DOI: 10.15514/ISPRAS-2025-37(5)-5

Coloring Symbolic Memory Graphs to Detect DRM-Specific Errors in Linux Drivers

E.M. Orlova, ORCID: 0009-0003-1654-3085 <e.orlova@ispras.ru>
A.A. Vasilyev, ORCID: 0000-0002-5738-9171 <vasilyev@ispras.ru>
O.M. Petrov, ORCID: 0009-0004-6245-9615 <o.petrov@ispras.ru>

Institute for System Programming of the Russian Academy of Sciences,
25, Alexander Solzhenitsyn st., Moscow, 109004, Russia.

Abstract. This paper discusses a particular type of subtle use-after-free errors in the Direct Rendering Manager (DRM) subsystem of the Linux kernel. These errors occur due to incorrectly allocated memory for structures accessible from user space via device callbacks. To detect these errors, we use a shape analysis based on the Symbolic Memory Graph (SMG) domain. We introduce the coloring of allocated memory to track its origin. Among 186 Linux DRM drivers, we have found 6 violations of the proposed rule.

Keywords: Linux drivers; use-after-free; shape analysis; software model checking; symbolic memory graphs.

For citation: Orlova E.M., Vasilyev A.A., Petrov O.M. Coloring Symbolic Memory Graphs to Detect DRM-Specific Errors in Linux Drivers. Trudy ISP RAN/Proc. ISP RAS, vol. 37, issue 5, 2025. pp. 67-80. DOI: 10.15514/ISPRAS-2025-37(5)-5.

Acknowledgements. The authors would like to thank Vadim Mutilin, colleagues from the Linux Verification Center, and the maintainers of the Linux DRM subsystem for their feedback and comments.

Окрашивание символьных графов памяти для выявления ошибок, специфичных для DRM-драйверов Linux

Е.М. Орлова, ORCID: 0009-0003-1654-3085 <e.orlova@ispras.ru>
А.А. Васильев, ORCID: 0000-0002-5738-9171 <vasilyev@ispras.ru>
О.М. Петров, ORCID: 0009-0004-6245-9615 <o.petrov@ispras.ru>

Институт системного программирования РАН,
Россия, 109004, г. Москва, ул. А. Солженицына, д. 25.

Аннотация. В статье рассматривается одна трудновоспроизводимая ошибка типа use-after-free в подсистеме Direct Rendering Manager (DRM) ядра операционной системы Linux. Её причиной является некорректный способ выделения памяти, доступной для пользовательского кода через обратные вызовы устройства. Для поиска ошибок работы с памятью мы используем анализ на основе символьных графов памяти (SMG). Чтобы отследить способ выделения памяти, мы добавили ей цвет. Среди 186 проанализированных драйверов DRM ОС Linux было найдено 6 нарушений предложенного правила.

Ключевые слова: драйверы Linux; уязвимость use-after-free; анализ динамической памяти; автоматическая статическая верификация; символьные графы памяти.

Для цитирования: Орлова Е.М., Васильев А.А., Петров О.М. Окрашивание символьных графов памяти для выявления ошибок, специфичных для DRM-драйверов Linux. Труды ИСП РАН, том 37, вып. 5, 2025 г., стр. 67-80 (на английском языке). DOI: 10.15514/ISPRAS–2025–37(5)-5

Благодарности: Авторы выражают благодарность Вадиму Мутилину, коллегам из Технологического центра исследования безопасности ядра Linux, а также разработчикам, поддерживающим подсистему DRM в Linux, за их отзывы и комментарии.

1. Introduction

The Linux operating system kernel is a widely used software system consisting of 25 million lines of code. Put simply, it consists of the kernel core and various subsystems and device drivers. To use a graphics processing unit (GPU), a user program invokes a system call so that the kernel core dispatches the appropriate callback in the corresponding device driver in the Direct Rendering Manager (DRM) kernel subsystem.

This paper discusses a particular type of errors related to the incorrect use of device resource management (devres) in the DRM subsystem. Incorrect memory allocation of structures accessible from user space can lead to a use-after-free memory access. Such outcomes can be detected with dynamic analysis, but situations in which the target errors cause the kernel to crash are specific and quite rare.

On the other hand, static verification methods [1] aim at detecting such subtle errors. Klever [2-3] is a software verification platform capable of automated static verification of industrial software systems using software model checking tools such as CPAchecker [4]. As many Linux errors are in its device drivers [5], Klever is tailored for bug-finding in the Linux subsystems.

Klever decomposes the kernel source code into modules, provides the environment models based on typical device usage, runs the verification tool, and displays the results, e.g. visualizes the reported error traces. Using this method, several hundred bugs in Linux subsystems were found and reported [6].

To verify memory safety, CPAchecker uses a shape analysis based on the Symbolic Memory Graph domain [7-10]. The analysis represents a program memory state as a bipartite graph, with its nodes being memory objects (concrete regions and abstracted linked lists) and symbolic values.

