Semester Project Report - RandCompile: Removing Forensic Gadgets from the Linux Kernel to Combat its Analysis

Tamar Abelson, Mackenzie Eng, Karwai Kang, and David Xiao

Hunter College

{Tamar.Abelson22 ,Mackenzie.Eng37,
Karwai.Kang76,David.Xiao67}@myhunter.cuny.edu

ABSTRACT. The idea of this project is to improve Linux kernel security by making it harder for attackers to receive useful information from compiled kernel binaries [3]. Forensic gadgets are predictable code patterns, memory layouts, or instruction sequences that forensic tools use to analyze a system's memory or execution behavior [7]. These forensic methods can be reduced by using RandCompile, a new approach for kernel randomization at compile time [3]. RandCompile disrupts forensic analysis by introducing randomized compilation techniques, removing predictable forensic gadgets while keeping kernel functionality. This project's goal is to explore and implement RandCompile as a defense mechanism against forensic analysis techniques that are used in cybercrime investigations [3]. By modifying the compilation process of the Linux kernel [6], we will introduce controlled variations that prevent deterministic forensic methods while maintaining performance and stability. In this paper, we expand the preliminary midterm report of RandCompile into a comprehensive study.

Keywords: RandCompile, Forensic Gadgets, Linux Kernel

1 BACKGROUND

1.1 Introduction

The Linux kernel is the core component of the Linux operating system and it is responsible for managing hardware resources, process scheduling, and system security [6]. Since it operates at the lowest level of the system, it has complete control over all system processes and user interactions. This makes it a critical target for security defenders as well as attackers, as it is widely used in cloud infrastructure, embedded systems, mobile devices, etc.

Forensic analysis helps to detect threats while also providing attackers insights of the system behavior. Attackers use forensic tools such as Volatility [1] and Rekall [5] to extract information about memory structures, kernel objects, and running processes.

Kernel Address Space Layout Randomization (KASLR) is an example of one existing kernel security mechanism that focuses on preventing unauthorized access to memory [4]. KASLR was created to make it more difficult for attackers to use memory vulnerabilities in the Linux kernel. It works by randomizing the memory addresses where the kernel and its components are loaded at run time. While KASLR makes direct exploitation more difficult, it does not eliminate forensic gadgets, which attackers can use to locate key kernel structures even in randomized memory layouts.

The goal of this project is to disrupt forensic analysis methods by introducing randomness at the compilation stage—an approach that is called RandCompile [3]. By implementing RandCompile, we aim to challenge traditional forensic analysis and improve Linux kernel security; by making it less predictable, harder to analyze, and ultimately more resilient against forensic.

RandCompile's approach removes or randomizes predictable forensic artifacts in compiled Linux kernels, thereby reducing their forensic footprint. Our work will be implemented and tested within a controlled virtualized environment to measure its effectiveness against known forensic techniques [3].

1.2 Forensic Analysis & Linux Kernel Security

Memory forensic frameworks like Volatility and Rekall traditionally require a profile of the target kernel (structure definitions, symbol addresses) to interpret memory dumps. Newer tools eliminate the need for pre-built profiles by deriving kernel layout from the dump itself. AutoProfile (ACM TOPS 2022) and LogicMem (NDSS 2022) use program analysis to automatically infer kernel structure layouts from a snapshot. Katana (RAID 2022) extends this approach with cross-architecture support, using intermediate representations to analyze how kernel code accesses data (Katana: Robust, Automated, Binary-Only Forensic Analysis of Linux Memory Snapshots) (Katana: Robust, Automated, Binary-Only Forensic Analysis of Linux Memory Snapshots). These tools can enumerate processes, modules, open files, etc., on a captured memory image by identifying tell-tale patterns in memory that correspond to kernel data structures.

Forensic Gadgets: Franzen et al. (ACSAC 2023) define forensic gadgets as the predictable memory patterns or code artifacts that such forensic tools exploit to reverse-engineer kernel state [3]. Several common leverage points have been identified [3]:

FG1 – Special comm values: The task_struct (process descriptor) embeds a fixed 16-byte name field (comm). Known init process names (e.g. "swapper/0", "init") appear at predictable locations. Forensic tools search for these hard-coded strings; finding "swapper/0" in memory reveals the address of the first process (init_task) [3].

FG2 – Symbol tables: Kernel symbol information (e.g. the kallsyms table of addresses) can leak locations of important globals. For example, the symbol init_task gives away the address of the task list head. Tools that can read the in-memory symbol table or use exported symbols can bootstrap their analysis [3].

FG3 – ABI calling conventions: On x86-64, the System V ABI dictates a fixed register order for function arguments (e.g. 1st argument in rdi, 2nd in rsi, etc.). This means a compiler will generate consistent instruction

sequences to load function parameters. Forensic analyzers like Katana use these offset-revealing instructions to match assembly to source code – e.g. if a function always loads a structure field into the first argument register, the offset used in that instruction can be mapped to a specific field in that struct [3].

FG4 – Stable field order: Even when structure offsets change (across kernel versions or configs), the relative order of fields in a struct is usually constant. Tools like LogicMem and AutoProfile use this rule: if they identify one field in a struct, they can infer which fields come before or after it in memory [3]. (Note: Linux has an optional feature called Structure Layout Randomization (SLR) that randomizes field order for certain sensitive structs. However, it was only applied to a handful of structures and thus many remained with deterministic layouts [3]).

FG5 – Pointer graph patterns: Kernel data structures often form distinctive pointer chains. For example, all task_struct instances are linked in a circular doubly-linked list (tasks.next/prev). A forensic tool can detect a cyclic list in memory by following pointers (resolvable addresses that eventually loop back) [3]. If such a cycle is found, it strongly indicates the global task list. Similarly, pointers that reference known structure layouts can be tested to confirm their type.

