Memory Safety in Linux Device Drivers: Enhancing Security with Formal Verification

Janislley O. Sousa, David A. O. Ferreira
Federal University of Amazonas (UFAM),
SIDIA Institute of Science and Technology
Manaus / Brazil
{janislley, david.ferreira-br}@ieee.org

Eddie B. De Lima Filho
Federal University of Amazonas,
TPV Technology
Manaus / Brazil
eddie.filho@tpv-tech.com

Lucas C. Cordeiro
Federal University of Amazonas,
The University of Manchester (UoM)
Manaus / Brazil
lucas.cordeiro@manchester.ac.uk

***Abstract*—Memory safety remains one of the most critical challenges in embedded systems, particularly within the Linux kernel, where device drivers are a primary source of vulnerabilities due to their close interaction with hardware and complex execution contexts. Such behavior is even more pronounced on consumer electronics devices, where restricted resources amplify the existing vulnerabilities. Traditional testing approaches, while effective at uncovering shallow bugs, often fail to guarantee the absence of subtle or deeply nested memory-safety issues. In this paper, we propose a formal verification methodology based on Bounded Model Checking (BMC) to detect memory-safety violations in Linux device drivers. Using a methodology based on LSVerifier, we systematically analyzed newly integrated device drivers across multiple kernel versions. Our approach uncovered three vulnerabilities, including out-of-bounds writes, out-of-bounds reads, and null pointer dereferences. It demonstrates that formal methods can uncover critical flaws early in the development process, complementing dynamic tools and strengthening kernel security.**

***Index Terms*—Linux Device Drivers, Bounded Model Checking, Formal Verification, Software Vulnerabilities**

I. Introduction

Over the past two decades, memory-safety vulnerabilities have accounted for about two-thirds of critical security flaws in major system software, affecting both open-source and proprietary platforms based on Linux, Android, iOS, Chromium, and FreeRTOS [1]. These vulnerabilities are primarily caused by the use of memory-unsafe languages such as C, which permit low-level memory manipulation without built-in safeguards. Common issues include out-of-bounds accesses, use-after-free errors, null-pointer dereferencing, and other forms of memory corruption [2]. Although many of these flaws are discovered and patched before exploitation, memory-safety bugs remain the root cause of a substantial proportion of zero-day vulnerabilities. For instance, Google's Project Zero reported that 67% of in-the-wild zero-day exploits disclosed in 2021 were due to memory corruption vulnerabilities, with 25% attributed specifically to spatial safety violations [1].

The authors would like to thank Centro de Tecnologia Eletrônica e de Informação (CETELI) and the Department of Electrical Engineering at UFAM, Brazil, for providing the resources that were fundamental to this research. The authors also gratefully acknowledge the Department of Computer Science at UoM, the Systems and Software Security (S3) Research Group, and the Google ASPIRE Fund 2021 Program: Formal Verification Driven Fuzzing of Trusty OS for their support and access to state-of-the-art resources. In addition, the authors acknowledge the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES), Brazil, for financial support.

The Linux kernel represents a particularly compelling case in the context of memory safety. As a monolithic and rapidly evolving codebase with over 28 million lines of code, it is maintained by a global community that contributes more than $12,000$ patches per release cycle. Among its subsystems, device drivers are especially error-prone due to their tight coupling with hardware, asynchronous execution models, and complex memory interactions. A study of $1,858$ kernel vulnerabilities, from 2010 to 2020, revealed that a significant portion originated in device drivers, often resulting in local privilege escalations or system crashes [3]. Moreover, this risk is further magnified in consumer electronics devices, which depend heavily on Linux and its derivatives (e.g., Android) and are constrained by limited computational resources. In this context, a high number of security vulnerabilities are discovered in Android each year [4].

In addition, Google has reported that approximately 70% of severe vulnerabilities in C/C++ codebases stem from memory-safety issues [5], showing a density of around $1,000$ memory-safety vulnerabilities per million lines of code [6]. This analysis highlights vulnerabilities in memory-unsafe languages and underscores the urgent need to improve verification of Linux device drivers as a critical step toward enhancing the security and resilience of modern computing systems.

To address this gap, we present an automated formal verification framework based on BMC for analyzing Linux device drivers. Built upon Large System Verifier (LSVerifier) [7] and Efficient SMT-based Context-Bounded Model Checker (ESBMC) [8], our framework introduces several key capabilities that address the limitations of traditional approaches (e.g., static analyzers and fuzzing). The proposed methodology automates the generation of verification harnesses, tailoring them to specific driver entry points, which significantly reduces manual effort and enables consistent testing of diverse behaviors. Also, it integrates directly with kernel development environments, allowing seamless adoption during upstream patch reviews and enabling maintainers to identify memory-safety issues in early development cycles. This enables the systematic detection of memory-safety violations, while producing counterexamples for debugging. Our contributions are: (I) a survey of memory-safety verification techniques for the Linux kernel; (II) a scalable methodology for Linux device driver verification based on BMC; and (III) a verification-driven analysis of vulnerabilities in upstream Linux drivers.

