method for generating Volatility 3 profiles from BTF debug information emitted by the kernel at runtime. In a simulated scenario on a real phone, this enabled us to recover contact details from the volatile memory of a terminated password manager. The contact details were unavailable in persistent storage and entirely absent from the logs, contradicting the belief that disk forensics alone is sufficient to extract all relevant evidence from an Android device.

Finally, in the spirit of open science and to support the forensics community, we release LEMON as an open-source project.

1. Introduction

Volatile memory forensics is a cornerstone of digital forensics, enabling incident response analysts and forensic investigators to uncover transient artifacts in system memory (RAM) that are not saved in persistent storage. In Linux and Android systems, RAM often contains important evidence of cybercrimes and real-world offenses, from credential theft and privilege escalation traces on servers and desktops, to messaging data, authentication tokens, and location records on Android smartphones. However, the evolution of system security mechanisms has introduced significant challenges to traditional volatile memory acquisition techniques on these two operating systems, which are similar in their Linux foundations yet present differences in their security design and deployment.

Historically, volatile memory acquisition on Linux systems relied on kernel modules, such as LiME (Sylve, 2012), to extract full physical memory dumps. However, the introduction of Secure Boot and Kernel Lockdown mechanisms now requires all kernel modules to be signed with pre-enrolled keys, making traditional kernel acquisition tools unusable in many scenarios. Disabling these protections involves rebooting the machine, which typically (Halderman et al., 2009; Müller and Spreitzenbarth, 2013) results in the irreversible loss of memory contents, unacceptable during forensic investigations.

As a workaround, user-space solutions like AVML (Microsoft, 2023) have been proposed, but their effectiveness is increasingly limited by modern system security hardening, which also restricts access to user-space to kernel interfaces needed to dump the physical memory of the system.

In Android, memory acquisition has long been hindered by vendor fragmentation. Prior to the introduction of the Generic Kernel Image (GKI), device manufacturers used heavily customized kernels, often without releasing the source code and, more critically, disabling support for kernel modules or restricting it to OEM-signed ones. This made nearly impossible to compile and execute kernel level acquisition tools such as LiME on real devices. GKI was introduced to address vendor fragmentation by decoupling a common, OEM-independent, kernel from vendor-specific modules. However, GKI enforces a kernel module loading policy (Google, 2025c) that does not rely on module signatures but on per-device and per-vendor whitelists of exported kernel functions accessible to external modules, again preventing the creation of a

* Corresponding author.

https://doi.org/10.1016/j.fsidi.2026.302045

universal, vendor and model-agnostic memory acquisition solution for Android devices.

The lack of reliable acquisition tools has hindered the practice of volatile memory forensics on Android devices, discouraging development of analysis frameworks for RAM dumps. Even popular platforms like Volatility 3 (Walker, 2017) lack compatible versions and plugins for ARM-based Android devices. Additionally, these tools cannot interpret physical memory dumps from such systems, as no general method exists to create valid profiles for specific kernels without debug builds.

Contribution – To extend the range of devices on which volatile memory acquisition is possible and address the deployment limitations of existing approaches, in this work we present LEMON which, to the best of our knowledge, is the first tool capable of acquiring full physical memory dumps from security-hardened Linux systems and Android devices running GKI kernels. Unlike existing tools, LEMON relies on the existing in-kernel eBPF infrastructure and does not require compilation against device-specific kernel headers, prior deployment on the target, or key pre-enrollment. It runs on both `x86_64` and `ARM64` architectures and is distributed as a static binary, with no runtime library dependencies, enabling complete physical memory acquisition saving the dump to local storage or transmitting it over the network using standard forensic file formats.

We conduct a detailed comparison of LEMON with state-of-the-art tools such as LiME and AVML, highlighting its advantages in terms of compatibility, deployability, and usability in hardened Linux and Android devices. We evaluate its performance across multiple Linux environments, measuring both acquisition time and dump byte-level atomicity. Additionally, we demonstrate LEMON's applicability to real-world Android devices by successfully acquiring memory from two modern smartphones not supported by LiME and AVML. By extending `btf2json` (Obst, 2024) tool, we generate a Volatility 3 profile for one of the devices, enabling the use of a Volatility 3 version adapted for `ARM64` devices. This allows us to extract crucial evidence of a simulated potential crime from a memory dump obtained from one of the two real phones, clearly illustrating that analyzing only the persistent storage of an Android device, as is common in most real-world cases, fails to provide investigators with critical information that reside exclusively in RAM.

2. Acquisition problems in Linux and Android

Volatile memory acquisition on modern Linux and Android systems presents complex technical challenges due to the lack of a standardized, universal, and stable interface for accessing physical memory. In addition, acquisition tools must handle a diverse and evolving set of kernel-level restrictions and platform-specific configurations that might have severe consequences on the feasibility and reliability of the acquisition process.

2.1. Linux acquisition

On Linux-based systems, two common approaches are used to perform memory acquisition. The first involves the use of special virtual devices (such as `/proc/kcore`), which allow user-space tools to read the physical memory of the system. This approach is employed by AVML (Microsoft, 2023), a portable user-space memory acquisition tool, designed primarily for cloud and production environments. AVML supports multiple user-space virtual devices for memory access, including `/proc/kcore`, `/dev/crash`, and `/dev/mem`. While this technique offers an effective mechanism for memory acquisition that does not require kernel modifications or code injection, the availability of this interface is increasingly constrained. In fact, on many modern distributions or systems with hardened compile-time kernel configurations, these virtual devices are disabled to prevent access to sensitive kernel information.

The second approach relies instead on custom kernel modules to directly read physical memory in kernel space and write it either to a file or transmit it over the network. This is the solution adopted by the open-source tool LiME (Sylve et al., 2012). Supporting multiple output formats, LiME has established itself as a reference tool in Linux memory forensics, and it has also defined the de-facto dump standard file format in this domain. While effective and widely adopted, LiME (and any other tool based on kernel modules) must be compiled against the exact same header files used by the target system, which significantly limits its portability as they are often unavailable, especially on mobile or IoT devices. Moreover, this approach requires the ability to load the compiled kernel module into the running system, a capability that is tightly regulated on modern Linux distributions. Compile and runtime kernel configuration flags can deny module loading entirely or enforce strict signature verification (`CONFIG_MODULE_SIG_FORCE`). These restrictions are further reinforced in systems with Kernel Lockdown. If enabled at compile time, the Kernel Lockdown feature–introduced in Linux 5.4 and enabled by default in kernels of widely used distributions such as Ubuntu, Debian, Fedora, and RedHat–is automatically enforced when Secure Boot is active and can operate in either *integrity* or *confidentiality* mode. In integrity mode, only kernel modules signed with trusted keys can be loaded. Adding a new signing key requires a reboot, which invalidates the content of the RAM and destroys the possibility of performing a forensics analysis. Confidentiality mode is even more restrictive: it disables access to various kernel subsystems, even for `root`. It removes interfaces such as `/proc/kcore`, and it blocks eBPF programs that might expose kernel data. However, the default activation of this security mode at kernel compile-time raises criticism (Garrett, 2020) due to the number of standard system administration tools it breaks. As a result, its deployment has remained opt-in by default and not used by any major Linux distribution, requiring system administrators to recompile the kernel to enable it explicitly. These constraints significantly limit the use of kernel-module-based memory acquisition tools on systems with full Secure Boot or enforced module signatures. Consequently, reliable acquisition increasingly depends on pre-installed and signed agents, an impractical scenario for post-mortem forensic investigations, where the analyst had no prior access to the target system.