These forensic gadgets allow an analyst to reconstruct kernel internals even if direct symbols or addresses are hidden (as KASLR does). Existing mitigations only partly address them: KASLR randomizes base addresses but not relative layouts; SLR (since Linux 4.13) randomizes field order for only a few structs (those historically related to exploits) [3], leaving many structures still deterministic. Hardware-based full memory encryption like AMD SEV can theoretically block introspection by making memory opaque to the hypervisor, but SEV is not universally available and has been broken by various attacks in recent years [3]. There is a clear gap for a software-based defense that broadly removes these forensic artifacts without needing special hardware.

2 Implementation

RandCompile consists of a patch set for the Linux kernel (in our case based on Linux 5.15 LTS) and a custom GCC compiler plugin[1]. The modifications are applied at compile time, producing a kernel binary that is functionally equivalent to the original but with various randomized or obfuscated aspects. Crucially, RandCompile's transformations are semantics-preserving: they do not change how the kernel behaves, only how it is represented in memory and machine code. This ensures that a RandCompile-hardened kernel can run normally on a system without requiring special runtime support. However, by altering certain low-level details, it frustrates forensic analysis and also potentially hinders some attacker exploits that assume a standard kernel memory layout[1].

The RandCompile framework implements several defenses, each targeting one or more of the forensic gadgets described earlier. Table 1 provides an overview of the main features and which gadget(s) they mitigate. We explain each defense in detail below.

Table 1: RandCompile Features and Targeted Forensic Gadgets

RandCompile Defense	Forensic Gadgets Mitigated
String Literal Encryption (e.g. "swapper/0")	FG1 (Known constant comm values) – prevents identification of init task by name.
Pointer Encryption (task list pointers)	FG3 (Linked-list pointer patterns) – prevents traversal of task list by hiding pointer values.
printk Format Externalization	FG1 (strings in logs) and general – hides constant kernel log strings, impeding log reconstruction.

Parameter Order Randomization (ABI shuffle)	FG4 (offset assumptions in code) – breaks assumption that register order corresponds to structure field order.
Bogus Parameter Insertion (with dummy accesses)	FG4 (offset inference) – adds noise by fake memory accesses, mislead analysis of code and data.
Selective Structure Padding (via SLR) [optional]	FG4 (if any remaining layout invariants) – complements existing structure layout randomization.

(Note: The last item refers to Linux's built-in SLR which RandCompile can complement; most of our focus is on the first five defenses implemented by RandCompile.)

Internally, RandCompile uses the compiler plugin to transform code on the fly during compilation. It also applies minor source-level or assembly-level patches for cases where the compiler alone cannot enforce the change (for instance, altering certain global data definitions). A fundamental challenge is to maintain kernel stability: randomizing the ABI or encrypting pointers can break assumptions in some kernel subsystems or modules. The RandCompile implementation includes careful blacklists and exceptions: certain functions that interact with assembly code or modules are not randomized to avoid breaking the kernel[1]. According to the authors, this approach preserved full functionality in their tests - the system booted and ran normally with all features enabled. We similarly found in our implementation that the hardened kernel operates correctly under standard workloads.

2.1 Defense Mechanism

String Literal Encryption (Process Names)

To defeat FG1 – the use of known process name strings like "swapper/0" – RandCompile encrypts or otherwise conceals these constants in the compiled kernel image. Specifically, the string "swapper/0" (and similar known values like "init" for the first user process) are no longer stored in plaintext in the kernel's data section. One approach (as described by Franzen et al.)[1] is to replace such literals with an encrypted version and decrypt them only when needed, or to generate them at runtime rather than hard-coding them. By doing so, a forensic tool scanning memory for "swapper/0" will not find it, effectively blinding techniques that rely on this signature.

In our implementation and evaluation, the encryption of "swapper/0" proved to be extremely effective: it stopped multiple forensic tools outright. According to the original study, hiding the idle task's name prevented LogicMem, the TrustZone-based rootkit by Marth et al., and HyperLink from operating at all. These tools presumably failed to locate the initial task and could not proceed with process enumeration. Another tool, FOSSIL (a 2023 OS-agnostic memory forensic tool), had its analysis performance degraded when string gadgets were removed. In our tests, after applying RandCompile, the memory snapshot contained no readable "swapper/0" string – confirming the success of this defense. The kernel still knows the idle task's name internally (it can be reconstructed or stored encrypted), but an external dump won't reveal it.

It's worth noting that encrypting such strings has negligible runtime cost and does not affect kernel logic (the name is rarely used in logic, mostly for display). Thus, this defense is a low-hanging fruit that yields high impact. We observed that even if this were the only defense enabled, it would significantly impair straightforward process-finding techniques. However, adversaries might try alternative signatures (like task struct size or other constants), so string encryption works best in combination with the pointer obfuscation described next.

Pointer Encryption (Task List Pointers)

RandCompile also targets FG3 by encrypting crucial pointers in kernel data structures. The primary focus is on the task_struct.tasks pointers (the next and prev links in the process list). These two pointers form the circular linked list connecting all tasks. By encrypting or masking these pointers in memory, RandCompile ensures that a forensic tool cannot follow them to iterate over tasks – the pointers no longer hold valid addresses of other tasks (at least not in plain form). In our implementation, we applied a simple XOR-based encryption with a secret key for the tasks.next and tasks.prev fields whenever they are written or read. The decryption/encryption key can be stored in the kernel (or even derived per boot, making it unknown to an offline analyst).

This defense breaks the classic technique of finding tasks via pointer patterns. If a forensic tool tries to follow the chain of tasks.next, it will dereference gibberish (the encrypted value), likely leading to an invalid address or a crash in the analysis. As reported by Franzen et al.[1], pointer encryption degrades the analysis capabilities of tools like LogicMem, HyperLink, and the TrustZone rootkit even further. In combination with string encryption, it leaves those tools with few leads to locate the task list. Pointer encryption can be extended beyond the task list: RandCompile's authors suggest future work to encrypt other kernel pointers that are frequently targeted (for example, pointers in the mm_struct or file descriptor tables). However, one must be careful to only encrypt pointers that are not used outside the kernel or across privilege boundaries, to avoid breaking functionality.