Contribution. We manually analyzed the Linux DRM subsystem and found a documented recommendation [11] on correct allocation that had been overlooked in 13 files [12]. This can lead

to potential use-after-free errors, some of which have already been reported by Kernel Address Sanitizer (KASAN).

We formulated a more general rule in natural language and formally specified it using Klever. To verify this rule, we adapted the memory analysis in the CPAchecker verification tool by adding color to the symbolic memory graphs. We evaluated this approach on 186 DRM modules, with all 6 reported violations of the rule manually confirmed. The results are discussed in comparison with the violations we found using Coccinelle [13]. The corresponding Coccinelle rule (*semantic patch*) was submitted to the kernel [14]. Finally, we are working on fixes for the discovered errors, and one patch has already been accepted upstream [15].

2. Problem Statement

There is a static driver structure through which device instances are managed. Through a certain interface, a device instance is accessible from the user space. Even if the driver is disabled, the device instance will still exist as long as it has at least one user. If memory is allocated incorrectly, the DRM device structure (or structures used by it) is automatically freed when the driver is unbound. Thus, while the device instance still exists, a user can cause access to this freed structure.

2.1 DRM device instance

At the core of every DRM driver is a `drm_driver` structure. It contains static information that describes the driver and features it supports, and pointers to methods that implement the DRM API. This structure is also used to create a device instance, which is then initialized and registered, providing callbacks accessible from the user space.

A device instance for the DRM driver is represented by the `drm_device` structure. It is allocated and initialized with `devm_drm_dev_alloc()` (or deprecated `drm_dev_alloc()`). After initialization of all the various DRM device subsystems when everything is ready for user space, the device instance can be published using `drm_dev_register()` [11].

When cleaning up, everything is done in reverse. First, the device instance is unpublished with `drm_dev_unregister()`. Then any other resources allocated at device initialization are cleaned up and drop the driver's reference to `drm_device` using `drm_dev_put()`. It is important to note that if `drm_device` still has some resource handles open when the driver is unbounded, the release of `drm_device` instance does not happen immediately, but only after the last handle is closed. Before that, `drm_device` remains user-accessible. This is why any allocation or resource which is visible to user space must be released only when the final `drm_dev_put()` is called, and not when the driver is unbound from the underlying physical `struct device`. Otherwise, using the device may result in accessing freed memory.

This imposes a restriction on which functions can be used to allocate structures that are accessible from user space through a `drm_device` instance.

2.2 Some Linux kernel memory allocation functions

Let's look at some of the memory allocation functions in more depth.

- `kmalloc()` – a kernel-space function similar to user-space `malloc()`. The memory allocated with this function must be freed by calling the `kfree()` function.
- `devm_kmalloc()` – devres-managed `kmalloc()`. Memory allocated with this function is automatically freed on driver detach. Its lifetime is linked to the device structure, a pointer to which is passed as a parameter.
- `drmm_kmalloc()` – DRM-managed `kmalloc()`. Memory allocated with this function is automatically freed on the final `drm_dev_put()`. Its lifetime is linked to the `drm_device` structure, a pointer to which is passed as a parameter.

`drmm_kmalloc()` is recommended to allocate the memory for the aforementioned structures accessible from user space. Then the memory will be automatically released when the `drm_device` is destroyed and only after its registration is canceled, as there will be no risk of accessing the released memory. The correct work of a DRM driver is shown in Fig. 1.

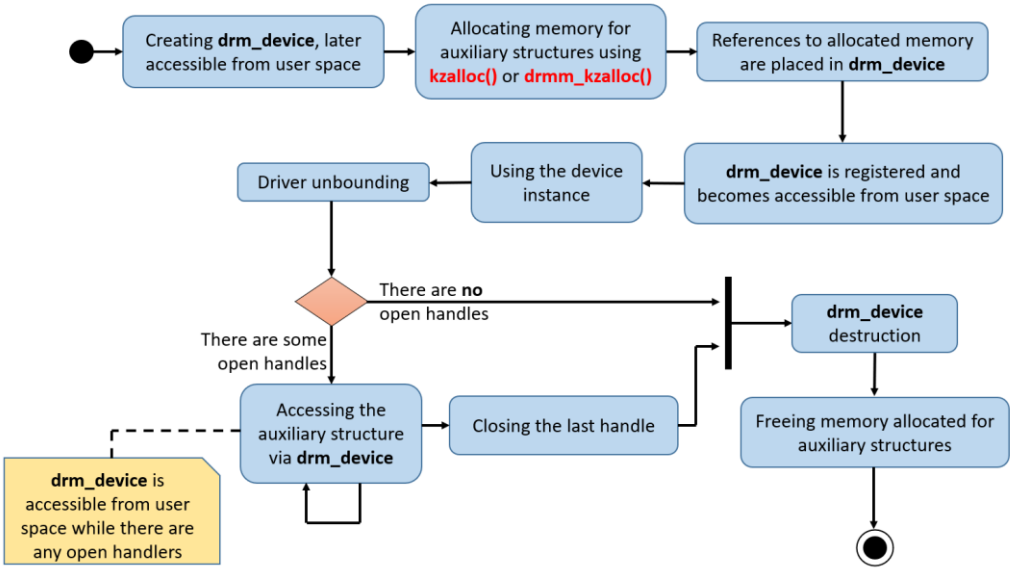

Fig. 1. A correct operation of a DRM driver.