In this context, any portable and robust memory acquisition strategy must address not only the technical aspects of memory access but also the layered security controls enforced by the kernel, such as compile and runtime kernel configuration flags, signature verification policies, and boot-time integrity mechanisms. It is therefore crucial to develop a universal memory acquisition tool capable of operating across these constraints.

2.2. Android acquisition

Android runs a modified version of the Linux kernel, which introduces specific modifications to support mobile hardware and to increase its security by further restricting kernel memory access and module loading. These restrictions are present on both legacy Android kernels and modern ones, enforced through different mechanisms.

On Android devices, standard user-space interfaces for accessing kernel memory, such as `/dev/kmem`, `/proc/kcore`, and `/dev/mem`, are generally disabled at compile time (Google, 2025a), completely precluding the use of user-space acquisition tools like AVML. Furthermore, on most legacy Android devices (pre-Android 12), kernel module loading is restricted beyond the standard Linux requirement of `root` privileges, being entirely disabled at boot by setting `/proc/sys/kernel/modules_disabled`. Even when module loading is allowed, production kernels enforce signature verification accepting only modules signed with device vendor keys. In these conditions, kernel modules for memory acquisition are mostly limited to experimental environments, like the Android Emulator (AVD).

The Android security model evolved with the introduction of the Generic Kernel Image (GKI) (Google, 2025b) in Android 12, which

became mandatory in Android 13. GKI replaced fragmented, vendor-customized kernels with standardized kernels maintained by Google. Any device-specific functionality was moved into vendor modules that interact with the generic kernel through a stable API. While GKI continues to block userspace access to privileged devices that expose kernel memory, it introduces a new model for kernel module loading. Post-boot loading of unsigned modules is now allowed, but these modules are loadable only if they use exported symbols within a whitelist. This whitelist, hardcoded at compile time in the signed kernel binary, and enforced at runtime, is defined *by the vendor* and changes *for each single device model*. As a result, LiME, which was useable on pre-Android 12 kernels in cases when headers were available, is incompatible with the new restrictions. Even with access to the GKI kernel source, building a universal dumping kernel module is not feasible, since each device may expose a different set of allowed symbols.

Moreover, *to the best of our knowledge, no kernel module or other open-source tool is available to acquire the volatile memory of real Android devices equipped with GKI kernels*.

2.3. Leveraging eBPF to acquire volatile memory

Originally designed to filter network packets, eBPF (extended Berkeley Packet Filter) is a powerful and flexible *event-driven* in-kernel virtual machine, already present in Linux kernels of widely used distributions as well as in GKI Android kernels, which enables the safe and efficient execution of user-defined bytecode (Vieira et al., 2020). eBPF programs are small, sandboxed snippets of code written in a restricted `C` dialect, compiled to eBPF bytecode, and attachable to various kernel hooks (e.g., networking stack, Kprobes, Uprobes, etc.) executed in the kernel context whenever the corresponding event occurs. Before an eBPF program is loaded into the kernel by a user-space program, it is validated by a strict in-kernel static verifier that checks for safety properties to ensure that eBPF programs cannot crash or compromise kernel stability. An eBPF program can use only a restricted set of kernel functions called *eBPF helper functions* that enable it to interact with the system (e.g., by redirecting packets, reading user and kernel memory, getting the current process context, etc.). To exchange data with user-space programs, eBPF programs can leverage *maps*–shared data structures that facilitate a secure communication between eBPF programs and user space.

To improve eBPF's portability, kernel 5.17 (early 2022) introduced the *CO-RE (Compile Once – Run Everywhere)* technology, backported by some Android vendors also to previous kernels of series 5.x. CO-RE allows to compile an eBPF program once and run it on any other system, even with different kernel versions or configurations without the need to recompile it. This is made possible through the embedding in the kernel binary of *BTF (BPF Type Format)* types which expose the definitions of types, structures layouts, and eBPF helper functions for the specific kernel version and configuration. If the kernel is compiled with option `CONFIG_DEBUG_INFO_BTF`, which is always the case for Linux distributions kernels and Android GKI kernels, the eBPF verifier can automatically modify the eBPF program at load time to adapt it to the layout of the kernel data structures specified by the BTF information. This model drastically simplifies the distribution of eBPF programs, especially when the kernel headers or the kernel configuration are not available. Furthermore, unlike kernel modules that must be cryptographically signed with a key loaded at boot time, eBPF programs do not require any digital signature. As a result, eBPF programs are more portable than kernel modules, and therefore they rapidly became the foundation for a broad range of applications (Cilium, 2025; Hubble, 2025; Isovalent, 2025; Falco, 2025; Tracee, 2025; Merbridge, 2025; Facebook, 2025).

Its portability, along with the fact that eBPF programs do not require prior signing and bypass SecureBoot and Kernel Lockdown integrity, makes it an ideal candidate for developing a portable and universal tool for RAM acquisition on hardneded Linux and Android systems.

3. LEMON

In light of the considerations discussed in the previous section, in particular the limitations imposed by SecureBoot hardened Linux systems and modern Android devices adopting the GKI kernel architecture, and given the capabilities offered by eBPF, we developed a memory acquisition tool that leverages eBPF to overcome the constraints of existing solutions: LEMON.

LEMON consists of two tightly integrated components bundled within a single ELF executable. The first is a user-space program, statically compiled to avoid dependency issues on the target system. This component is responsible for loading the eBPF program, coordinating the memory acquisition process, saving the output to disk in the LiME file format, facilitating the integration of LEMON into forensic analysis pipelines, or sending it over the network for remote collection. The second component is the eBPF program itself, which performs in-kernel physical memory reads when instructed by the user-space component. LEMON can be compiled in three different modes: a CO-RE (Compile Once, Run Everywhere) mode for BTF-enabled kernels, a legacy mode that compiles against specific kernel headers for systems without BTF support, and a universal legacy mode that produces a no CO-RE binary without requiring specific kernel headers, making it useable even on no CO-RE/BTF capable kernels when the headers are unavailable. This third build option is possible because LEMON's eBPF component is carefully designed to minimize dependencies on kernel data structures and helper functions, relying only on the most backward-compatible features. As a result, LEMON supports Android and Linux kernels starting from version 5.5 (released in 2020) and runs on `x86_64` and `ARM64` architectures.

Finally, to support the memory forensics community we release LEMON as open-source project (Oliveri et al., 2025b).