The performance impact of pointer encryption is minimal. An XOR on pointer read/write is trivial for the CPU, and these operations are not in the hottest code paths (modifying a task list pointer happens when processes start or stop, not every system call). Our microbenchmark results (see Results section) indicated no measurable overhead attributable specifically to pointer encryption alone. We also did not encounter stability issues – the kernel itself "knows" to decrypt/encrypt when following the list, so everything works internally. From the forensic perspective, this measure forces an analyst to resort to heavy-weight strategies (perhaps guessing the key or observing pointer usage via code… which leads to the next defense).

printk Format String Externalization

Linux kernel's printk function is used for logging, and format strings used in printk calls (like printk("Initializing device %s\n", dev_name)) are typically stored as static strings in the kernel's read-only data section. These strings can constitute forensic gadgets as well: by scanning for certain log messages or keywords, an analyst might locate the kernel's log buffer or infer the occurrence of certain events. Moreover, the entire kernel log (dmesg ring buffer) can often be retrieved by forensic tools if they find its location, revealing a wealth of information about the system's runtime state.

RandCompile introduces a defense mechanism wherein printk format strings are externalized or obfuscated so that they are not readily available in plaintext in memory. The idea is to prevent an offline analysis from simply reading the kernel's log messages or using them as signatures. According to the implementation by Franzen et al., printk formats are handled specially by the compiler plugin[1]. One approach is to remove the actual format strings from the binary and store them in a separate section or encode them, such that when printk is called, it uses an indirect reference or an identifier to fetch the real string. In a memory dump, the ring buffer would contain only an encoded representation of messages, which would appear as gibberish to an analyst without the means to decode or map them back.

A side effect of some of RandCompile's strategies (particularly bogus parameters, discussed next) is that it can intentionally desynchronize the printk format strings from the actual arguments, resulting in meaningless output if an analyst tries to read it directly. For instance, RandCompile may add dummy parameters to printk calls; the kernel's console output might then show format specifiers with no corresponding meaningful data, effectively making the log hard to read unless post-processed. However, the legitimate user is not intended to lose the log. The authors implement a reintegration tool that can reconstruct the proper kernel log given the hardened kernel's knowledge. They describe a process where the previously introduced dummy parameters (marked in a special way) are stripped out, and the real format strings are reinserted, yielding a correct dmesg output for the user. This means RandCompile

provides anti-forensic benefits without permanently sacrificing debuggability – authorized users can still retrieve logs, but an unauthorized dump reader cannot.

In our project setup, we enabled the printk format externalization feature of RandCompile and observed that an offline dump of the kernel log was indeed not intelligible. For example, where we would normally see a message like "[0.123456] Linux version 5.15.63 ..." in the log, the hardened kernel's memory contained an altered string. When we attempted to use a forensic tool's "retrieve dmesg" function on the hardened kernel, it failed to produce the correct output (in one case, no output at all). This confirms the effectiveness of the approach: the kernel's log is no longer a straightforward forensic artifact. There is a balance to strike here – by hiding format strings, we must ensure not to break any kernel functionality that relies on them (some subsystems might search their own logs in memory). RandCompile's modifications appeared to handle this gracefully.

ABI Randomization: Parameter Order Randomization

One of RandCompile's core techniques is randomizing the function call ABI within the kernel, which addresses FG4 by breaking the assumption that certain registers or stack positions correspond to certain structure fields. Modern CPU architectures (like x86_64) pass function arguments in a defined order (e.g., first argument in RDI, second in RSI, etc., for the System V ABI on x86_64). Compiler-generated code thus consistently uses, say, RDI to hold the pointer to a structure and then accesses fixed offsets from it. A forensic tool analyzing many functions can assume that if a particular offset (e.g. 0x100) is always accessed from the first argument register, then offset 0x100 likely is an important field in the structure passed as the first argument.

RandCompile introduces randomness in how arguments are passed to functions within the kernel (this does not affect external interfaces like system calls, only internal kernel function calls). The compiler plugin can randomly permute the order of arguments for each function and adjust all call sites accordingly. For example, if a function originally is defined as func(struct task_struct *t, int flag), normally t would always be in the first register and flag in the second. With RandCompile, on one compilation t might end up in the second register and flag in the first, with all calls to func changed to swap the parameters accordingly. Another function might get a completely different random ordering of its parameters. This means that from an external viewpoint (without knowing the randomization seed or scheme), there is no consistent ABI across functions – the order and placement of arguments is unique to that build[1].

This parameter order randomization thwarts techniques that rely on correlating register usage with structure fields. Katana and similar tools often assume that, for instance, the first argument to certain routine is the data structure of interest (like the task list or a file struct) and that offsets accessed in that routine belong to that structure. If the calling convention is randomized, then the tool's analysis heuristics either fail or must solve a much more complex permutation guessing game. Essentially, every function becomes a "snowflake" with its own calling convention.

However, there is a challenge: if a function has fewer than the maximum number of register arguments (e.g., on x86_64 the first 6 arguments are in registers), then randomizing within its smaller set doesn't add much entropy, especially if it's just one or two arguments. Also, functions with very few parameters might always pass the important pointer as the sole argument, which doesn't change. To improve this, RandCompile uses the next technique: adding bogus parameters to increase the parameter count.

It's important that ABI randomization not break kernel interfaces where the order is assumed. RandCompile therefore avoids randomizing any function that is called from assembly or that needs a specific prototype (for example, interrupt handlers or module entry points). Those are left in their standard form. The randomization is applied mostly to internal static functions and other flexible points. The authors reported no stability issues from this in their experiments. In our usage, we similarly did not observe any crashes or misbehavior attributable to parameter reordering – the kernel operated normally, indicating that the blacklisting of sensitive functions was effective.

Bogus Parameter Insertion (with Dummy Memory Accesses)

To maximize the effectiveness of ABI randomization and confuse code analysis (FG4), RandCompile adds bogus parameters to functions that have a low number of arguments. The idea is to pad every function's parameter list up

to a certain length (up to 6 on x86_64, since 6 registers are available for integer arguments) by inserting dummy arguments. These extra arguments are not used by the original logic of the function, but RandCompile makes them appear used by inserting artificial memory accesses or operations involving them. This prevents the compiler from optimizing them away and also makes the compiled code a lot less transparent to an analyzer[1].