The remainder of the paper is organized as follows. Section II provides background on Linux kernel vulnerabilities and reviews current memory-safety techniques and tools. Then, Section III outlines our verification approach. Next, Section IV presents our evaluation results, which identify real vulnerabilities in Linux device drivers. Finally, Section V summarizes our key findings and contributions.

II. Memory Safety in Linux Device Drivers

A. Linux Kernel Vulnerabilities

The upstream Linux kernel has experienced a steady increase in reported vulnerabilities, particularly those classified as high or critical severity by the Common Vulnerability Scoring System (CVSS), as reflected in Common Vulnerabilities and Exposures (CVE) data since 2024 [9] and shown in Table I. According to Luedtke *et al.* [10], this trend reveals an expanding attack surface, reinforcing the urgent need for enhanced security practices. One key contributor to this rise is the rapid pace of kernel development, which incorporates approximately $12,500$ commits per release from around $1,500$ developers. Each release modifies more than $700,000$ lines of code and expands the Linux codebase by roughly $290,000$ lines [11]. Such a scale and complexity significantly elevate the risk of memory-safety flaws, including memory leaks, buffer overflows, use-after-free errors, and boundary violations.

TABLE I
Linux Kernel Vulnerabilities in the Last Ten Years.

Year	Total of CVEs	CVSS Severity			
		Low	Medium	High	Critical
2016	216	1	102	88	25
2017	452	3	131	256	62
2018	180	3	87	85	5
2019	291	12	143	108	28
2020	129	2	75	52	0
2021	163	8	75	79	1
2022	310	8	163	135	4
2023	313	4	167	132	10
2024	4346	94	3084	1148	20
2025	4580	1625	2210	745	0

B. Linux Kernel Security Vulnerability Detection Techniques

A wide range of techniques have been proposed to detect memory-safety issues in the Linux kernel. These techniques are enforced through a combination of static and dynamic analysis, run-time enforcement, fuzzing, formal methods, and hardware-assisted mechanisms. Each technique offers trade-offs in scalability, precision, and completeness [12]. Table II provides an overview of the most widely adopted techniques currently used to detect memory-related vulnerabilities.

TABLE II
Current Memory-safety Techniques used in the Linux Kernel.

Category	Verification Approach
Static Analysis	Compile-time detection of memory-related issues using code inspection, data-flow analysis, or adherence to safe coding standards.
Dynamic Analysis	Instrumentation-based runtime analysis to track program behavior and detect memory violations.
Fuzzing Techniques	Such tools generate malformed or random inputs and feed them into a target application to trigger unexpected behavior.
Run-time techniques	Runtime enforcement mechanisms detect memory-safety violations and either halt execution, mask behavior, or reduce exploitability. Software-only mechanisms include stack canaries, Address Space Layout Randomization, and Control-Flow Integrity.
Software Fault Isolation	This technique enables deterministic sandboxing using processes or virtual machines to contain the impact of memory-safety violations.
Hardware memory protection	Systems that enforce memory safety at runtime by deterministically detecting violations through hardware extensions. These mechanisms can prevent or trap invalid memory accesses.
Formal methods	Methods used to prove memory safety and broader correctness in system properties.

Static analysis tools do not execute code, using control and data-flow analysis to identify potential issues. In that sense, Coverity [13], Smatch [14], and Coccinelle [15], widely used options, help detect vulnerabilities, including buffer overflows, null-pointer dereferences, and use-after-free bugs. Specifically, Coverity is commonly adopted in the industry, but it is proprietary. It relies on annotations to suppress false positives, which can obscure true bugs or require manual reassessment as code evolves. Furthermore, other static tools target specific vulnerability classes: KINT detects integer overflows [16], UniSan finds information leaks due to uninitialized variables [17], and UBITect uses type-qualifier inference to catch use-before-initialization bugs [18]. Moreover, K-Miner [19] performs inter-procedural analysis to detect memory corruption, while CRIX [20] and K-MELD [21] identify missing error-state checks and memory leaks, respectively. Finally, tools such as HERO [22], RID [23], and EeCatch [24] focus on bugs in error-handling and reference-counting logic.

Besides, dynamic analysis tools instrument programs to monitor their behavior during execution. Specifically, Valgrind [25] uses shadow memory to detect invalid memory accesses, though it incurs significant runtime overhead. AddressSanitizer (ASAN) [26], in turn, tracks heap, stack, and global buffer overflows, as well as use-after-free and double-free bugs by inserting memory red zones and intercepting allocation routines. Furthermore, UndefinedBehaviorSanitizer (UBSAN) [27] detects undefined behaviors, including type mismatches, overflows, and invalid shifts. Moreover, for kernel-space verification, KernelAddressSanitizer (KASAN) [28] adapts ASAN to detect out-of-bounds and UAF errors in the kernel memory model, relying on shadow memory to track allocation states. Finally, the Kernel Concurrency Sanitizer (KCSAN) [29] detects data races through sampling-based watchpoints.