Contrarily, using `devm_kmalloc()` in most of these cases is a mistake, since the release may occur ahead of time. The driver will still work, but sudden crashes will happen periodically. The reason is shown in Fig. 2. If the device still has a user after the driver detach, the user can try to access the structure previously allocated with `devm_kmalloc()` and released on driver detach. There are vulnerabilities of this type in the Linux kernel. Therefore, structures accessible from the user space after `drm_device` registration should not be allocated with devres-managed functions like `devm_kmalloc()`.

Accordingly, there is a documented restriction on the second argument of 5 functions that initialize preallocated DRM-specific structures – `drm_encoder_init`, `drm_connector_init`, `drm_connector_init_with_ddc`, `drm_crtc_init_with_planes`, and `drm_universal_plane_init` – namely, the second argument should not be allocated with `devm_kzalloc` or similar devres functions. This is a more obvious violation of the general rule.

3. Related Work

There is a wide variety of approaches and tools for bug finding in industrial software systems. Here, we limit the discussion to those most relevant to the Linux kernel [16].

3.1 Dynamic analysis

Dynamic analysis tools [17] typically look for a class of issues occurring in the running kernel. One example is Kernel Address Sanitizer (KASAN) [18] which can detect invalid memory accesses such as out-of-bounds and use-after-free errors.

The presence of the target errors in the code poses a risk of accessing freed memory, so they can be detected by KASAN. Indeed, it is mentioned in comments of some target error fixes accepted into the kernel and can be used to confirm the fix.

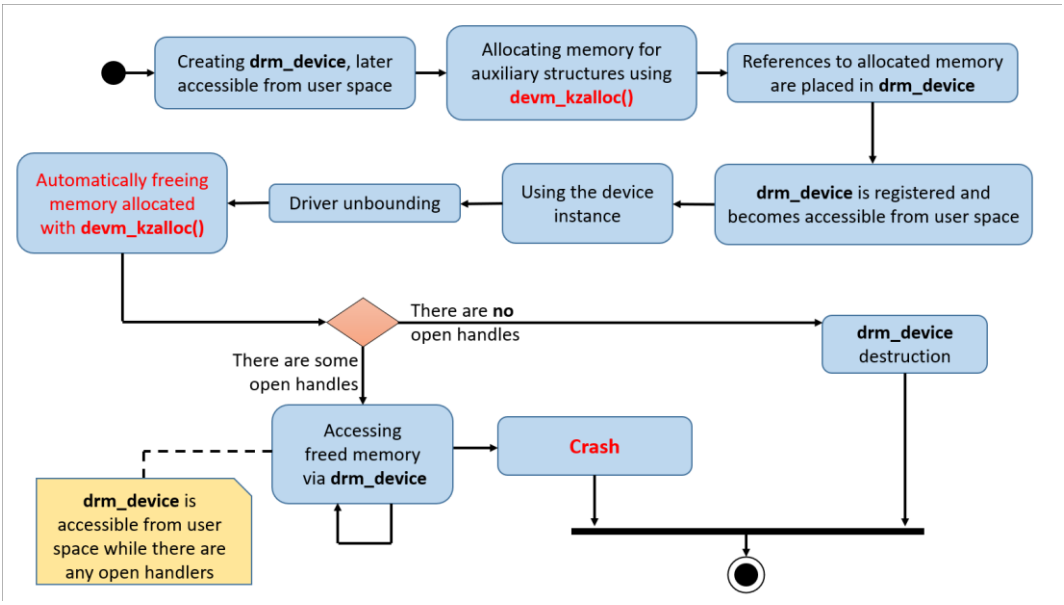

Fig. 2. An operation of a DRM driver with a target error.

However, dynamic analysis can only check the parts of the code that are reachable through the tests. Moreover, the target errors can cause races in very specific situations. Another drawback is the need for particular hardware to run certain drivers. These limitations effectively prevent dynamic analysis from reliably detecting the target errors.

3.2 Static analysis

Lightweight static analysis, such as abstract syntactic tree (AST) analysis and data-flow analysis (DFA), searches for defects in the program source code without execution. It can detect potential errors, vulnerabilities and non-compliance with standards at early stages of development, thereby saving time and resources. The tools most closely tied to the Linux kernel and used by its development community include Coccinelle, Sparse, and Smatch [16].