For example, consider a function int do_fork(struct task_struct *parent). Normally it has one parameter. RandCompile might add five more integer parameters to it, making it int do_fork(struct task_struct *parent, long p1, long p2, long p3, long p4, long p5). At each call site, additional dummy values are passed (which can be anything or derived from existing data). Inside do_fork, RandCompile injects some fake uses, say it might read an integer from parent at a pseudorandom offset and store it to p1 (which is meaningless, but ensures p1 is "used"), or perform a calculation involving p2 etc. These dummy memory accesses are carefully chosen: they might point to valid addresses within structures to seem plausible. The effect is that any static analysis of the code sees multiple memory accesses and cannot easily distinguish which ones correspond to real fields and which are bogus. The authors mention generating a set of plausible alternative offsets for a field and using them in bogus parameters, to confuse statistical analysis.

This strategy raises the uncertainty for tools like LogicMem or Katana. Instead of seeing a clear pattern of "field X is always accessed at offset Y", the tools now see many accesses at various offsets, some of which are fake. The analysis might either attempt to consider all of them (leading to incorrect reconstructions) or fail to converge on a correct profile. In Franzen et al.'s evaluation, adding bogus parameters with artificial accesses caused a noticeable drop in the success of structure reconstruction for more complex analyses like task listing and file listing. The decrease was small but significant – enough to cause automated analysis to falter or require manual intervention. The combination of parameter reordering (random ABI) with bogus parameters is referred to as the full RandCompile ABI defense.

One must be mindful of the overhead: adding dummy parameters increases register pressure and code size. We chose not to exceed 6 parameters so that on x86_64 all dummy parameters still go in registers (not spilling to stack). This avoids additional memory overhead for each call. Even so, the code does extra dummy loads, which could impact performance slightly. We evaluate this in the Results section. RandCompile's design allows disabling the bogus parameter feature if a user wants to trade some security for performance; the "no bogus" variant of RandCompile uses only reordering without adding new arguments.

Maintaining Compatibility and Security

RandCompile's transformations were implemented with attention to compatibility. As mentioned, certain functions are blacklisted from ABI changes to avoid breaking low-level mechanisms. Kernel modules pose an interesting challenge: a module compiled in the usual way expects the standard ABI and data structures. A RandCompile-hardened kernel thus would not be ABI-compatible with a module compiled without RandCompile. In practice, this means that modules (especially out-of-tree proprietary modules) might not work unless they are compiled with the same RandCompile plugin and randomization seed. In our experiments, we focused on the core kernel and did not extensively test loadable modules; however, this is noted as a general limitation – RandCompile is most straightforward to use in environments where the entire kernel and its modules can be built together (e.g., a custom kernel for a cloud image or device firmware). As a mitigation, one could disable RandCompile for the module interface or provide a "bridge" for known modules, but that might open gadgets. This trade-off will be discussed later.

Another point is debuggability: a kernel built with RandCompile is harder to debug using standard tools. Since it changes calling conventions and data layouts (within limits), tools like crash analyzers or kgdb that assume a normal kernel may not work properly. In our usage, we indeed found that GDB scripts expecting to find tasks via init_task symbol or follow tasks list did not work on the hardened kernel without custom adaptation. This is acceptable in a production environment where security is prioritized, but developers might need to turn off RandCompile for debugging sessions.

Despite these caveats, an important upside is that RandCompile's approach has very low overhead relative to many security features. The authors report under 5% performance penalty on average, which we corroborate with our benchmarks. This overhead is smaller than some widely used defenses like KASLR or certain compiler sanitizers, making it a compelling option for hardening. Furthermore, by eliminating forensic gadgets, RandCompile not only thwarts forensic analysis but also can hinder attackers' exploits that rely on kernel knowledge. For example, an exploit that needs to locate the system call table or iterate through tasks would face similar difficulties as a forensic tool would. Encrypting pointers is a known mitigation used in other systems (Windows uses pointer obfuscation in its pools, glibc uses pointer encryption for security cookies). RandCompile thus contributes to general kernel security, not just forensic evasion.

In summary, RandCompile implements a multi-faceted compile-time transformation of the Linux kernel. These transformations (string encryption, pointer encryption, format string externalization, ABI randomization, bogus parameters) synergize to remove the static patterns (forensic gadgets) that memory analysis tools depend on. Next, we describe how we set up an environment to test these features and measure their impact.

2.2 Experimental Setup

To evaluate RandCompile's effectiveness and overhead, we prepared a testing environment using a QEMU/KVM virtual machine running a 64-bit Linux kernel (version 5.15.63). We chose this kernel version to align with RandCompile's original implementation (which was built for Linux 5.15 LTS) and to ensure that known forensic tools support it. All kernels were built for the x86_64 architecture using GCC 11 with the RandCompile compiler plugin when applicable. We used a Debian-based minimal userland (via Buildroot) and configured QEMU to facilitate introspection tests (e.g., enabled easy snapshotting and GDB access).

Kernel Variants: We compiled a series of kernel variants to compare the presence or absence of RandCompile defenses. These variants are listed in Table 2. The baseline is a stock Linux 5.15.63 with no special modifications ("base"). We also compiled a baseline with debug symbols and FTRACE enabled ("base_ftrace") to simulate a kernel with typical instrumentation (this is mainly for performance comparison). The RandCompile-hardened kernels include: nobogus, bogusargs, bogusmem, and forensic_hardening. The nobogus kernel had the RandCompile patch set and plugin enabled, but we disabled the bogus parameter insertion feature – this variant uses only pointer/string encryption and basic ABI shuffling (it represents an intermediate hardening without the heavy fake-parameter defense). The bogusargs kernel enabled the ABI randomization and bogus parameters (function-level defenses) but not pointer encryption or printk externalization – effectively focusing on the function call obfuscations. Conversely, bogusmem enabled the memory content defenses (pointer encryption, string encryption, printk obfuscation) but did not insert extra function parameters – focusing on data structure level defenses. Finally, forensic_hardening turned on all RandCompile features (this is the full-strength configuration). All hardened kernels were built with the same RandCompile plugin and a fixed randomization seed to ensure reproducibility across runs (in a real deployment, the seed could be varied per build for uniqueness).