In addition, fuzzing is one of the most effective techniques for code evaluation. It works by generating malformed or randomized inputs and observing the system for crashes or anomalous behavior. In that sense, Modern fuzzers use instrumentation to measure code coverage and evolve inputs toward unexplored execution paths [30]. Specifically, Syzkaller [31], developed by Google, is the most prominent syscall-level fuzzer for the Linux kernel. It integrates with sanitizers such as KASAN and coverage tools like KCOV [32] to detect memory violations and report them via syzbot [33]. Additionally, several research projects have proposed enhancements to kernel fuzzing. IMF [34] infers syscall dependencies, MoonShine [35] distills high-quality seeds from syscall traces, HFL [36] combines fuzzing with symbolic execution for hybrid exploration, and Difuze [37] uses static analysis to create well-formed inputs for device driver fuzzing. Finally, race condition detection has also been addressed by tools such as Razzer [38] and KRace [39], whereas fuzzers such as Janus [40] and PeriScope [41] focus on file systems and hardware interfaces.

In contrast, formal verification exhaustively checks software behavior against safety properties. In particular, BMC is well-suited to detecting shallow but critical bugs in kernel C code. Moreover, CBMC [42] and ESBMC [43] symbolically execute all paths of a program up to a fixed bound, generating verification conditions solved via Satisfiability modulo theories (SMT) solvers. Additionally, Saturn [44] scales this approach using function summaries and loop unrolling, while Post and Küchlin [45] proposed automatic test harness generation to enable BMC-based verification of Linux drivers. Beyond individual components, the seL4 microkernel project [46] demonstrates the feasibility of verifying entire OS kernels using interactive theorem proving. Furthermore, EBF [47] integrates BMC and fuzzing by using BMC-generated counterexamples as fuzzing seeds, enhancing bug discovery through thread delay injection and branch coverage guidance.

Indeed, various verification techniques have been proposed to mitigate memory-safety issues in Linux device drivers, including hybrid fuzzing, symbolic execution, and BMC. Tools like CBMC and Saturn symbolically explore execution paths to uncover bugs such as buffer overflows and null dereferencing, while others automate test harness generation for scalability. However, many of these solutions suffer from practical limitations, such as a lack of integration with kernel build systems, high manual effort, or poor support for incremental verification of evolving drivers. To address these gaps, our approach provides a scalable and automated BMC-based framework tailored for Linux drivers. It generates harnesses automatically, integrates with kernel environments, and focuses on isolating and verifying the codebase of Linux device drivers.

III. Verification Methodology

We use LSVerifier, a model-checking-based tool that can detect software vulnerabilities in Linux device drivers, whose workflow is shown in Fig. 1. Unlike static analyzers limited to entry points or fuzzers constrained by coverage, LSVerifier systematically explores all paths within bounded execution lengths. In that sense, drivers are instrumented and then verified by a BMC checker that employs SMT solvers, producing counterexamples when a safety property is violated.

LSVerifier conducts a comprehensive verification process by specifying the target source-code directory and the required configuration, including solver, encoding, and verification methods. Subsequently, all `.c` files in the input directory are examined. In fact, all functions are listed and then ranked according to a priority system, which considers their failure likelihood. For instance, functions with pointers as parameters or that perform memory manipulation are usually more prone to errors, which leads to a priority evaluation. This way, the most fragile elements are handled first, which results in more stable systems at early development phases.

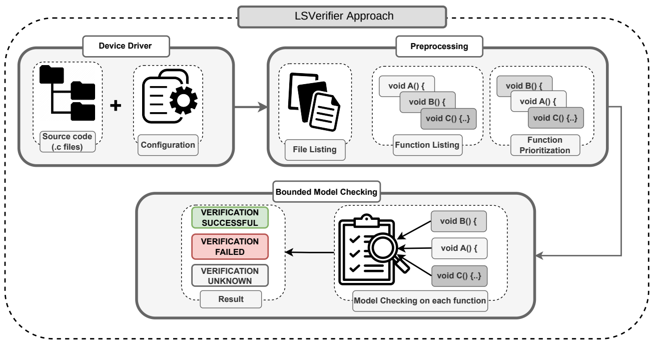


Fig. 1. The LSVerifier verification approach for Linux device drivers.

At the final stages, an underlying BMC checker executes C programs up to a bounded depth, in a function-by-function fashion, thus issuing analysis results: verification successfully, verification failed, or verification unknown. Different checkers can be employed, which shall follow the desired assessment.

Any property violations found during a verification procedure are informed and categorized by LSVerifier via a detailed report. For example, if a buffer overflow is detected, it flags the problematic function, shows the violated bounds, and generates a detailed report with the corresponding counterexample, which aids developers in understanding the root cause. It includes a sequence of states and transitions that show how a system evolves from an initial state to a state where a property is violated. Consequently, by analyzing a counterexample, developers can understand what causes a violation and then take corrective actions to fix its underlying issue.

IV. Experimental Results

We have analyzed different kernel versions. Each driver was instrumented and checked using LSVerifier within a bounded model-checking context.