The memory acquisition process is divided into several steps, as summarized in Fig. 1. At the beginning (Step 1), the analyst transfers the LEMON binary to the target system and executes it with suitable options, for example, to specify whether the dump should be saved locally or transmitted over the network. In Step 2, LEMON verifies system compatibility and, parsing `/proc/kallsyms`, locates the head of the in-kernel linked list of the physical memory regions installed and the base of the kernel direct memory mapping region. This region of the kernel virtual address space provides a 1:1 linear mapping of the entire physical address space of the system. Thus, LEMON can access any specific physical page simply by accessing the corresponding virtual address calculated as:

$$\texttt{virtual addr} = \texttt{physical addr} + \texttt{direct mapping base}$$

It is important to note that `/proc/kallsyms` contains these symbols only if the kernel is compiled with `CONFIG_KALLSYMS_ALL` option, which is mandatory in Android GKI kernels but can be disabled in older

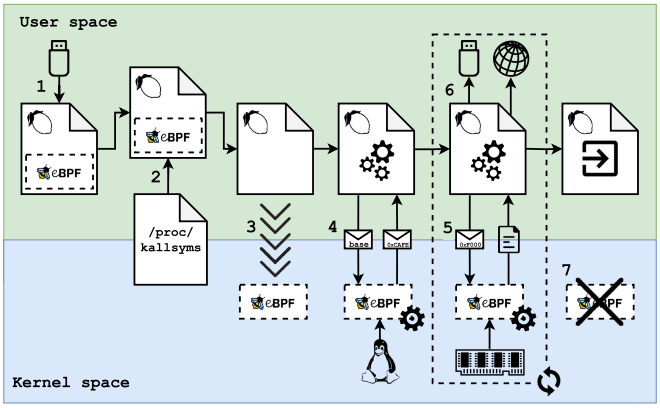

Fig. 1. Overview of the acquisition functioning of LEMON.

kernels.

In Step 3, the user-space component loads the eBPF program into the kernel. As discussed in Section 2.3, eBPF programs are inherently event-driven.[1] To be able to trigger an event, LEMON registers a Uprobe (Linux Foundation, 2025) on one of its own user-space component functions and assigns it to the LEMON eBPF program. A Uprobe allows dynamic instrumentation of a user-space function by inserting a breakpoint at its entry point. Then, when the function is called, at its first instruction, the breakpoint is triggered, and the kernel executes the associated eBPF code.

Then LEMON (Step 4) obtains the `direct mapping base` address from a kernel global variable and walks the list of physical memory regions to be dumped, reading them directly from the kernel memory and considering only the ones marked as `System RAM`, ignoring reserved and MMIO regions. To do it, LEMON passes the kernel data structure and global variable's virtual address and size to read as arguments of the Uprobe instrumented function and then calls it. This invocation triggers the registered Uprobe, causing the kernel to execute the eBPF program. The eBPF program extracts, from the instrumented function arguments, the virtual address and the size to be read and uses the `bpf_probe_read_kernel()` eBPF helper function to read its content. It then stores the data in an eBPF map and returns the control to the LEMON user-space component.

During Step 5, LEMON iterates over each 2 MiB huge page[2] to be acquired, computes its virtual address in the direct mapping region, and invokes the instrumented function, passing this address and the huge page size as arguments and the kernel then returns the page content through the eBPF map.

Finally, in Step 6, the user-space component retrieves the content of the huge page from the map and writes it to disk or transmits it over the network, according to the option specified by the user. At the end of the acquisition process, LEMON unloads the eBPF program from the kernel and exits (Step 7).

3.1. Comparison with other acquisition tools

Table 1 compares LiME, AVML, and our proposed tool, LEMON. As previously discussed, LiME operates as a kernel module, implementing its memory acquisition logic and accessing physical memory entirely from kernel space. In contrast, AVML relies on existing user-exposed mechanisms to access physical memory. LEMON introduces a different approach by decoupling the memory acquisition logic, which resides in user space, from the memory access mechanism performed by eBPF programs loaded into kernel space, reducing the tool's footprint on the kernel's privately allocated memory regions.

Unlike LiME, which requires compiling a kernel module against specific kernel headers, LEMON leverages the eBPF virtual machine and, when BTF is supported, avoids any direct dependency on the running kernel's headers. However, if BTF is not supported, LEMON can still be compiled in a universal non-CO-RE mode and run even on kernels without BTF support, thanks to its design that requires no dependencies on internal kernel structures and headers.

LiME and AVML must always be started from a process running in a security context with elevated Linux capabilities, such as `CAP_SYS_ADMIN`, which is generally equivalent to running them as `root`. In contrast, the architecture of LEMON and its design choices, such as retrieving the list of physical memory regions directly from the kernel instead of from `/proc/iomem`, allow it to reduce the required capabilities significantly. Although LEMON typically requires execution with `root` privileges to disable specific security measures such as `kptr_restrict`, in the best-case scenario, it can be executed by a process with only `CAP_BPF` and `CAP_SYSLOG` capabilities, making it useable even on devices where an exploit provides only these capabilities but does not grant full `root` permissions.

Furthermore, in contrast to the other two tools, since eBPF programs do not require cryptographic signatures, LEMON can be used on systems with Secure Boot, Kernel Lockdown `integrity` mode, and policies that prohibit the loading of unsigned kernel modules or restrict access to user-space virtual devices that allow reading the physical memory of the system.

As shown in Table 1, all three tools support two file formats: a raw dump and the LiME file format, which is the de facto standard for Linux and Android memory acquisition. However, unlike AVML, LEMON supports both disk and network acquisition, and, as we will demonstrate in Section 4.1, sending the acquired pages over the network significantly reduces the memory footprint of the acquisition process, preserving a larger portion of potentially valuable forensic artifacts, especially those residing in memory regions recently deallocated by the system.

Finally, LEMON is the only available tool that supports the acquisition of RAM from recent Android devices and hardened Linux systems protected by Secure Boot extending the range of devices on which volatile memory acquisition is possible.

Table 1
Features and requirements of LiME, AVML and LEMON.

	LiME	AVML	LEMON
Binary type	Kernel	User	Hybrid [a]
Run without root privileges	×	×	× [b]
Kernel headers independent	×	✓	✓ [c]
Run without signature	×	✓	✓
Dump format	LiME Raw	LiME Raw	LiME Raw
Dump on disk	✓	✓	✓
Dump over network	✓	× [d]	✓
Runs on Hardened Linux	×	×	✓
Runs on Modern Android	× [e]	×	✓

[a] The acquisition logic is implemented in the user component, while the memory read primitive runs in the eBPF in-kernel virtual machine.
[b] In the best scenario requires only `CAP_BPF` and `CAP_SYSLOG`.
[c] If compiled in CO-RE or universal legacy mode.
[d] It uploads the dump to the cloud after its acquisition on disk.
[e] It supports only old non-GKI kernels, typical of pre-Android 12 releases, and only if kernel headers are available.

4. Experiments and results

In this section, we evaluate LEMON's capabilities compared to other volatile memory acquisition tools available for Linux and Android, focusing particularly on the byte-level atomicity of the dumps produced by the three tools. Following an approach inspired by Campbell (2013) and more recent works by Rzepka et al. (2025) and Oliveri and Balzarotti (2025), we define byte-level atomicity as the number of bytes that differ between a memory dump acquired by a tool and an atomic snapshot of the system's memory obtained freezing the entire system during the acquisition process, as is performed during the acquisition of a virtual machine using the hypervisor interface.