Table 2: Linux Kernel Variants Used in Evaluation.

Variant Name	Description
base	Vanilla Linux 5.15.63 (no RandCompile, no special debug).
base_ftrace	Baseline with FTRACE and debugging symbols enabled (to measure baseline overhead of instrumentation).

nobogus	RandCompile applied, but no bogus parameters (only pointer/string encryption + ABI reordering). Essentially, RandCompile with bogus param feature off.
bogusargs	RandCompile with function call obfuscation: ABI randomization + bogus parameters, but pointer/string encryption features disabled. (Focus on argument-level defenses).
bogusmem	RandCompile with memory obfuscation: pointer encryption and printk string externalization enabled, but no bogus parameters (no ABI shuffle). (Focus on data-level defenses).
forensic_hardening	Full RandCompile: all protections enabled (string encryption, pointer encryption, ABI randomization, bogus parameters, printk externalization).

Each kernel was built with the same configuration (apart from the variations above). We ensured that all kernels boot successfully in QEMU and run a basic init process.

Forensic Tools & Procedures: We tested two state-of-the-art memory forensic approaches: (1) offline analysis using Katana, and (2) live introspection using HyperLink. Additionally, we attempted to use LogicMem for offline analysis, although as we discuss, it did not function even on the baseline in our environment, which is an interesting data point in itself.

- Katana (offline memory analysis)[2]: Katana requires a memory dump of the target system in an ELF core format. To obtain this, we used QEMU's ability to produce a core dump of the VM's memory. For each kernel variant, we booted the VM, allowed it to reach a stable state (init process running, a few seconds of uptime), and then paused the VM to dump memory. We then ran Katana's analysis on the dump. We were particularly interested in Katana's core analyses: listing processes, listing loaded kernel modules, listing open files, and extracting the kernel's dmesg log. These correspond to the typical outputs a forensic investigator might want. We recorded whether each analysis succeeded or failed, and if it succeeded, how many structure members or entries it reconstructed (as a measure of completeness). Katana, being a robust tool, might sometimes partially succeed (e.g., find some processes but not all, or list module names but not details) – so we captured the level of success.
- HyperLink (live introspection)[4]: HyperLink is a framework that can attach to a running VM via VMI (using a hypervisor or debugging interface) and extract information like a process list without needing kernel symbols. We used an approach similar to that described by RandCompile's authors: launching QEMU with debugging enabled (-s -S options to wait and allow GDB attachments). We then used a GDB session attached to the VM to assist HyperLink. Specifically, HyperLink's hyperlink-ps tool can list processes if given the address of init_task. On an unhardened kernel, one can retrieve the init_task address via symbols or by scanning for "swapper/0". In our test, for the base kernel, we simply looked up init_task through GDB (since debug symbols were available in base_ftrace variant) and fed it to HyperLink. For hardened kernels, this direct method would not work (symbols removed or pointers encrypted), so we effectively simulate an adversary who does not have an easy way to get init_task – they would have to find it via analysis (which should fail). In practice, our

HyperLink test consisted of trying to perform a process listing on each running VM, and observing whether it succeeds (prints the processes) or fails (e.g., cannot follow the task list). We used GDB to verify what happens when following the tasks pointers on hardened vs base kernels (to confirm pointer encryption is working). The HyperLink workflow thus demonstrates real-time forensic analysis attempt on a live system.

- LogicMem (offline)[3]: We attempted to use the LogicMem prototype by Qi et al. on our memory dumps. LogicMem requires certain inputs like the physical offset of the kernel in memory, and possibly the page table base (CR3) and some initial structure signatures (the authors provided some scripts for mainstream distros). We provided it with the needed information for our base kernel dump (since we knew the layout from QEMU). However, in our environment, LogicMem failed to produce meaningful results even on the base kernel. It appears LogicMem has a very strict expectation of certain structure layouts (task_struct, mm_struct, etc.), and if anything differs (even different compile-time options), it struggles. In fact, this aligns with the RandCompile paper's note that they too failed to get LogicMem to extract structures on their test systems. We suspect that the kernel configuration and version we used was not fully supported by the LogicMem release. Additionally, since our kernels have Structure Layout Randomization enabled by default (Linux 5.15 has partial SLR for some structures), LogicMem's logic rules might not match exactly, causing it to misfire. We include this in our results to illustrate that some forensic frameworks are brittle – and RandCompile's changes (like field padding changes) further invalidate LogicMem's assumptions. In summary, LogicMem did not yield any process or module listings even on the baseline, so we cannot directly quantify RandCompile's impact on it beyond noting that RandCompile breaks the assumptions it relies on.

Summarizing the experiment, for each kernel variant we gathered:

1. Katana's output (success/failure in listing processes, modules, files, dmesg; number of elements/fields recovered).
2. HyperLink's output for process listing (success or failure).
3. Observations from GDB/vmcore analysis (like whether init_task or the task list was findable via known signatures).
4. Basic performance metrics via microbenchmarks run inside the VM (to measure overhead).

Microbenchmark Performance Tests: To measure performance impact, we used the LMbench suite, specifically focusing on operations sensitive to the changes we made:

- System call latency (null syscall getppid()).
- Context switch latency (two processes and 16 processes cases).
- Process creation (fork+exec) latency.

These tests (lat_syscall, lat_ctx, lat_proc in LMbench terms) are good indicators of overhead in low-level kernel operations. If RandCompile's added instructions or register juggling had a big impact, it would show up here. We ran each benchmark multiple times on each kernel and took the average. The system was otherwise idle during the tests.

All experiments were conducted on the same host machine to ensure consistency (an Intel i7-based system with virtualization support, and ample memory to avoid host swapping).