Coccinelle [13] is a program matching and transformation tool focused on patterns in source code structure. It operates on semantic patches – high-level patterns that resemble git patches but are abstracted with metavariables and ellipses. The tool employs a temporal logic (extended CTL [19]) to reason about a function's control flow. The strength of the tool is the relative ease of writing semantic patches for known patterns and its ability to generate patches automatically. However, its major limitation for our purposes is the lack of any data-flow reasoning and the very limited support for interprocedural analysis.

Sparse [20] is a source parser and analyzer that extends the C type system with kernel-specific annotations. These include address-space qualifiers to prevent mixing user and kernel pointers, and endianness markers to detect incorrect byte-order handling. Sparse also performs simple intraprocedural DFA for context-tracking [21] (i.e. matching context counters on entry and exit against annotations) and uses this together with locking annotations to warn about imbalanced or missing lock acquisition and release. Applying type-checking to the target errors would require manually annotating pointers along the data flow, which is more effort than manual inspection of the DRM drivers.

More conventional static analyzers, such as Smatch and Svace, use full-fledged data-flow analysis. Smatch [22] builds on Sparse by adding a "cross-function flow analysis" [23]. It is path-sensitive and can track a range or multiple values for a given variable. Svace [24], used by the Linux

Verification Center (LVC) [25, 26], employs both syntactic and flow analysis. It leverages a bottom-up technique: functions lower in the call graph are summarized using data-flow analysis and symbolic execution, and these summaries are then reused at call sites when analyzing functions higher in the call graph.

Although Smatch and Svace are effective at detecting various generic and kernel-specific errors, lightweight static analysis is generally unable to find the target errors in the complex scenarios characteristic of our target errors. The inherent trade-off between scalability and false positive rate forces the use of heuristics, which sacrifices soundness and precision.

3.3 Software model checking

Software model checking, or static verification, can be considered a heavyweight form of static analysis. The approach aims at thorough exploration of a program's state space, which allows detecting subtle errors, e.g. data races, and thus is used for critical system verification [1]. The drawbacks of the approach are high resource consumption and the frequent need for handwritten formal specifications.

Klever [2, 3] is a verification platform designed to automate the software model checking for industrial systems. For scalability, Klever decomposes large codebases into smaller, verifiable modules. Specifications needed for particular requirements or missing function bodies can be written in C [27]. The environment model, i.e. calls to the module, is provided based on the typical scenarios of device usage [28-29].

Klever together with the CPAchecker static verification tool have been used to find several hundred errors in the kernel, including memory safety violations [7], data races [30], and errors specific to Linux device drivers [3, 5]. To find the target errors, we need both write the specification for our DRM-specific rule and modify the underlying memory analysis to remember the allocating function for allocated memory regions.

4. Colored Symbolic Memory Graphs

To verify memory safety, CPAchecker [4] uses a shape analysis based on the Symbolic Memory Graph (SMG) domain [7-8] that first appeared in the Predator shape analyzer [9-10, 31].

The analysis represents a program memory state as a labeled bipartite graph of memory objects and symbolic values, and edges between them. A "has-value" edge from a memory object to a symbolic value means that the value is stored in the object (the offset and bitsize are labeled on the edge). A "points-to" edge from a symbolic value to a memory object means that the value points to the object (again, the offset from the start of the object is labeled on the edge).

To distinguish objects allocated in a certain way, we have introduced memory coloring for the analysis. The color of an allocated region is determined by the allocating function:

- *DRM* for the `drm_device` structures allocated by `drm_dev_alloc()` or `devm_drm_dev_alloc()`,
- *DEVM* for the `devm_kmalloc()`-allocated memory,
- and default (colorless) memory is allocated by all other functions.

Now, we can reformulate our rule in terms of the colored memory graphs: *DRM*-colored memory objects should not store pointers to *DEVM*-colored memory objects.

Simplified erroneous code is provided in Listing 1. A probe driver method allocates its own device structure and `drm_device` structure. When the analysis traverses the first line with a call to `devm_kzalloc()`, it adds a new heap object (shown as "DEVM" in Fig. 3) and a new symbolic value ("s1") that points to its start. As the pointer to the new allocation is stored in the variable `ldev`, a has-value edge `ldev` → s1 is added, too. As `devm_kzalloc()` is a colored function, the allocation gets the corresponding color (shown as red).

```
// allocate DRM device ddev with the given dev as parent
struct drm_device *ddev = drm_dev_alloc(&drv_driver, &pdev->dev);
// allocate the specific device with actually the same dev as parent
struct ltdc_device *ldev =
  devm_kzalloc(ddev->dev, sizeof(*ldev), GFP_KERNEL);
// the rule violation
ddev->dev_private = (void *)ldev;
// ldev may be accessed after it is released before ddev is released
```

Listing 1. Simplified erroneous code (before the patch)
from `drivers/gpu/drm/stm/drv.c`*, functions* `stm_drm_platform_probe` *and* `drv_load`*.*

The `ddev` initialization is analyzed in the same manner, with the new allocation colored DRM (shown as green). The result SMG (without the labels on the edges) can be seen in Fig. 3 on the left; unimportant parts (such as previous stack frames and global variables) are not shown. After the assignment in the last line, the SMG looks like in Fig. 3 on the right. Note that *DRM*-colored allocation now has a field (DRM → s1) that points to a *DEVM*-colored allocation (s1 → DEVM). When such an assignment happens, the analysis reports an error.