A. Challenges In Verifying The Linux Kernel

The Linux kernel verification presents unique challenges due to its complexity and specific architectural characteristics: one significant issue is the frequent reuse of global data structures. Indeed, many subsystems and components share access to global memory, making it difficult to track their state and interactions accurately.

Another challenge lies in synchronizing the memory states between individual system calls and the global kernel state. Each call operates within its own context but often interacts with shared global resources, resulting in intricate dependencies and race conditions that are challenging to analyze and verify systematically. Additionally, the Linux kernel's use of pointers introduces deeply nested and complex aliasing relationships. Pointers may reference other pointers or dynamically allocated memory, creating layers of indirection that complicate the identification of true data dependencies and the detection of potential vulnerabilities, such as dangling pointers and use-after-free errors.

Moreover, scalable static analysis tools originally designed for user-space programs are often ill-suited for the Linux kernel's environment. For instance, data-flow analysis in user-space applications typically starts from a well-defined initial state, such as the program's `main` function. This state provides a clear starting point for propagating value flows throughout the analysis. In contrast, the Linux kernel lacks a single entry point. Instead, it relies on various system call entry points, interrupt handlers, and other dynamically invoked functions, making it far more challenging to establish a consistent initial state. These aspects highlight the need for kernel-specific verification techniques and tools that ensure both scalability and accuracy in the analysis process.

B. Experimental Setup

Our experimental evaluation targeted Linux kernel versions 4.19, 5.10, and 6.1, the most commonly deployed in embedded system projects. ESBMC was integrated into LSVerifier as the underlying BMC checker, which led to the instrumentation of device driver code by inserting assertion checks around critical functions. Since kernel drivers often include deep pointer chains, dynamic memory allocations, and other complex structures, we imposed a per-driver time limit on each verification run to ensure tractable analysis. The BMC symbolic engine addresses memory safety issues (e.g., pointer safety, array bounds, and arithmetic overflows) with support from SMT backends, making it adaptable to a range of verification tasks and complex memory models, such as those found in kernel drivers. Finally, we link the identified weaknesses with the Common Weakness Enumeration (CWE) list.

C. Experiments and Discussion

1) The Interfacing Driver: In the Linux kernel, the Human Interface Device (HID) driver handles communication with input devices like keyboards, mice, and gamepads. The core HID driver parses reports, gets field indexes, and manages events. The HID subsystem supports various transports, including USB, Bluetooth, I2C, and user-space input/output.

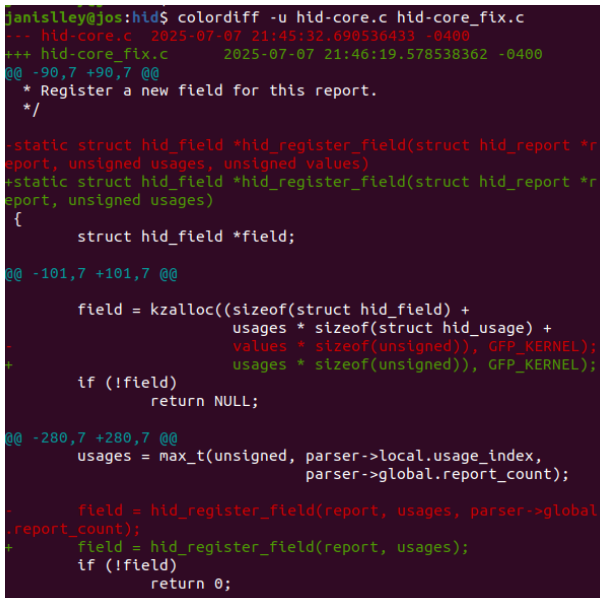
During the verification of kernel version 4.14, we were able to exploit a vulnerability in file *hid-core.c*. The function *hidinput_change_resolution_multipliers()* in *hid-input.c* could perform an out-of-bounds write into the *valid_bitmap* heap buffer, due to mismatched array sizes for elements `usages` and `values`, classified as CWE-787. An exploit of this flaw enables a local attacker to corrupt heap metadata and achieve privilege escalation without additional execution privileges or user interaction. The verification result is described in Fig. 2.

To address this, the arrays were resized to the same size. Our fix unifies the allocation of the arrays `values` and `usages` by deriving both sizes from a single parameter. In *hid_register_field()*, we allocate a single contiguous block of memory, as shown in Fig. 3. In the allocator call, we now reserve entries twice: once for the array `usages` and once for the array `values`, by multiplying usages by the combined size *sizeof(struct hid_usage) + sizeof(unsigned)*. We set both `field->maxusage` and `field->maxvalue` to

```
/kernel-4.14/drivers/hid/hid-core.c
Building error trace

Counterexample:

State 2 file hid-core_fix.c line 102
function hid_register_field thread 0
----------------------------------------
Violated property:
file hid-core_fix.c line 102
function hid_register_field
array bounds violated: attempt to write 1
element past 'usages * sizeof(struct hid_usage)'

VERIFICATION FAILED
```

Fig. 2. Verification results for the HID driver.