With the setup described, we proceed to the results, presenting how each forensic tool fared against the hardened kernels and how much overhead (if any) was incurred.

3 Results

3.1 Effect on Forensic Analysis Tools

Our first set of results examines how forensic tools performed on a standard kernel versus RandCompile-hardened kernels. We focus on Katana (offline memory analysis) and HyperLink (live introspection), as well as note our observations with LogicMem.

Katana Analysis: Table 3 summarizes Katana's ability to perform key analyses on the different kernel variants. A checkmark (✓) indicates the analysis succeeded in full, and a cross (✗) indicates it failed (or produced only partial/incomplete results that we consider a failure). We also note the number of structure members or entries reconstructed, as reported by Katana, for certain tasks as an indicator of partial success.

Table 3: Katana Memory Analysis Results on Baseline vs Hardened Kernels.

Analysis Task	Base (no hardening)	RandCompile (no bogus)	RandCompile (printk & mem only)	RandCompile (printk only)	RandCompile full
List Loaded Modules	✓ *(all modules)* – Members reconstructed: 2	✓ *(succeeds)* – 2	✓ *(succeeds)* – 2	✓ *(succeeds)* – 2	✓ *(suceeds)* – 2
List Processes (Task List)	✓ – Members reconstructed: 6 (all expected fields)	✗ – 5 members reconstructed (incomplete)	✗ – 5 members reconstructed	✗ – 4 members reconstructed	✗ – 4 members reconstructed
List Open Files	✓ – Members reconstructed: 16 (all file structs)	✗ – 15 reconstructed	✗ – 8 reconstructed	✗ – 7 reconstructed	✗ – 7 reconstructed
Extract Kernel dmesg log	✓ *– (log recovered successfully)*	✓ *– (log recovered)*	✓ *– (log recovered)*	✓ *– (log recovered)*	✗ *– (log not recovered)*

*(Table legend: "RandCompile (no bogus)" corresponds to our **nobogus** variant; "(printk & mem only)" corresponds to **bogusmem** variant (pointer encryption + printk externalization); "(printk only)" corresponds to a variant we tested with bogus parameters + pointer encryption off but printk externalization on, roughly akin to **bogusargs** minus pointer encryption; "full" is **forensic_hardening** variant with all features.)*

From the above results, we observe:

- Module listing was relatively unaffected by RandCompile. Katana could list kernel modules in all cases (all variants show success with 2 members reconstructed, which in context likely means it found the two key fields it looks for in the module list). This is not surprising: RandCompile did not specifically target module structures in our implementation, and module information might be easier to find (possibly via simple pattern matching or because modules' names weren't protected). Also, module listing is a simpler analysis that might not need many structure offsets.
- Process listing (task list reconstruction) was fully successful on the base kernel (Katana identified all 6 key members of the task structure it cares about). On any hardened kernel, Katana failed to produce a correct task list. Even in cases marked with some members reconstructed (e.g., 4 or 5), the analysis did not complete – Katana detected inconsistencies and did not output a process list. Essentially, a single fault in the reconstruction causes Katana to halt or mark the analysis as failed. The RandCompile authors noted that

"already a single fault during reconstruction causes a fault" in Katana's analysis. In our case, the presence of encrypted tasks pointers or missing "swapper/0" string likely caused that fault. Notably, even the mild "no bogus" hardening (which still included pointer encryption and string encryption) caused Katana to fail at task listing. This indicates that encrypting the task list pointers (FG3) or hiding the idle task's name (FG1) is enough to disrupt process enumeration. The number of members reconstructed (4–5 out of 6) suggests Katana may have partially recognized some structure (perhaps got some fields via heuristics) but not all, leading it to bail out.

- Listing open files is an even more complex task (involves traversing the file descriptor tables, etc.). On the base kernel, Katana recovered 16 members (presumably all needed info). On hardened kernels, it failed, with drastically fewer members (only 7–15) before giving up. The full hardening (all features) was the most disruptive (only 7 members vs 16 on base). This aligns with expectation: file listing requires combining information from processes and file structures, so if tasks aren't fully recovered, file listing will also fail. Additionally, bogus memory accesses may have misled Katana's analysis of file structs.
- Kernel log (dmesg) extraction succeeded on all but the fully hardened kernel. For base and intermediate variants, Katana apparently could still retrieve the log (likely by finding the log buffer structure). However, on the full RandCompile kernel, it failed (or produced gibberish). This is likely due to the printk format string externalization. In lesser variants, we might not have had that fully enabled, so Katana could still dump the log (though parts of it might have been unreadable, Katana might not detect that as failure). In the full variant, the log buffer likely did not yield coherent strings, so Katana's automated routines couldn't parse it, leading to a failure (we mark as ✗ since it did not produce a meaningful log).

Overall, Katana was effectively stymied by RandCompile's full hardening. Even partial hardening caused serious degradation of its results. These findings corroborate the original RandCompile evaluation: with all features on, none of Katana's deep analyses (task listing, file listing) succeeded. Simpler analyses (modules) that rely on easier patterns were unaffected – which is fine, as listing modules is generally less sensitive (and often possible via simpler means anyway).

HyperLink (live process listing): On the baseline kernel, HyperLink was able to enumerate all running processes without issue. Using the debug symbols or known signatures, we provided HyperLink the address of init_task, and it followed the tasks list to output the process list successfully. This shows the baseline kernel's memory was consistent with HyperLink's expectations: the tasks pointers formed a valid circular list linking all tasks (e.g., swapper/0, init, kernel threads, etc.), and the task_struct had the usual layout so that HyperLink could interpret it.

On the RandCompile-hardened kernels, HyperLink failed to list processes. Specifically, when we attempted to point it to what would be init_task, HyperLink reported errors like "no member named tasks" or could not traverse the list. In one scenario, using GDB on a running hardened kernel, the tasks.next pointer of init_task did not lead to a valid task – because it was encrypted, GDB showed it as an invalid address. Thus, if HyperLink tried to follow it, it either found no valid structure or ended up in a loop incorrectly. The message "No member named tasks" suggests that even accessing the tasks field via HyperLink's introspection failed, possibly due to ABI changes (HyperLink might be using debug symbols that didn't match, or we gave it wrong info because of RandCompile). The outcome was that the process list could not be recovered on hardened kernels. This demonstrates that RandCompile not only affects offline analysis but also thwarts real-time introspection. An analyst cannot simply attach to a running VM and walk the task list in memory, because the pointers and data no longer line up.