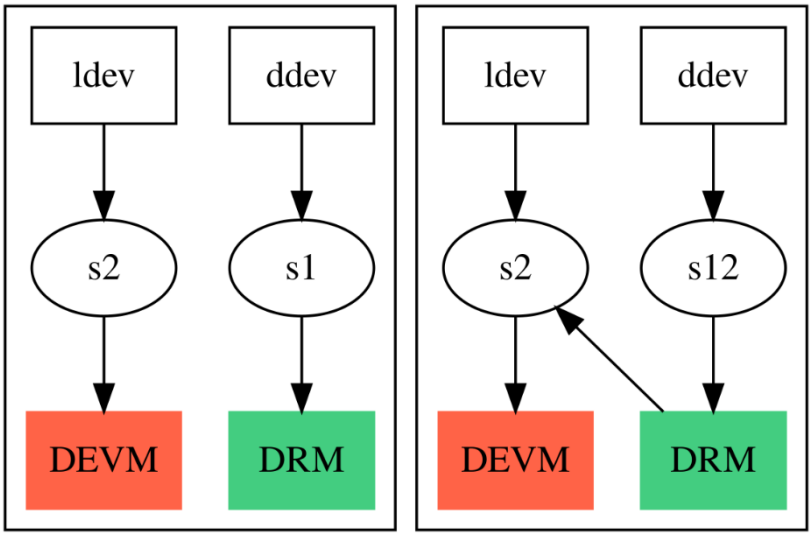

Fig. 3. Left: the symbolic memory graph for Listing 1 before assignment.
Right: the symbolic memory graph after assignment;
the presence of DRM → s2 → DEVM path is a violation of the proposed rule.

5. Specification of DRM subsystem in Klever

Klever decomposes the kernel into modules, with the result that CPAchecker runs on each of the modules separately [29]. This solves the issue of running heavy analysis on large code, but there is a problem with functions defined in other modules. Their bodies are not visible to CPAchecker, so during the verification of a module, it assumes that such a function is pure, i.e. it does not affect the analyzed code. If a function is important for finding the target error (e.g. it initializes a pointer important for the analysis), one has to write a *model* for it that CPAchecker will traverse instead of the original function.

Klever allows us to write such models for functions and replaces every call to the original function in the module with a call to the given model [28]. This is implemented using aspect-oriented programming [27]. Suppose there is a function `foo()` in the kernel code that we want to replace

with a model. Then we write a `ldv_foo()` function in the `.c` file, and we specify in the `.aspect` file that instead of `foo()` calls, `ldv_foo()` should be actually called.

The capability to replace a function with a model is also useful if it needs to be abstracted from insignificant details (simplified) or given a new feature, such as color for memory it allocates.

In our case, some memory allocation and releasing should be colored appropriately. To do this, a special "color" function was called in the bodies of the models (`ldv_color_drm_kmalloc()` or `ldv_color_devm_kmalloc()`, depending on the desired color). Models were also required for a number of imported functions in which bindings between structures were created. Basically, instead of a function initializing all fields of the structure, a model filled in several pointers, the value of which influenced the success of the target error search.

For the DRM subsystem, we have modelled the following functions in Klever:

- Devres specification. It includes `devm_kmalloc()` and its analogs (`devm_kzalloc()`, `devm_kcalloc()`, `devm_kmalloc_array`). Memory allocated with this function is automatically freed on driver detach. These functions paint memory in the color *DEVM*. If a reference to memory of the color *DEVM* appears in `drm_device`, an error is reported.
- `drm_device` managed resources specification. It includes memory allocation functions (`drmm_kmalloc()` and its analogs, which paint memory in the color *DRM*) and implementation of various ways to free it.
- Special functions used to allocate and deallocate `drm_device` and `drm_driver` memory and to initialize them: `drm_dev_alloc()`, `drm_dev_init()`, `drm_dev_release()`, etc. The function models responsible for initialization create references needed to find target errors.
- Models of functions that initialize structures used by DRM device. `drm_encoder_init()` for `srtuct drm_encoder`, `drm_universal_plane_init()` for `srtuct drm_plane`, and so on. In them, the structures are linked to `drm_device`, and if their memory was allocated incorrectly – i.e., with the *DEVM* color – an error is detected at the moment of storing a reference to such memory.
- Models of other functions that create and destroy links between structures: `drm_dev_put()`, `get_device()`/`put_device()`, `kref_init()`/`kref_put()`.

6. Evaluation

We applied our approach to 186 loadable DRM driver modules from `drivers/gpu/drm/` in Linux 5.10.238, targeting the ARM architecture with the `allmodconfig` build configuration.