It is important to note that LEMON is introduced to broaden the range of devices on which volatile memory acquisition is possible, rather than to provide performance improvements over existing solutions. The measurements of byte-level atomicity and execution time are presented only to show that LEMON performs on par with current tools in terms of efficiency and reliability, while significantly extending the spectrum of

[1] It exists also a test mode, `BPF_PROG_TEST_RUN`, that allows user-space programs to call an eBPF program on demand. It is used by LEMON in case Uprobe support is unavailable but introduces an overhead.

[2] Linux and Android kernels map all physical memory regions into the direct mapping area using 2 MiB huge pages, rather than the default 4 KiB pages, to reduce TLB flushes. By reading memory at the same granularity, LEMON minimizes the number of memory access operations required from the kernel. In contrast, LiME and AVML perform the dump using a 4 KiB page size.

supported devices.

After this, we present two examples of volatile memory acquisition on two different real Android devices and demonstrate how acquiring only the persistent storage of an Android device, as currently performed in most real-world cases, fails to provide analysts with critical evidence that may reside exclusively in RAM. To this end, we adapt an existing tool for generating Volatility 3 profiles from BTF symbols to account for Android-specific characteristics, and we present a minimal example of structured data extraction from a memory dump acquired with LEMON on a real Android phone using Volatility 3.

4.1. Runtime performance comparison

Similar to other volatile memory acquisition tools, LEMON performs a *live* physical memory acquisition – i.e., the operating system and user-space applications continue to execute during the acquisition process. Consequently, the resulting memory dump does not constitute an atomic snapshot of the system's memory state; instead, it may be affected by ongoing system activity, including that of the acquisition tool itself. Past (Kornblum, 2007; Moser and Cohen, 2013; Case and Richard III, 2017; Pagani et al., 2019) and recent (Ottmann et al., 2023; Rzepka et al., 2024, 2025; Oliveri and Balzarotti, 2025) works has extensively documented that inconsistencies can significantly undermine the coherence of the resulting memory dumps and, in extreme cases, render post-mortem analysis unreliable.

To assess the extent to which this issue affects LEMON, we conducted a comparative evaluation aimed at quantifying the divergence between memory dumps acquired using AVML, LiME and LEMON and those approximating an ideal atomic snapshot. Each tool was evaluated under all supported acquisition modes: disk-based (supported by all tools) and network-based (supported by LEMON and LiME). For the comparison we used the latest versions of the tools at the time of writing: LiME commit `1f99bc6`, AVML v1.4.0 and LEMON 1.2.

Since obtaining an *atomic* memory dump from a physical machine is not possible, this experiment was conducted within virtual machines (VMs) running under QEMU/KVM. This environment enabled us to initialize each experiment from an identical, controlled system state and to obtain atomic memory snapshots by pausing the virtual machine and capturing its memory image. These snapshots served as ground truth and were compared against the memory acquired using live acquisition tools. It is important to emphasize that the use of virtualization was solely for experiment control and measurement purposes; it does not constitute a requirement or dependency of the acquisition tools themselves. Furthermore, the usage of different distributions or kernel versions does not affect the results of the experiments as LEMON depends only on kernel features that are always enabled in modern Linux distributions and the measurements are relative between the tools and not absolute.

To conduct our experiments, we employed two distinct `x86_64` virtual machines, each configured with 4 virtual CPUs and 8 GB of RAM, a virtual hard disk connected to a virtual USB 3 controller and a gigabit network interface.

The first was a minimal system running only the Linux kernel, a shell and the acquisition tool to test, allowing us to isolate and quantify the intrinsic impact of the acquisition tools on the resulting dump. The second was a Ubuntu 25.04 desktop environment, simulating typical user activity by running a full desktop session with a web browser (with several open tabs) and a PDF viewer. The host machine in both the experiments was instead equipped with an Intel i7-1365U CPU 12 cores, 32 GB of RAM and an NVMe SSD for storage running Fedora 42.

Due to the fact that neither LiME nor AVML can run on modern Android systems, as explained in Section 2.2, we focus only on Linux-based systems for this set of experiments.

Algorithm 1. Dump Atomicity Experiments Workflow

```
1: Initialization:
2: VMgroundstate←savevm()
3: for iteration in repetitions do
4:   Prepare idle state:
5:   VMstate ←VMgroundstate
6:   vm_start()
7:   wait()
8:   Acquire atomic dump:
9:   vm_pause()
10:  VMidlestate←savevm()
11:  Atomicdump←dump_guest_memory()
12:  vm_resume()
13:  for tool in [LiME, AVML, LEMON] do
14:  for mode in supported_modes(tool) do
15:  Acquire using tool:
16:  VMstate ←VMidlestate
17:  Tooldump←tool()
18:  vm_shutdown()
19:  Compare atomic and tool dump:
20:  diff←diff(Atomicdump, Tooldump)
21:  measure[tool][mode][iteration]←diff
```

Algorithm 1 outlines the automatized workflow followed to conduct the experiments on both virtual machines. We begin by taking a snapshot of the powered-off VM (referred to as the ground state), which serves as the common starting point for all repetitions of the experiment (Line 1).

In our experiment, for each VM, we perform 10 repetitions in which (Lines 4–7) we restore the VM to the ground state, start it, and wait for it to complete the boot process. After booting, we wait 10 min to ensure that all startup applications have finished executing and the kernel and different user-space programs reach an idle state. We then (Lines 8–12) take a snapshot of the running machine, which is used to execute each acquisition tool under consistent initial conditions. With the VM's execution paused, we use the `dump_guest_memory` QEMU/KVM command to obtain an atomic snapshot of its memory.

Next, we restore the VM to the idle-state snapshot, resume execution and run one of the tools in one of its supported modes–disk or network for LiME and LEMON, and disk-only for AVML (Lines 15–18)–producing a dump in LiME format.

Finally (Lines 19–21), we retrieve the dump produced by the acquisition tool and compare it against the atomic snapshot using a custom analysis tool that we have developed (Oliveri et al., 2025c). The comparison is performed at the byte level, computing the total number of differing bytes. This process is repeated 10 times for each supported dump mode of each tool.

In Fig. 2, we report the percentage of bytes over the RAM size that differ between the atomic dump and the dumps acquired by the various tools in each dump mode for the minimal machine. For each tool and dump mode, we performed 10 independent acquisitions, each represented as a separate bar in the plot. This experiment, in particular,

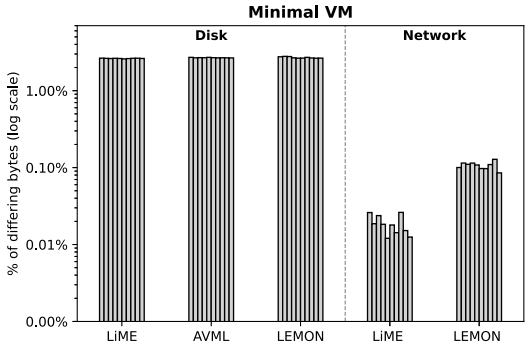

Fig. 2. Percentage of differing bytes between the atomic dump and dumps acquired by LiME, AVML and LEMON on a minimal Linux system, experiments were repeated 10 times for each configuration.

allows us to isolate and quantify the direct impact of the acquisition tool itself on memory dump consistency, without interference from unrelated user-space activity, thereby representing the best-case scenario in case of memory acquisition.