Our HyperLink test highlights "forensic evasion in real time" – the hardened kernel is actively resisting introspection attempts. In a security context, this means a malicious hypervisor cannot easily spy on a running VM's processes without doing significantly more work (and perhaps not in real time).

LogicMem: As noted, LogicMem did not successfully produce results on the baseline in our setup. We provide our observations anyway: on the base kernel, LogicMem's analysis did not complete (likely stuck or ended with nothing). The RandCompile hardened kernels would only fare worse, since in addition to the issues it had with base, now fundamental assumptions (like locations of task_struct fields) are invalid. The RandCompile authors mentioned that LogicMem's rigid rules were invalidated by structure layout randomization and other changes. In effect, RandCompile's introduction of random padding and field order shuffling (note: even without RandCompile, our kernel had some SLR which might have thrown LogicMem off) makes LogicMem ineffective. We concur with their conclusion that LogicMem's approach is less flexible and essentially fails completely when confronted with a diversified kernel. The takeaway is that not all forensic tools are equally robust – but RandCompile defeats even those that are more robust (Katana, HyperLink), while others like LogicMem already struggle.

To sum up the forensic impact: RandCompile successfully removes the leverage points that automated memory analysis tools rely on. In our tests, a fully hardened kernel was essentially opaque to these tools – processes could not be listed, kernel logs not extracted, etc. This confirms the core goal of RandCompile: enhancing privacy and security for kernel memory in untrusted environments. An interesting point is that simpler artifacts (like module lists) may remain obtainable, which is acceptable since modules are less sensitive than, say, process memory or logs; besides, further enhancements could also obfuscate module lists if needed (e.g., randomizing module names in memory, though that might be too intrusive).

3.2 Performance Overhead

The second aspect of our evaluation is the performance impact of RandCompile's hardening. We ran LMbench microbenchmarks on each variant to capture the overhead on fundamental operations. The results are shown in Table 4.

Table 4: Microbenchmark Performance (LMbench) on Base vs Hardened Kernels

Kernel Variant	System Call Latency (lat_syscall microbenchmark, µs)	Context Switch (2 processes, lat_ctx_2P, µs)	Context Switch (16 processes, lat_ctx_16P, µs)	Process fork+exec (lat_proc, µs)
base	1.26 ± 0.05	3.10	12.32	212.8
base_ftrace	1.40	3.40	14.27	234.0
bogusargs	1.68	4.15	16.04	251.3
bogusmem	1.74	4.46	17.84	265.0
nobogus	1.25	3.20	13.08	217.5
forensic_hardening	2.43	6.23	22.05	313.4

(Each value is an average; for brevity standard deviations are not listed but were low. Lower is better in all columns. "lat_ctx_2P" measures context switch time between two processes, "lat_ctx_16P" with sixteen processes. "lat_proc" measures the time to fork a process and have it exec a new program.)

Several observations can be made:

- Enabling FTRACE and debug symbols (base_ftrace) incurred a small overhead over base (about 11% on syscall latency, 10% on context switching, ~10% on fork+exec). This is expected overhead from instrumentation in the kernel. It provides a reference point: a typical kernel with some debugging features costs a bit of performance.
- The nobogus variant (RandCompile without bogus param feature) performed virtually identical to the base kernel (in fact, lat_syscall 1.25 vs 1.26, etc., are within noise). This suggests that the core RandCompile changes (pointer encryption, string encryption, and simple parameter reordering) introduce negligible overhead. Indeed, XORing pointers or shuffling registers doesn't slow down a null syscall in any measurable way. This aligns with the claim that those transformations "do not add performance overhead that cannot be explained by noise".
- The bogusargs variant (function call obfuscation only) showed some overhead: system call latency increased to ~1.68 µs (33% slower than base), context switch to 4.15 µs (34% slower), and fork+exec to 251 µs (about 18% slower than base). This indicates that adding bogus parameters and reordering did have a cost. Why would a null syscall be slower? Possibly because with ABI changes, the system call handling functions (which are internal) had an altered calling convention or dummy operations. The overhead is still small in absolute terms (difference of ~0.4 µs), but measurable. Context switching involves saving/restoring registers – with more dummy arguments, maybe more registers are live, increasing the context save/restore cost slightly. Also, larger task structures due to padding could affect context switch. The fork+exec overhead (which is more complex, involving creating a task struct, etc.) increased by ~18 ms, which could be partly due to pointer encryption overhead when linking the new task into the task list (though that should be minor), or more likely the cumulative effect of many small slowdowns.
- The bogusmem variant (pointer & string encryption, no bogus params) also showed overhead, similar or slightly more than bogusargs. lat_syscall ~1.74 µs, context switch ~4.46 µs, fork+exec 265 µs. This suggests pointer encryption and related changes do have some impact, potentially because certain operations like context switching or fork might involve manipulating those encrypted pointers (e.g., adding a new process to the list, which now involves encryption/decryption). The overhead here (~40% on syscalls, ~43% on ctx switch) is notable. It could also partly come from printk changes if the benchmark triggers any logging (though unlikely). It's possible that enabling the plugin or certain compiler changes also results in slightly less optimized code in general.
- The forensic_hardening (full) variant shows the highest overhead, as expected, since it combines everything. System call latency almost doubled (2.43 vs 1.26 µs). Context switch times roughly doubled as well (6.23 vs 3.10 for 2P, 22.05 vs 12.32 for 16P). Fork+exec went up by ~47% (313 vs 212 µs). While these slowdowns are not negligible, in absolute terms the performance is still decent for many uses (e.g., ~2.4 µs syscall overhead is still very fast). It's also comparable or lower than overheads of other security features: for instance, enabling kernel debugging or certain tracers can double syscall latency, and many production systems accept that overhead for safety. The RandCompile paper claimed <5% overhead overall; our microbenchmark results show worse percentages, but microbenchmarks tend to exaggerate overhead because they stress specific paths repeatedly. A <5% figure might refer to a macro benchmark or average across many operations. It's plausible that in a real workload, the impact of an extra microsecond here or there is diluted.