The experiment was carried out with Klever, derived from version 4.0.1 [32], together with our fork of CPAchecker [33] on a machine with an Intel Core i7-11700 2.50GHz CPU (8 cores, 16 threads), 2x16 GB DDR4 RAM, and an SSD.

In total, verification has taken 10 h of CPU time (40 min of wall time). We have limited the CPAchecker verification tool to 270 s per module; it has used up to 1.3 GB for a module and consumed 4.2 h of CPU time in total. Table 1 details the results of the verification.

- 108 modules were verified as safe, with CPAchecker exhausting all reachable states without detecting any target or generic memory safety error.
- 33 modules resulted in verifier timeout, where CPAchecker did not complete within the allotted time.
- For 3 modules, CPAchecker stopped after encountering a recursive call.
- 17 driver modules were not verified due to a composition problem, where Klever was unable to compose a module to verify due to atypical module init or exit, or missing declarations.

The analysis reported 25 modules as unsafe:

- 6 target errors;
- 1 generic memory error, specifically a non-target use-after-free;
- 18 false alarms for generic memory errors. These were primarily due to analysis imprecision (e.g., inability to calculate a dereferenced address). One of them was caused by inline assembler code in the sources.

Table 1. Verification results for the 186 Linux 5.10.238 DRM drivers.

Verdict	Count	%
Unsafe	**25**	**13**
target error	6	3.2
use-after-free	1	0.5
false alarm	18	9.7
Safe	**108**	**58**
Unknown	**53**	**28**
verifier timeout	33	17.7
recursion in module	3	1.6
composition problem	17	9.1
Total:	***186***	***100***

6.1 Estimating Missed Errors with Coccinelle

We used Coccinelle to estimate the amount of the target errors in the kernel code and to assess the false negative rate of our approach. As the presence of an error-prone pattern implies the need to fix multiple files in a module, we count the reported modules instead of matches for Coccinelle.

Following the documented restriction, we wrote the **arg** rule illustrated in Listing 2. It finds a devres-managed memory pointer passed as the second argument to one of the 5 drm-init functions with the documented restriction discussed in Section 2. We used Coccinelle to find 5 more functions – `drm_writeback_connector_init`, `drm_crtc_init`, `drm_plane_init` `drm_bridge_connector_init`, and `drm_simple_encoder_init` – that are simple wrappers to those and should thus have the same restriction applied. While we can continue to elaborate the rule, in practice we do not expect much more alarms.[1]

```
devm =@p devm_kzalloc(...);
...
drm_crtc_init_with_planes(@q e,<+...devm...+>,...)
```

*Listing 2. A snippet of the **arg** rule for DEVM-allocated second argument to a `drm_*_init` function.*

All target errors identified by Klever involved an assignment of DEVM-allocated memory pointer to the `dev_private` field of a DRM-allocated `drm_structure`. Moreover, 3 modules reported by Klever were not reported by the **arg** rule. This motivated a second Coccinelle rule, **field** (Listing 3) designed to detect assignments of a DEVM-allocated pointer to a field of a DRM-allocated structure. See more elaborated rules as submitted to the kernel in [14].

[1] There are a considerable number of cases where a DEVM-allocated pointer is first assigned to another local variable which is later passed to a drm-init function. However, these occur in modules already reported by the simpler rule.

```
drm = drm_dev_alloc(...);
...
devm = devm_kzalloc(...);
...
drm->f =@p <+...devm...+>;
```

*Listing 3. A snippet of the **field** rule for assigning a DEVM-allocated pointer to a DRM-allocated **field**.*

Table 2 presents a per-module comparison between the findings from Klever and the two Coccinelle rules, 27 modules in total. The column Klever shows the outcomes of our verification runs, while Coccinelle/field and Coccinelle/arg mark the modules in which the corresponding rule found violations. All 27 modules are reported by one of the Coccinelle rules; notably, only 4 are reported by both. Although our analysis targeted Linux 5.10.238, many of the bugs are still present in recent versions (6.17). In the Klever column, the outcomes are encoded as follows:

- target error – analysis reported a violation of the color rule (true positive);
- non-target alarm – a reported generic memory error turned out to be a false alarm;
- safe – full state-space exploration without detecting violations of the color rule or generic memory safety;
- unknown (timeout) – CPAchecker exceeded the allocated CPU time;
- unknown (recursion) – CPAchecker stopped on encountering a recursive function call;
- unknown (oom) – out-of-memory during module composition;
- unknown (comp. iss.) – compilation issue during module composition;
- unknown (arch) – module not included in the ARM build.

6.2 Error classification

True Positives. As shown in Table 2, Klever successfully identified target errors in 6 modules. All 6 modules were also reported by at least one Coccinelle rule: 5 modules were reported by the **field** rule and 3 by the **arg** rule. Notably, the assignment to the field in `stm/stm-drm` module was not reported by the **field** rule because one of the allocations happens in another function, and handling such interprocedural cases is limited in Coccinelle.