usages, so that any later index into either array will stay within bounds. This guarantees that any subsequent index into either array remains within its allocated region. By collapsing the former two-parameter interface into a single-parameter application programming interface and enforcing identical bounds for both arrays, we fully eliminate the out-of-bounds write in *hidinput_change_resolution_multipliers()* as well as the corresponding read in *hidinput_count_leds()*.



```
janislley@jos:hid$ colordiff -u hid-core.c hid-core_fix.c
--- hid-core.c  2025-07-07 21:45:32.690536433 -0400
+++ hid-core_fix.c      2025-07-07 21:46:19.578538362 -0400
@@ -90,7 +90,7 @@
  * Register a new field for this report.
  */

-static struct hid_field *hid_register_field(struct hid_report *r
eport, unsigned usages, unsigned values)
+static struct hid_field *hid_register_field(struct hid_report *r
eport, unsigned usages)
 {
        struct hid_field *field;

@@ -101,7 +101,7 @@

        field = kzalloc((sizeof(struct hid_field) +
                         usages * sizeof(struct hid_usage) +
-                        values * sizeof(unsigned)), GFP_KERNEL);
+                        usages * sizeof(unsigned)), GFP_KERNEL);
        if (!field)
                return NULL;

@@ -280,7 +280,7 @@
        usages = max_t(unsigned, parser->local.usage_index,
                                 parser->global.report_count);

-       field = hid_register_field(report, usages, parser->global
.report_count);
+       field = hid_register_field(report, usages);
        if (!field)
                return 0;
```

Fig. 3. Out-of-bounds write (CWE-787) in the HID driver.

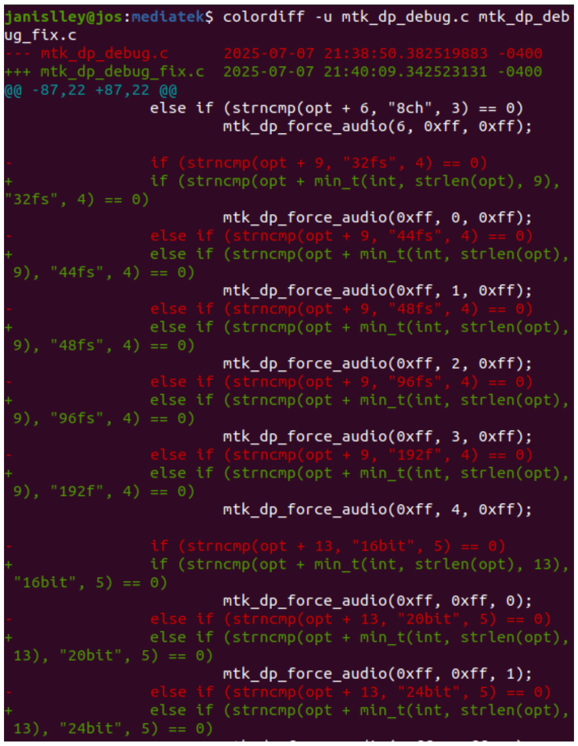
2) The Rendering Driver: The Direct Rendering Manager (DRM) driver manages communication with Graphics Processing Units (GPUs), thus enabling user-space programs to interact with them. It provides features such as display management, memory management, and command submission. During the verification of kernel version 5.10, we identified an out-of-bounds read vulnerability in *mtk_dp_debug.c* file, classified as CWE-125. The defect is triggered when the code evaluates `strncmp(opt + 9, "32fs", 4)` without first ensuring that `opt` is long enough, causing `strncmp()` to dereference bytes beyond the end of the option buffer, poten-

```
/kernel-5.10/drivers/gpu/drm/mediatek/mtk_dp_debug.c
Building error trace

Counterexample:

State 2 file mtk_dp_debug.c line 90
function mtk_dp_debug thread 0
----------------------------------------
Violated property:
file mtk_dp_debug.c line 90
function mtk_dp_debug
array bounds violated: attempt to write 1
element past 'strncmp(opt + 9, "32fs", 4) == 0'

VERIFICATION FAILED
```

Fig. 4. Verification results for the DRM driver.

tially leading to information disclosure or system instability, as described in Fig. 4.