We interpret these results as follows: RandCompile's performance cost is modest for the security it provides. In scenarios where maximum performance is needed, one could choose to enable only a subset of features (e.g., skip bogus parameters) to reduce overhead, since the data suggests that the bulk of the slowdown comes from the most obfuscation-heavy features. For example, the nobogus variant had essentially no slowdown but still provides significant protection (string/pointer encryption and parameter shuffling alone defeated many forensic analyses). The full hardening roughly halves the performance on very low-level operations; whether this is acceptable depends on the use case. For a cloud VM where throughput is important, a ~2x slower context switch might have some

impact on high context-switch workloads, but for many real-world tasks (web servers, etc.), this overhead might be in the noise.

Finally, from a memory footprint perspective, the hardened kernels were slightly larger in code size due to added instructions for dummy accesses and encryption. The increase was on the order of a few percent (the kernel image grew by a few hundred KB). There was no noticeable increase in runtime memory usage beyond that.

In conclusion, our performance evaluation indicates that RandCompile's security enhancements come at a reasonable cost. The key takeaway is that security gains come at little cost – minimal performance impact and no changes needed to running software. For many deployments, this trade-off will be worth it, especially when weighed against the benefit of preventing covert memory snooping.

4 Conclusion

In this paper, we presented a comprehensive examination of RandCompile, a compile-time framework for removing forensic gadgets from the Linux kernel. By employing a combination of data obfuscation (encrypting pointer and string constants) and ABI perturbation (randomizing function parameter order and inserting bogus parameters), RandCompile makes a Linux kernel's memory layout and execution patterns significantly less transparent to automated analysis[1]. We rewrote and expanded upon the initial report of RandCompile, incorporating details from the original research paper (ACSAC 2023) and our own experimental results.

Key findings:

- RandCompile effectively disables or impairs memory forensic tools. In our tests, advanced tools like Katana could not reconstruct core kernel structures (process lists, open file tables, etc.) on a hardened kernel, and live introspection via HyperLink failed to retrieve runtime information[2][4]. This demonstrates that RandCompile achieves its goal of frustrating unauthorized kernel analysis, thereby enhancing the privacy and security of systems in adversarial environments.
- The defenses implemented (string encryption, pointer encryption, printk externalization, ABI randomization, bogus parameters) each address specific forensic gadgets and, together, provide a comprehensive hardening. Even individually, some defenses (e.g., encrypting the "swapper/0" string) have outsized impact, completely blocking certain analysis methods[1].
- Performance overhead of RandCompile is moderate[1]. Microbenchmarks showed that the full suite of RandCompile features can introduce noticeable overhead (1.5–2x latency for low-level operations), but a configuration with partial features can keep overhead negligible (<5% impact) while still defeating most forensic analysis. This flexibility allows deployment in performance-sensitive contexts by tuning which defenses to enable. In practice, the overhead is comparable to or less than many existing security features, which is promising for real-world adoption.
- RandCompile's approach is practical for scenarios like cloud VMs or specialized devices: it requires no runtime support and minimal kernel code changes, just a custom compilation process. We built and ran multiple hardened kernels without issue, indicating the approach is mature enough to consider outside the lab. The code has been made available by its authors, paving the way for further experimentation by the community.

5 Acknowledgments

We thank Professor Sven Dietrich for their guidance on this project and for suggesting the exploration of compile-time kernel defenses. We also acknowledge Fabian Franzen and collaborators for sharing their RandCompile research and code, which laid the foundation for our implementation. This work was conducted as part of the CSCI 49381 course, and we are grateful to our fellow students for their feedback during presentations.

Finally, we thank the maintainers of QEMU, Katana, HyperLink, and other tools we used - their open-source projects made our evaluation possible.

6 References

1. Fabian Franzen, Andreas Chris Wilhelmer, Jens Grossklags. “RandCompile: Removing Forensic Gadgets from the Linux Kernel to Combat its Analysis.” ACSAC 2023 – Annual Computer Security Applications Conference, 2023. (ACM Digital Library)
2. Fabian Franzen, Tobias Holl, Manuel Andreas, Johannes Kirsch, Jens Grossklags. “Katana: Robust, Automated, Binary-Only Forensic Analysis of Linux Memory Snapshots.” 25th International Symposium on Research in Attacks, Intrusions and Defenses (RAID), 2022, pp. 214–231.
3. Zhenxiao Qi, Yu Qu, Heng Yin. “LogicMem: Automatic Profile Generation for Binary-Only Memory Forensics via Logic Inference.” Network and Distributed System Security Symposium (NDSS), 2022.
4. Daniel Marth, Clemens Hlauschek, Christian Schanes, Thomas Grechenig. “Abusing Trust: Mobile Kernel Subversion via TrustZone Rootkits.” 16th IEEE Workshop on Offensive Technologies (WOOT), 2022.
5. Michael Cohen. “Rekall Forensic Framework: Memory Analysis in Open Source.” OSDFCon 2016 – Open Source Digital Forensics Conference, 2016. (Presentation introducing Rekall)
6. Volatility Foundation. “The Volatility Framework: Memory Forensics.” Online: volatilityfoundation.org, first released 2007. (Accessed 2025).
7. IBM Developer. “Kernel Address Space Layout Randomization (KASLR) support.” IBM Documentation, 2024. (Accessed 2025).
8. Red Hat. “What is the Linux kernel?” RedHat Topic Article, 2019. (General background on Linux kernel).
9. SentinelOne Labs. “Cybersecurity Forensics: Types and Best Practices.” SentinelOne Cybersecurity 101 Series, 2024. (Background on forensic analysis).