We can observe two distinct and noteworthy behaviors. First, the byte-level atomicity of all three tools when saving the dump to disk is comparable. On the other hand, there is a significant discrepancy in atomicity between dumps performed via disk and those performed over the network. Specifically, considering the Minimal VM, in the disk mode, an average of 230 MB, corresponding to 2.68 % of total memory, differs from the atomic dump, with a maximum variance of just 5 MB (less than 0.06 % of memory) between the three tools. This confirms that LEMON performs, in terms of byte-level atomicity, on par with AVML and LiME in disk acquisition mode.

Conversely, LEMON shows lower byte-level atomicity performance than LiME during network-based memory acquisition, which is expected given that LiME operates entirely in kernel space. This difference is likely due to LEMON's need to copy data multiple times between user and kernel space.

Nevertheless, both tools achieve near-atomic behavior during network acquisition, introducing less than 10 MB of inconsistencies (below 0.2 % of total memory), which is approximately 23 times less than what is observed in disk-based acquisition (even, as we will see later, network acquisition requires five times more time). This strongly suggests that, when feasible, memory acquisition over the network is preferable to dumping to disk, confirming the results of Oliveri and Balzarotti (2025).

These observations are further corroborated by the results obtained on the full-fledged Ubuntu machine, shown in Fig. 3. Even if in this case we observe an order-of-magnitude increase in the average number of differing bytes for both disk and network dumps, we still observe the same comparable performances among the three tools during disk acquisition, as well as the same pronounced gap between the byte-level atomicity of disk and network-based dumps.

In addition we also measured the time required by each tool to acquire memory. Since this test can be reliably performed on both virtualized and physical environments, we conducted it on both QEMU/KVM-based virtual machines and a bare-metal system equipped with 32 GB of RAM running Ubuntu 24.04 with Secure Boot and Kernel Lockdown `integrity` mode enabled. Testing on this physical hardware and software configuration allowed us also to verify that the virtualized environment does not introduce measurement distortions due to emulated devices and demonstrate that LEMON is able to correctly run also in case of hardened Linux systems. The results indicate that acquisition times in virtual machines align with those on the physical machine, scaled proportionally to RAM size, confirming the consistency of the virtualization measurement approach.

The measurements obtained on the physical machine, as shown in

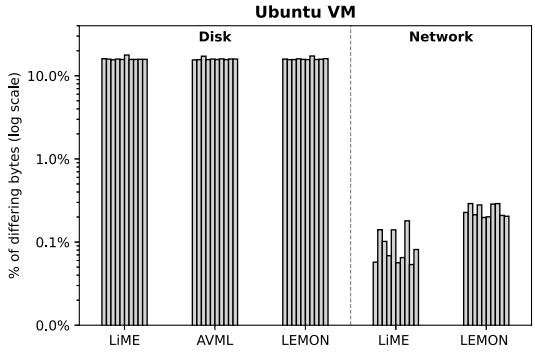

Fig. 3. Percentage of differing bytes between the atomic dump and dumps acquired by LiME, AVML and LEMON on a full-fledged Ubuntu desktop system, experiments were repeated 10 times for each configuration.

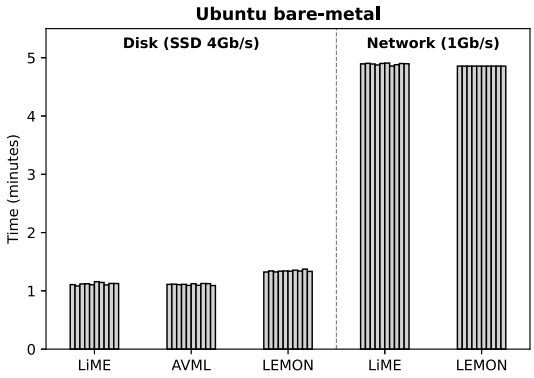

Fig. 4. Time required to acquire a full memory dump from a bare-metal Ubuntu 24.04 system with 32 GB of RAM, experiments were repeated 10 times for each configuration.

Fig. 4, indicate that all tools are capable of saturating the available bandwidth of their respective acquisition mediums – 4 Gb/s for the disk and 1 Gb/s for the network interface–which emerges as the primary factor influencing acquisition time. In the case of network acquisition, LEMON and LiME perform similarly, both fully utilizing the available network bandwidth. For LEMON, the variance between runs was so small that the differences are not appreciable at the scale of the figure, which is why bars appear identical. Notably, although network acquisition is considerably slower due to physical bandwidth limitations rather than software overhead, it offers the advantage of producing more atomic dumps, as demonstrated in the previous experiments and as a consequence of the reduced number of kernel caches in which the data is copied before being sent over the network (Oliveri and Balzarotti, 2025).

In disk-based acquisition, LEMON performs on average 19 % slower than LiME and AVML. An analysis of the source code of the other two tools revealed that only LEMON issues an `fsync()` at the end of the dump to ensure that the acquired RAM contents are fully written to disk before the application exits. In a confirmation experiment, disabling this step in LEMON resulted in acquisition times comparable to the other tools. However, it is important to note that in scenarios such as system compromise by an attacker, the system may become unstable. In such cases, it is preferable to immediately flush all data to storage using `fsync()` at the end of the dump, as also recommended by the `libc` documentation (libc, 2025), rather than relying on the device being cleanly unmounted, when flushing would otherwise occur automatically.

In conclusion, this set of experiments confirms LEMON's effectiveness both in terms of dump byte-level atomicity and acquisition performance, with results comparable to existing tools. However, unlike LiME and AVML, LEMON is the only one of the three tools capable of performing a memory acquisition on modern hardened Linux and Android systems without any dependency on kernel source code availability.

4.2. Acquisition on real android devices

To prove the effectiveness of LEMON in real-world scenarios, we used it to acquire the memory of two Android phones:

- **Samsung Galaxy A15 (SM-A155F)** running Android 14 with the OEM-customized GKI kernel `5.10.198-android12-9-28575149`.
- **Google Pixel 6a (bluejay)** running Android 14 with stock GKI kernel `5.10.198-android13-4-00050- g12f3388846c3-ab11920634`.

As described in Section 3, we assume a scenario where the analyst either uses an exploit (Sun et al., 2015) to gain `root` privileges or has compromised a system app running in a context with sufficient Linux capabilities to execute LEMON. Since no publicly available privilege escalation exploits exist for our test phones, we rooted the devices, maintaining the original OEM kernel. Then, we used the Android Debug Bridge (`adb`) to deploy LEMON, as illustrated in Listing 1. Once executed, LEMON successfully acquired the memory of both devices using both disk and network modes.

```
# On the analyst machine
nc -l 192.168.0.1 -p 2304 > dump_net.lime &
adb push lemon.aarch64 /data/local/tmp/
adb shell

# On the Android phone
su
cd /data/local/tmp/
chmod +x lemon.aarch64
./lemon.aarch64 -d dump_disk.lime
./lemon.aarch64 -n 192.168.0.1
exit

# On the analyst machine
adb pull /data/local/tmp/dump_disk.lime
```

Listing 1. Acquire memory using disk and network modes on Android phones.