To address this potential vulnerability, we ensure that the value passed to *strncmp()* never moves beyond its valid range, as shown in Fig. 5. LSVerifier has confirmed that this change resolves all out-of-bounds warnings in this function.



```
janislley@jos:mediatek$ colordiff -u mtk_dp_debug.c mtk_dp_deb
ug_fix.c
--- mtk_dp_debug.c      2025-07-07 21:38:50.382519883 -0400
+++ mtk_dp_debug_fix.c  2025-07-07 21:40:09.342523131 -0400
@@ -87,22 +87,22 @@
                else if (strncmp(opt + 6, "8ch", 3) == 0)
                        mtk_dp_force_audio(6, 0xff, 0xff);

-               if (strncmp(opt + 9, "32fs", 4) == 0)
+               if (strncmp(opt + min_t(int, strlen(opt), 9),
"32fs", 4) == 0)
                        mtk_dp_force_audio(0xff, 0, 0xff);
-               else if (strncmp(opt + 9, "44fs", 4) == 0)
+               else if (strncmp(opt + min_t(int, strlen(opt),
 9), "44fs", 4) == 0)
                        mtk_dp_force_audio(0xff, 1, 0xff);
-               else if (strncmp(opt + 9, "48fs", 4) == 0)
+               else if (strncmp(opt + min_t(int, strlen(opt),
 9), "48fs", 4) == 0)
                        mtk_dp_force_audio(0xff, 2, 0xff);
-               else if (strncmp(opt + 9, "96fs", 4) == 0)
+               else if (strncmp(opt + min_t(int, strlen(opt),
 9), "96fs", 4) == 0)
                        mtk_dp_force_audio(0xff, 3, 0xff);
-               else if (strncmp(opt + 9, "192f", 4) == 0)
+               else if (strncmp(opt + min_t(int, strlen(opt),
 9), "192f", 4) == 0)
                        mtk_dp_force_audio(0xff, 4, 0xff);

-               if (strncmp(opt + 13, "16bit", 5) == 0)
+               if (strncmp(opt + min_t(int, strlen(opt), 13),
 "16bit", 5) == 0)
                        mtk_dp_force_audio(0xff, 0xff, 0);
-               else if (strncmp(opt + 13, "20bit", 5) == 0)
+               else if (strncmp(opt + min_t(int, strlen(opt),
 13), "20bit", 5) == 0)
                        mtk_dp_force_audio(0xff, 0xff, 1);
-               else if (strncmp(opt + 13, "24bit", 5) == 0)
+               else if (strncmp(opt + min_t(int, strlen(opt),
 13), "24bit", 5) == 0)
```

Fig. 5. Out-of-bounds read (CWE-125) in the DRM driver.

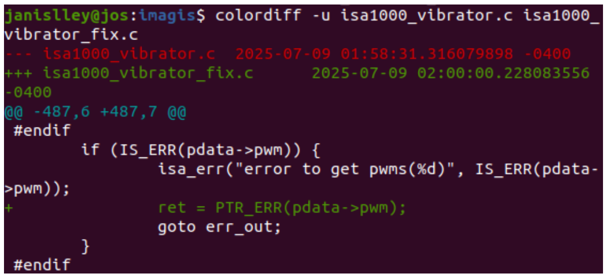
3) The Vibrator Driver: The vibrator driver is responsible for controlling haptic feedback on devices such as smartphones and is part of the input subsystem. This subsystem manages various input devices, including those that provide tactile feedback through vibration.

In the Linux kernel version 6.1, we identified a null pointer dereference vulnerability in the vibrator driver's *isa1000_get_devtree_pdata* function, where the pointer `data->pwm` may be NULL, which is classified as CWE-476. A NULL pointer dereference usually results in the process failing unless exception handling (on some platforms) is available and implemented. Even when exception handling is being used, it can still be very difficult to return the software to a safe state of operation. This potential vulnerability can result in a kernel panic when the Linux kernel is unable to handle the NULL pointer dereference, as shown in Fig. 6.

```
/kernel-6.1/drivers/vibrator/imagis/isa1000_vibrator.c
Building error trace

Counterexample:

State 1 file isa1000_vibrator.c line 488
function isa1000_get_devtree_pdata thread 0
----------------------------------------------------
Violated property:
file isa1000_vibrator.h line 488
function isa1000_get_devtree_pdata
dereference failure: NULL pointer

VERIFICATION FAILED
```

Fig. 6. Verification results for the Vibrator driver.

To address this issue, we ensure that the returned error code is captured by inserting ret = PTR_ERR (`data->pwm`). This change prevents unintended null-pointer dereferences by properly propagating the error code.



```
janislley@jos:imagis$ colordiff -u isa1000_vibrator.c isa1000_
vibrator_fix.c
--- isa1000_vibrator.c  2025-07-09 01:58:31.316079898 -0400
+++ isa1000_vibrator_fix.c      2025-07-09 02:00:00.228083556
-0400
@@ -487,6 +487,7 @@
 #endif
        if (IS_ERR(pdata->pwm)) {
                isa_err("error to get pwms(%d)", IS_ERR(pdata-
>pwm));
+               ret = PTR_ERR(pdata->pwm);
                goto err_out;
        }
 #endif
```

Fig. 7. NULL pointer dereference (CWE-476) in the Vibrator driver.

V. Conclusion

Memory-safety vulnerabilities continue to pose significant risks in low-level systems, and decades of empirical evidence suggest that detection and mitigation alone have not been sufficient to eliminate them [11], [30]. Consequently, a complementary methodology is indeed necessary, aiming to catch problems through automated code analysis.

The present work fills in this gap by identifying and patching previously undocumented vulnerabilities in upstream Linux device drivers through formal verification. To address these persistent challenges, we introduced a scalable formal verification framework that focuses on verifying individual subsystems rather than re-analyzing the entire kernel for each release. Using LSVerifier with ESBMC, we identified and resolved critical memory errors in Linux device drivers, including an out-of-bounds write, CWE-787, an out-of-bounds read, CWE-125, and a null pointer dereference, CWE-476.