True Negatives. We did not assess true negatives.

False Positives. We found no false positives among the 6 target errors reported by Klever.

False Negatives. Klever missed a violation in 21 modules reported by Coccinelle: 8 reported by the **field** rule and 15 modules reported by the **arg** rule, respectively. Of these, 11 misses can be attributed to the limitations of our approach (timeouts, recursion, oom, comp. arch, non-target alarms). For the 10 modules reported as safe, the coverage appears to be lacking, as the relevant functions were not reached. This suggests the need to refine or add the specifications for the modules so the analysis can reach the DRM functions.

7. Conclusion

We have discussed a subtle use-after-free error in Linux DRM drivers that originates from misusing managed memory allocation for device structures. To find such errors, we proposed a coloring rule, introduced such coloring to the SMG analysis in the CPAchecker verification tool, and wrote specifications for the respective functions of the DRM subsystem.

For the specification and component-wise verification of 186 modules in the DRM subsystem, we have used the Klever verification platform. Klever was able to carry out the verification for 169 modules and reported 25 of them as unsafe. Among these, 6 modules contain a target error, and 1 module contains a generic memory error (use-after-free).

Moreover, we developed two Coccinelle rules: **arg**, which finds violations of the documented restriction, and **field**, which was motivated by the pattern in the errors found by Klever. While **arg**

reports errors in 18 modules, **field** reports 9 additional modules. Together, these approaches provide complementary coverage and demonstrate the effectiveness of combining lightweight and heavyweight methods.

Future work. We plan to continue submitting patches for the discovered errors. We also intend to refine and extend the specifications to improve the coverage across DRM modules.

Table 2. Comparison of target errors found by Klever and Coccinelle.

DRM module	Klever	Coccinelle	
arc/arcpgu	non-target alarm		arg
arm/hdlcd	safe	field	arg
arm/mali-dp	unknown (timeout)	field	
atmel-hlcdc/atmel-hlcdc-dc	non-target alarm		arg
fsl-dcu/fsl_dcu_drm	**target error**	**field**	
ingenic/ingenic-drm	non-target alarm		arg
lima/lima	**target error**	**field**	
meson/meson-drm	unknown (recursion)	field	arg
meson/meson_dw_hdmi	safe		arg
msm/msm	unknown (oom)		arg
panfrost/panfrost	unknown (arch)	field	
pl111/pl111_drm	safe	field	
rcar-du/rcar_du	non-target alarm	field	
rockchip/rockchip_drm	unknown (timeout)	field	
shmobile/shmob_drm	**target error**	**field**	**arg**
sti/sti-drm	safe		arg
stm/stm-drm	**target error**		**arg**
sun4i/sun4i-drm	unknown (timeout)	field	
5 modules: sun4i/sun4i-{backend,drm-hdmi,tcon,tv} and sun4i/sun8i-mixer	5 safe		5 arg
tilcdc/tilcdc	**target error**	**field**	**arg**
tve200/tve200	**target error**	**field**	
vc4/vc4	unknown (timeout)		arg
zte/zx_drm	safe		arg
Modules with target errors:	**6 (22%)**	**13 (48%)**	**18 (67%)**

References

[1]. E.M. Clarke, T.A. Henzinger, H. Veith, and R. Bloem. Handbook of Model Checking. Springer International Publishing, Cham. 2018. DOI: 10.1007/978-3-319-10575-8.

[2]. I.S. Zakharov, M.U. Mandrykin, V.S. Mutilin, E.M. Novikov, A.K. Petrenko, and A.V. Khoroshilov. Configurable toolset for static verification of operating systems kernel modules. Programming and Computer Software, vol. 41, no. 1. 01.01.2015. pp. 49–64. DOI: 10.1134/S0361768815010065.

[3]. I. Zakharov, E. Novikov, and I. Shchepetkov. Klever: Verification Framework for Critical Industrial C Programs. 2023. DOI: 10.48550/arXiv.2309.16427.

[4]. D. Baier, D. Beyer, P.-C. Chien, M.-C. Jakobs, M. Jankola, M. Kettl, N.-Z. Lee, T. Lemberger, M. Lingsch-Rosenfeld, H. Wachowitz, and P. Wendler. Software Verification with CPAchecker 3.0: Tutorial and User Guide. Formal Methods. 2025. pp. 543–570. DOI: 10.1007/978-3-031-71177-0_30.

[5]. V.S. Mutilin, E.M. Novikov, and A.V. Khoroshilov. Analysis of typical faults in Linux operating system drivers. Trudy ISP RAN/Proc. ISP RAS, 2012, vol. 22, pp. 349–374 (in Russian). DOI: 10.15514/ispras-2012-22-19.

[6]. Found Bugs by Klever. [Online]. Available at: https://github.com/ldv-klever/klever?tab=readme-ov-file#found-bugs, accessed 09.09.2025.