Notably, the Samsung Galaxy A15 features an OEM-customized GKI kernel without CO-RE support, as confirmed by the absence of `CONFIG_DEBUG_INFO_BTF` in `/proc/config.gz`. Despite this limitation, we successfully ran LEMON and obtained a full memory capture using a version of the tool compiled in universal no CO-RE mode. This result highlights the power of the LEMON design, which minimizes the requirements on kernel data structures and eBPF helper functions, relying solely on the most retro-compatible versions of required features and without any kernel headers.

4.3. Profile generation for real android devices

In modern memory forensics, obtaining a memory dump alone is insufficient; it is also necessary to retrieve or generate a valid profile that defines kernel data structures and global kernel variable locations, enabling tools like Volatility 3 (Walker, 2017) to accurately identify artifacts within the dump. Therefore, to demonstrate the usefulness of memory dumps obtained with LEMON on modern Android systems, we must find a way to generate valid profiles for these kernels.

Before the introduction of BTF, generating profiles required either access to a debug kernel or the kernel source code and vendor compilation configuration that generally are not available for mobile and embedded devices. However, with BTF integration, the kernel itself includes the definitions of its data structures, enabling the automatic generation of valid profiles for *any* BTF-enabled kernel. In our case, this kernel feature enables the analysis of memory dumps from the Google Pixel 6a, which exposes BTF information. To achieve this, we modified the open-source tool `btf2json` (Obst, 2024) to support Android kernel specifics, releasing it as open-source project (Oliveri et al., 2025a). The original tool extracts BTF data from the kernel binary and symbol information from the `System.map` file, which is generated at compile time and is unique to each kernel build. Android devices, however, do not store the kernel binary in `/boot` as typical Linux distributions do; instead, the kernel is stored in a raw, even encrypted format in a dedicated NAND partition, complicating its retrieval. However, if the kernel is compiled with BTF support, it exposes BTF data to user space via `/sys/kernel/btf/vmlinux`. This configuration allows us to read the BTF information directly from the file and, with a modified version of `btf2json`, use it in place of the raw kernel binary to generate the required profile.

Another peculiarity of Android is that, due to disk space constraints, the `System.map` file required by `btf2json`, is rarely included in the devices. To work around this limitation, we reconstructed its contents using the symbol information available at runtime in `/proc/kallsyms`. It is important to remember that, as explained in Section 3, all forensically-relevant symbols are exported in `/proc/kallsyms` only if the kernel is compiled with the `CONFIG_KALLSYMS_ALL` option. However, along `CONFIG_DEBUG_INFO_BTF` that enable BTF support, this option is mandatory in GKI kernels.

The `/proc/kallsyms` file lists the in-memory addresses of kernel symbols *after* the kernel has booted, and Kernel Address Space Layout Randomization (KASLR) has been applied. KASLR adds a random offset to the kernel's base address at boot time, effectively shifting the entire kernel symbol table. As a result, the addresses in `/proc/kallsyms` differ from those in `System.map`, which contains static, compile-time addresses. Thankfully, by comparing the address of `_stext` (which does not depend on the compile options nor the kernel version) to its runtime counterpart, we can determine the KASLR offset. We can then subtract this offset from all the symbol locations in `/proc/kallsyms` and obtain the original addresses in `System.map`. By applying this modification to `btf2json` in conjunction with the previously described changes related to BTF support, we are able to generate valid Volatility 3 profiles *for any Android and Linux kernel that exposes BTF symbols even without access to the kernel binary and/or the* `System.map` *file*. In Fig. 5 we summarize the entire process of acquiring the memory of a physical phone with LEMON and generating a profile by using our version of `btf2json`.

4.4. Evidence extraction from a real device

We now conclude by showing an example of the analysis we can perform on the memory dump and on the information we can extract that generally are not available in a traditional disk-only forensics investigation. This minimal example is not meant to provide a full analysis of the memory dump, but to demonstrate how using LEMON enables, for the first time, the acquisition of volatile memory on modern Android devices, allowing the extraction of evidence otherwise unattainable.

In our example, we simulate the activity of a malicious actor on a Google Pixel 6a device and impersonate a forensic analyst assigned to examine it. Specifically, the simulated malicious activity involves the suspect using a password manager application to store sensitive contact information in encrypted form on persistent storage. Once they read the contact information from the password manager, they terminate the application removing it from active processes, minimizing the traces left behind.

We emphasize that the acquisition was conducted on a real device, following the simulated malicious activity, and all the evidences presented in this section were extracted from an actual memory dump of the physical device. We have performed the acquisition only on the Pixel 6a device because, as mentioned in Section 4.2, the Samsung Galaxy A15 does not support BTF, and therefore we cannot generate a valid Volatility 3 profile for it using our modified version of `btf2json`.

We suppose that the investigator, after obtaining sufficient privileged access to the phone via an exploit, acquires a complete disk image and searches it for plaintext identifiers, such as names, phone numbers, and other contact information relevant to the investigation. However, unable to locate any of these identifiers in persistent storage, the analyst proceeds to acquire the device's volatile memory hoping to extract additional information that was not present in persistent storage.

Following the memory acquisition procedure described in the previous section and using our modified version of `btf2json` to generate a valid Volatility 3 profile compatible with the stock Pixel 6a kernel, the analyst proceeds to analyze the memory dump. For this purpose, we used a modified version of Volatility 3 (Sarkar, 2024) that adds support for `ARM64` CPU paging system.

```
PID     TID   PPID        COMM          EXIT_STATE
....                      ....            ....
1887    2994     1   android.hardwar   TASK_RUNNING
2916    2963     1   pixelstats-vend   TASK_RUNNING
9970   10100  847   pi.nordpass.com     EXIT_DEAD
9970   10114  847   pi.nordpass.com     EXIT_DEAD
....                      ....            ....
```

Listing 2. Output of `psscan` showing a dead Password Manager.

The investigation begins by using the `pslist` plugin to enumerate active processes at the time of acquisition. As no suspicious activity is identified among the running processes, and after inspecting their virtual address spaces for artifacts relevant to the case, the analyst continues the investigation by using the `psscan` plugin. This plugin performs memory carving to recover `task_structs` that may belong to previously terminated processes.

As shown in Listing 2, `psscan` reveals the presence of the well-known multiplatform password and secret manager app NordPass (Nord Security, 2025) that had already terminated before the memory dump was taken. Consequently, it was no longer running on the system and was inaccessible via the device's user interface. Typically, password managers save stored secrets, such as contact details or credentials, in an encrypted form on the persistent storage, by using a master password, thus making the underlying data unrecoverable without that password.[3] However, using the `volshell` interactive shell, given the addresses of the `task_struct` of the terminated processes and leveraging the definitions of kernel data structures provided by the generated profile, the analyst can manually reconstruct the address space of the terminated password manager.[4]

The analyst then can perform a targeted search within the address space of the terminated process for strings indicative of encrypted contact information, such as *"email"* or *"address"*. The search is deliberately restricted to the password manager's address space to ensure that the recovered data could be confidently attributed to user interactions with the application. As illustrated in Listing 3, the analyst is ultimately able to recover plaintext evidence of sensitive data, including the suspect's contacts. To further validate our test, we searched for the same artifacts in the device's persistent storage without success, as this data was encrypted by the application prior to storage, making it inaccessible without the appropriate master password.