The obtained results highlight that formal verification not only complements existing dynamic techniques, such as fuzzing and sanitization, but also excels at uncovering deep semantic bugs that traditional methods often miss. Future work will focus on enhancing kernel verification by addressing module stacking interactions, asynchronous task handling, and contextual resource allocation, as well as extending the analysis toward inter-procedural behaviors, concurrency effects, interrupt-driven executions, and driver life-cycles verification to better reflect full driver execution semantics, building on the prioritization approach that strengthened LSVerifier.

References

[1] Google Project Zero, "The more you know, the more you know you don't know," https://googleprojectzero.blogspot.com/2022/04/the-more-you-know-more-you-know-you.html, 2022, accessed: Oct. 20, 2025.

[2] P. C. van Oorschot, "Memory errors and memory safety: C as a case study," *IEEE Security & Privacy*, vol. 21, no. 2, pp. 70–76, 2023.

[3] A. Shameli-Sendi, "Understanding linux kernel vulnerabilities," *Journal of Computer Virology and Hacking Techniques*, vol. 17, no. 4, pp. 265–278, 2021.

[4] E. Andrade, J. Sousa, M. Lopes, D. Barbosa, W. Lima, and J. Lacerda, "Experiences with google approval process: an automated approach to enhancing efficiency in android releases," in *Simpósio Brasileiro de Testes de Software Sistemático e Automatizado (SAST)*. SBC, 2024, pp. 83–85.

[5] C. S. Team, "An update on memory safety in chrome," https://security.googleblog.com/2021/09/an-update-on-memory-safety-in-chrome.html, Sep. 2021, accessed: Nov. 04, 2025.

[6] Jeff Vander Stoep, "The more you know, the more you know you don't know," https://security.googleblog.com/2025/11/rust-in-android-move-fast-fix-things.html, November 2025, accessed: Nov. 19, 2025.

[7] J. O. de Sousa, B. C. de Farias, T. A. da Silva, E. B. de Lima Filho, and L. C. Cordeiro, "Lsverifier: A bmc approach to identify security vulnerabilities in c open-source software projects," in *Simpósio Brasileiro de Segurança da Informação e de Sistemas Computacionais (SBSeg)*. SBC, 2023, pp. 17–24.

[8] L. Cordeiro, B. Fischer, and J. Marques-Silva, "Smt-based bounded model checking for embedded ansi-c software," *IEEE Transactions on Software Engineering*, vol. 38, no. 4, pp. 957–974, 2011.

[9] cvedetails, "Linux kernel security vulnerabilities," https://www.cvedetails.com/vulnerability-list/vendor_id-33/product_id-47/Linux-Linux-Kernel.html, 2025, accessed: Nov. 19, 2025.

[10] N. Luedtke, "Linux kernel cves," https://www.linuxkernelcves.com/, 2024, accessed: Oct. 20, 2025.

[11] M. Jiang and *et al.*, "Understanding vulnerability inducing commits of the linux kernel," *ACM Transactions on Software Engineering and Methodology*, vol. 33, no. 7, pp. 1–28, 2024.

[12] O. Llorente-Vazquez, I. Santos-Grueiro, and P. G. Bringas, "When memory corruption met concurrency: Vulnerabilities in concurrent programs," *IEEE Access*, vol. 11, pp. 44 725–44 740, 2023.

[13] Synopsys, "Coverity scan: Static analysis service for open-source projects," https://scan.coverity.com/, 2025, accessed: Oct. 20, 2025.

[14] Coverity, "Coverity scan static analysis," https://github.com/error27/smatch, accessed: Oct. 20, 2025.

[15] J. Lawall and G. Muller, "Coccinelle: 10 years of automated evolution in the linux kernel," in *USENIX ATC*, 2018, pp. 601–614.

[16] X. Wang and *et al.*, "Improving integer security for systems with {KINT}," in *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, 2012, pp. 163–177.

[17] K. Lu, C. Song, T. Kim, and W. Lee, "Unisan: Proactive kernel memory initialization to eliminate data leakages," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016, pp. 920–932.

[18] Y. Zhai and *et al.*, "Ubitect: a precise and scalable method to detect use-before-initialization bugs in linux kernel," in *Proceedings of the 28th ACM Joint Meeting on ESEC/FSE*, 2020, pp. 221–232.

[19] D. Gens, S. Schmitt, L. Davi, and A.-R. Sadeghi, "K-miner: Uncovering memory corruption in linux." in *NDSS*, 2018.

[20] K. Lu, A. Pakki, and Q. Wu, "Detecting {Missing-Check} bugs via semantic-and {Context-Aware} criticalness and constraints inferences," in *28th USENIX Security Symposium*, 2019, pp. 1769–1786.

[21] N. Emamdoost, Q. Wu, K. Lu, and S. McCamant, "Detecting kernel memory leaks in specialized modules with ownership reasoning," in *NDSS*, 2021.