This toy example analysis highlight the new investigative scenarios made possible by LEMON and by generating profiles from BTF symbols, on modern Android devices but also on hardened Linux systems.

```
{
 "type": "identity",
 "value": {
    "name":"Pablo␣Escobar",
    "email":"pablo.escobar@car.tel",
    "phone_number":"+1-234-567-8901",
    "address1":"Calle_de_los_Limones-86",
    "city":"San␣Cristobal",
    "country":"Krakozhia",
    "last_used_at":"2025-05-10T22:17:45Z"
    }
}
```

Listing 3. Contact extracted from the terminated password manager.

5. Limitations and future extensions

While LEMON represents a significant step forward in expanding the range of devices that can be dumped, it also comes with some limitations. The tool relies on eBPF and on the presence of the `CONFIG_KALLSYMS_ALL` kernel option to resolve kernel symbols needed for memory dumping. Although this is not an issue for modern GKI kernels (for which both of them are mandatory) and most recent Linux distributions, this may prevent LEMON from functioning on older systems or custom kernels where the option is disabled. A similar consideration applies to the requirement that Kernel Lockdown `confidentiality` mode must not be enabled; nowadays it is disabled by default in kernels shipped with the major Linux distributions, and completely absent in GKI Android's security model. Moreover, while LEMON is designed to minimize the required system capabilities (`CAP_BPF` and `CAP_SYSLOG`), systems are generally not configured with such fine-grained capabilities; in practice, `root` privileges are required both to load eBPF programs and to disable `kptr_restrict`. This requirement may not always be feasible in certain forensic scenarios such Android phones for which exploits that permits to gain `root` privileges are not available, however this limitation is not LEMON related but it is shared by all existing volatile memory acquisition tools for Linux and Android.

Future work could also focus on reducing the overhead caused by communication between kernel and user space via maps, for example by leveraging eBPF RINGs, and by minimizing data copies within kernel space due to filesystem caches and socket buffers, thereby improving byte-level atomicity.

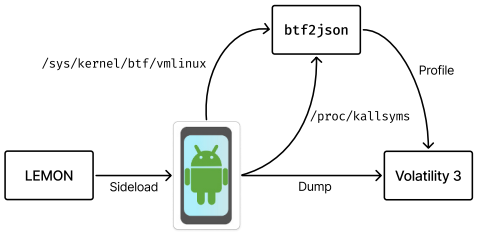

Fig. 5. Overview of the workflow for Android volatile memory acquisition and analysis using LEMON.

6. Related works

The memory forensics community has extensively studied the challenge of acquiring volatile memory from Android and Linux systems. A milestone in this field was established by Sylve et al. (2012), who introduced a kernel-based acquisition tool–initially known as "Droid Memory Dumper" (DMD), and later renamed to LiME. Complementing the kernel-level acquisition offered by LiME, Microsoft's AVML (Microsoft, 2023) provides a userland memory acquisition tool, obviating the need for on-target compilation or detailed knowledge of the system's OS distribution or kernel version.

A dynamic approach that retrieves device information at runtime and avoids compile-time kernel dependencies is used by AMExtractor (Yang et al., 2016) that exploits `/dev/kmem` on emulators or certain devices to execute code in kernel space. However, this approach is no longer applicable to production devices, as `/dev/kmem` is typically disabled for security reasons. An alternative approach, proposed by Yang et al. (2017), introduces AMD (Android Main Memory Dumping tool), which leverages the device's firmware update protocol to remotely acquire RAM without requiring root access or a reboot. However, this method requires, for each individual product from a vendor, a tedious manual reverse-engineering of the firmware update protocol to search for potential, though not guaranteed, vulnerabilities or hidden commands that would allow memory dumping.

More recent and generalized approaches were presented in a series of works by Tobias Latzo et al. The authors explored different low-level memory acquisition techniques that operate beneath the operating system, thus evading the restrictions imposed by features such as Secure Boot. They proposed three distinct techniques based on UEFI (Latzo et al., 2021a), Baseboard Management Controllers (BMCs) (Latzo et al.,

[3] Our analysis confirmed that the password manager zero-out the master password from the volatile memory when the app is closed.

[4] At the time we write, Volatility 3 does not provide a plugin to automatically extract the virtual address space of a Linux terminated process.

2020), and Intel Direct Connect Interface (DCI) (Latzo et al., 2021b). These methods are particularly valuable when kernel-level tools are rendered ineffective by security constraints. However, they typically require pre-incident setup or specialized hardware configurations, which are not commonly available during forensic analysis of third-party systems.

Another line of research adopts live response techniques, which do not acquire RAM directly but instead extract memory artifacts via debugging interfaces or app-level APIs. For instance, on Android, memgrab (Thing et al., 2010) uses `/proc/<pid>/maps` and `ptrace` to trace a process and dump its address space. This process-level acquisition tool can extract user-space memory without rebooting the device, but it typically requires the target app to be explicitly marked as debuggable and fails to capture kernel-level evidences.

7. Conclusions

Kickstarted by the "Memory Analysis" challenge proposed at the DFRWS conference in 2005, the field of memory forensics rapidly became one of the most active areas of research in digital forensics and saw tremendous advancements over the following two decades. Unfortunately, this trend has gradually reversed: this shortfall is especially pronounced in Linux- and Android-based environments, where features such as Secure Boot, Kernel Lockdown, and strict module signing policies have rendered traditional acquisition tools ineffective or unusable. As a result, volatile memory acquisition, once central to forensic investigations, is now overlooked due to technical barriers and the misconception that persistent storage analysis alone suffices.

Motivated by these challenges, in this paper we introduced LEMON, the first volatile memory acquisition tool based on eBPF. By leveraging this powerful and portable in-kernel virtual machine, LEMON overcomes the limitations of kernel module-based approaches, providing a universal and robust solution for acquiring memory from modern hardened Linux systems and GKI Android devices. Its CO-RE and non-CO-RE compilation modes enable broad compatibility without requiring kernel headers or signed binaries.

Our experimental results confirm that LEMON matches the performance, byte-level atomicity, and reliability of existing tools, while extending memory acquisition to systems that were previously unsupported. Moreover, by demonstrating real-world acquisitions on modern Android devices and recovering artifacts from terminated applications, unavailable through disk forensics, we highlight how memory acquisition remains not only relevant, but necessary in Android environment.

We hope this work will not only offer the community a practical and effective tool, but also help reinvigorate research in this important and too-often-neglected area of digital forensics.

Code availability and acknowledgments

In the spirit of open-science and to enable the memory forensics community to use and further develop it, LEMON (Oliveri et al., 2025b), modified `btf2json` tool (Oliveri et al., 2025a) and the script (Oliveri et al., 2025c) used to compare the dumps are released as an open-source project.

This work was supported by the French National Research Agency (ANR) under the Plan France 2030 bearing the reference ANR-22-PECY-0007 and ANR-22-PECY-0009.

References

Campbell, W., 2013. Volatile Memory Acquisition Tools – a Comparison across Taint and Correctness.
Case, A., Richard III, G.G., 2017. Memory forensics: the path forward. Digit. Invest. 20, 23–33.
Cilium, 2025. Cilium: ebpf-powered Networking, Security, and Observability. URL: https://cilium.io/.
Facebook, 2025. Katran: a High Performance Layer 4 Load Balancer. URL: https://github.com/facebookincubator/katran.
Falco, 2025. Falco: Cloud-Native Runtime Security. URL: https://falco.org/.
Garrett, M., 2020. Linux Kernel Lockdown, Integrity, and Confidentiality. URL: https://mjg59.dreamwidth.org/55105.html.
Google, 2025a. Android Kernel Configs. URL: https://android.googlesource.com/kernel/configs.
Google, 2025b. Generic Kernel Image (Gki). URL: https://source.android.com/docs/core/architecture/kernel/generic-kernel-image.
Google, 2025c. Kernel Abi Monitor. URL: https://source.android.com/docs/core/architecture/kernel/abi-monitor.
Halderman, J.A., Schoen, S.D., Heninger, N., Clarkson, W., Paul, W., Calandrino, J.A., Feldman, A.J., Appelbaum, J., Felten, E.W., 2009. Lest we remember: cold-boot attacks on encryption keys. Commun. ACM 52, 91–98.
Hubble, 2025. Hubble: ebpf-based Observability for Kubernetes. URL: https://github.com/cilium/hubble.
Isovalent, 2025. Tetragon: ebpf-based Security Observability and Runtime Enforcement. URL: https://github.com/cilium/tetragon.
Kornblum, J.D., 2007. Using every part of the buffalo in windows memory analysis. Digit. Invest. 4, 24–29.
Latzo, T., Brost, J., Freiling, F., 2020. BMCLeech: introducing stealthy memory forensics to BMC. Forensic Sci. Int.: Digit. Invest. 32. https://doi.org/10.1016/j.fsidi.2020.300919 cRIS-Team WoS Importer:2020-06-23.
Latzo, T., Hantke, F., Kotschi, L., Freiling, F., 2021a. Bringing forensic readiness to modern computer firmware. In: Proceedings of the Digital Forensics Reesearch Conference Europe (DFRWS EU) 2021.
Latzo, T., Schulze, M., Freiling, F., 2021b. Leveraging intel DCI for memory forensics. In: Proceedings of the Digital Forensics Research Conference USA (DFRWS US).
Linux Foundation, 2025. Uprobe. URL: https://docs.kernel.org/trace/uprobetracer.html.
Merbridge, 2025. Merbridge: Accelerate Service Mesh with Ebpf. URL: https://github.com/merbridge/merbridge.
Microsoft, 2023. Avml (Acquire Volatile Memory for Linux). URL: https://github.com/microsoft/avml.
Moser, A., Cohen, M.I., 2013. Hunting in the enterprise: forensic triage and incident response. Digit. Invest. 10, 89–98.
Müller, T., Spreitzenbarth, M., 2013. Frost. In: Jacobson, M., Locasto, M., Mohassel, P., Safavi-Naini, R. (Eds.), Applied Cryptography and Network Security. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 373–388.
Nord Security, 2025. Nordpass. URL: https://nordpass.com/.
Obst, V., 2024. btf2json - Leverage the Bpf Type Format (Btf) to Generate Volatility 3 Profiles. URL: https://github.com/vobst/btf2json.
Oliveri, A., Balzarotti, D., 2025. A comprehensive quantification of inconsistencies in memory dumps. In: Symposium on Research in Attacks, Intrusion, and Defenses (RAID). IEEE.
Oliveri, A., Cavenati, M., De Rosa, S., Lakshmi Narasimhan, S., Balzarotti, D., 2025a. btf2json Tool for Android. URL: https://github.com/eurecom-s3/lemon-btf2json/.
Oliveri, A., Cavenati, M., De Rosa, S., Lakshmi Narasimhan, S., Balzarotti, D., 2025b. LEMON. URL: https://github.com/eurecom-s3/lemon/.
Oliveri, A., Cavenati, M., De Rosa, S., Lakshmi Narasimhan, S., Balzarotti, D., 2025c. LEMON-experiments. URL: https://github.com/eurecom-s3/lemon-experiments/.
Ottmann, J., Breitinger, F., Freiling, F., 2023. An experimental assessment of inconsistencies in memory forensics. ACM Transactions on Privacy and Security 27, 1–29.
Pagani, F., Fedorov, O., Balzarotti, D., 2019. Introducing the temporal dimension to memory forensics. ACM Transactions on Privacy and Security (TOPS) 22, 1–21.
Rzepka, L., Ottmann, J., Freiling, F., Baier, H., 2024. Causal inconsistencies are normal in windows memory dumps (too). Digital Threats: Research and Practice 5, 1–20.
Rzepka, L., Ottmann, J., Stoykova, R., Freiling, F., Baier, H., 2025. A scenario-based quality assessment of memory acquisition tools and its investigative implications. Forensic Sci. Int.: Digit. Invest. 52, 301868.
Sarkar, P., 2024. Volatility 3 - arm64. URL: https://github.com/Panchajanya1999/volatility3-arm64.
Sun, S.T., Cuadros, A., Beznosov, K., 2015. Android rooting: methods, detection, and evasion. In: Proceedings of the 5th Annual ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices, pp. 3–14.
Sylve, J., 2012. Lime-linux memory extractor. In: Proceedings of the 7th ShmooCon Conference.
Sylve, J., Case, A., Marziale, L., Richard, G.G., 2012. Acquisition and analysis of volatile memory from android devices. Digit. Invest. 8, 175–184.
Thing, V.L., Ng, K.Y., Chang, E.C., 2010. Live memory forensics of mobile phones. Digit. Invest. 7, S74–S82. https://doi.org/10.1016/j.diin.2010.05.010 the Proceedings of the Tenth Annual DFRWS Conference. https://www.sciencedirect.com/science/article/pii/S174228761000037X.
Tracee, 2025. Tracee: Runtime Security and Forensics Using Ebpf. URL: https://github.com/aquasecurity/tracee.
Vieira, M.A.M., Castanho, M.S., Pacífico, R.D.G., Santos, E.R.S., Júnior, E.P.M.C., Vieira, L.F.M., 2020. Fast packet processing with ebpf and xdp: concepts, code, challenges, and applications. ACM Comput. Surv. 53. https://doi.org/10.1145/3371038. URL:

Walker, A., 2017. Volatility framework: volatile memory artifact extraction utility framework. URL. https://volatilityfoundation.org/.
Yang, H., Zhuge, J., Liu, H., Liu, W., 2016. A tool for volatile memory acquisition from android devices. In: Advances in Digital Forensics XII: 12Th IFIP WG 11.9 International Conference, New Delhi, January 4-6, 2016, Revised Selected Papers 12. Springer, pp. 365–378.
Yang, S.J., Choi, J.H., Kim, K.B., Bhatia, R., Saltaformaggio, B., Xu, D., 2017. Live acquisition of main memory data from android smartphones and smartwatches. Digit. Invest. 23, 50–62.