[22] Q. Wu, A. Pakki, N. Emamdoost, S. McCamant, and K. Lu, "Understanding and detecting disordered error handling with precise function pairing," in *30th USENIX Security Symposium*, 2021, pp. 2041–2058.

[23] J. Mao, Y. Chen, Q. Xiao, and Y. Shi, "Rid: finding reference count bugs with inconsistent path pair checking," in *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, 2016, pp. 531–544.

[24] A. Pakki and K. Lu, "Exaggerated error handling hurts! an in-depth study and context-aware detection," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, 2020, pp. 1203–1218.

[25] Valgrind Developers, "Valgrind: Tool Suite for Debugging and Profiling Linux Programs," https://valgrind.org/, 2025, accessed: Oct. 20, 2025.

[26] LLVM Project, "AddressSanitizer," https://clang.llvm.org/docs/AddressSanitizer.html, 2024, accessed: Oct. 20, 2025.

[27] ——, "UndefinedBehaviorSanitizer," https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html, 2025, accessed: Oct. 20, 2025.

[28] Linux Kernel Doc., "KASAN," https://docs.kernel.org/dev-tools/kasan.html, 2025, accessed: Oct. 20, 2025.

[29] ——, "KCSAN," https://docs.kernel.org/dev-tools/kcsan.html, 2025, accessed: Oct. 20, 2025.

[30] Z. Yu, Z. Liu, X. Cong, X. Li, and L. Yin, "Fuzzing: Progress, challenges, and perspectives." *Computers, Materials & Continua*, vol. 78, no. 1, 2024.

[31] D. Vyukov, "Syzkaller," https://github.com/google/syzkaller, accessed: Oct. 20, 2025.

[32] Linux Kernel Doc., "KCOV ," https://docs.kernel.org/dev-tools/kcov.html, 2025, accessed: Oct. 20, 2025.

[33] Google, "syzbot," https://syzkaller.appspot.com/upstream/, accessed: Oct. 20, 2025.

[34] H. Han and S. K. Cha, "Imf: Inferred model-based fuzzer," in *Proceedings of the 2017 ACM SIGSAC conference on computer and communications security*, 2017, pp. 2345–2358.

[35] S. Pailoor, A. Aday, and S. Jana, "{MoonShine}: Optimizing {OS} fuzzer seed selection with trace distillation," in *27th USENIX Security Symposium (USENIX Security 18)*, 2018, pp. 729–743.

[36] K. Kim, D. R. Jeong, C. H. Kim, Y. Jang, I. Shin, and B. Lee, "Hfl: Hybrid fuzzing on the linux kernel." in *NDSS*, 2020.

[37] J. Corina, A. Machiry, C. Salls, Y. Shoshitaishvili, S. Hao, C. Kruegel, and G. Vigna, "Difuze: Interface aware fuzzing for kernel drivers," in *Proceedings of the 2017 ACM SIGSAC conference on computer and communications security*, 2017, pp. 2123–2138.

[38] D. R. Jeong, K. Kim, B. Shivakumar, B. Lee, and I. Shin, "Razzer: Finding kernel race bugs through fuzzing," in *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019, pp. 754–768.

[39] M. Xu, S. Kashyap, H. Zhao, and T. Kim, "Krace: Data race fuzzing for kernel file systems," in *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2020, pp. 1643–1660.

[40] W. Xu, H. Moon, S. Kashyap, P.-N. Tseng, and T. Kim, "Fuzzing file systems via two-dimensional input space exploration," in *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019, pp. 818–834.

[41] D. Song and *et al.*, "Periscope: An effective probing and fuzzing framework for the hardware-os boundary," in *2019 Network and Distributed Systems Security Symposium (NDSS)*. Internet Society, 2019, pp. 1–15.

[42] E. Clarke, D. Kroening, and F. Lerda, "A tool for checking ansi-c programs," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2004.

[43] M. R. Gadelha, R. S. Menezes, and L. C. Cordeiro, "Esbmc 6.1: automated test case generation using bounded model checking," *STTT*, vol. 23, no. 6, pp. 857–861, 2021.

[44] Y. Xie and A. Aiken, "Saturn: A sat-based tool for bug detection," in *International Conference on Computer Aided Verification*. Springer, Berlin, Heidelberg, 2005.

[45] H. Post and W. Küchlin, "Automatic data environment construction for static device drivers analysis," in *Proceedings of the 2006 Conference on Specification and Verification of Component-Based Systems*, 2006.

[46] G. Klein and *et al.*, "sel4: formal verification of an os kernel," in *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, ser. SOSP '09. New York, NY, USA: Association for Computing Machinery, 2009, p. 207–220.

[47] F. K. Aljaafari, R. Menezes, E. Manino, F. Shmarov, M. A. Mustafa, and L. C. Cordeiro, "Combining bmc and fuzzing techniques for finding software vulnerabilities in concurrent programs," *Ieee Access*, vol. 10, pp. 121 365–121 384, 2022.