[7]. A.A. Vasilyev. Static verification for memory safety of Linux kernel drivers. Trudy ISP RAN/Proc. ISP RAS, 2018, vol. 30, issue 6, pp. 143–160. DOI: 10.15514/ISPRAS-2018-30(6)-8.

[8]. A.A. Vasilyev and V.S. Mutilin. Predicate Extension of Symbolic Memory Graphs for the Analysis of Memory Safety Correctness. Programming and Computer Software, vol. 46, no. 8, 01.12.2020, pp. 747–754. DOI: 10.1134/S0361768820080071.

[9]. K. Dudka, P. Peringer, and T. Vojnar. Byte-Precise Verification of Low-Level List Manipulation. in F. Logozzo and M. Fähndrich (eds). Static Analysis. Springer Berlin Heidelberg, Berlin, Heidelberg. 2013. pp. 215–237. DOI: 10.1007/978-3-642-38856-9_13.

[10]. K. Dudka, P. Muller, P. Peringer, V. Šoková, and T. Vojnar. Algorithmic Details behind the Predator Shape Analyser. 2024. DOI: 10.48550/arXiv.2403.18491.

[11]. DRM Internals – The Linux Kernel documentation. [Online]. Available at: https://www.kernel.org/doc/html/latest/gpu/drm-internals.html, accessed 29.09.2025.

[12]. E. Orlova. [PATCH v4] drm/stm: Avoid use-after-free issues with crtc and plane. [Online]. Available at: https://lore.kernel.org/all/20240216125040.8968-1-e.orlova@ispras.ru/, accessed 06.10.2025.

[13]. J.L. Lawall and G. Muller. Coccinelle: 10 Years of Automated Evolution in the Linux Kernel. USENIX Annual Technical Conference. 2018. [Online]. Available at: https://www.usenix.org/system/files/conference/atc18/atc18-lawall.pdf, accessed 06.10.2025.

[14]. O. Petrov. [PATCH] cocci: drm: report devm-allocated arguments and fields. [Online]. Available at: https://lore.kernel.org/all/20250924140126.23027-1-o.petrov@ispras.ru/, accessed 24.09.2025.

[15]. E. Orlova. drm/stm: Avoid use-after-free issues with crtc and plane. [Online]. Available at: https://web.git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=19dd9780b7ac673be95bf6fd6892a184c9db611f, accessed 15.07.2024.

[16]. M. Schmitt. Linux kernel device driver testing. How are device drivers being tested? Master's Thesis, Institute of Mathematics and Statistics, University of São Paulo, São Paulo. 17.10.2022. DOI: 10.11606/D.45.2022.tde-30112022-152524.

[17]. A. Konovalov. Sanitizing the Linux kernel: On KASAN and other Dynamic Bug-finding Tools. Linux Security Summit Europe. 2022. [Online]. Available at: https://www.youtube.com/watch?v=KmFVPyHyfqQ.

[18]. K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov. AddressSanitizer: A Fast Address Sanity Checker. USENIX ATC 2012. 2012. [Online]. Available at: https://www.usenix.org/conference/usenixfederatedconferencesweek/addresssanitizer-fast-address-sanity-checker, accessed 06.10.2025.

[19]. J.L. Lawall, J. Brunel, N. Palix, R.R. Hansen, H. Stuart, and G. Muller. WYSIWIB: A declarative approach to finding API protocols and bugs in Linux code. DSN'09 – The 39th Annual IEEE/IFIP International Conference on Dependable Systems and Networks. 2009. pp. 43–52. DOI: 10.1109/DSN.2009.5270354.

[20]. N. Brown. Sparse: a look under the hood. 2016. [Online]. Available at: https://lwn.net/Articles/689907/, accessed 06.10.2025.

[21]. L. Torvalds. Sparse 'context' checking. [Online]. Available at: https://lwn.net/Articles/109066/, accessed 18.09.2025.

[22]. N. Brown. Smatch: pluggable static analysis for C. 22.06.2016. [Online]. Available at: https://lwn.net/Articles/691882/, accessed 06.10.2025.

[23]. D. Alden. Finding locking bugs with Smatch. 11.06.2025. Write-up of Dan Carpenter's talk at Linaro Connect 2025. [Online]. Available at: https://lwn.net/Articles/1023646/, accessed 06.10.2025.

[24]. A. Belevantsev, A. Borodin, I. Dudina, V. Ignatiev, A. Izbyshev, S. Polyakov, E. Velesevich, and D. Zhurikhin. Design and Development of Svace Static Analyzers. 2018 Ivannikov Memorial Workshop (IVMEM). 2018. pp. 3–9. DOI: 10.1109/IVMEM.2018.00008.

[25]. Linux Verification Center — Static Analysis (in Russian). [Online]. Available at: https://portal.linuxtesting.ru/activity.html#menu3, accessed 29.09.2025.

[26]. Found by Linux Verification Center (linuxtesting.org) with SVACE. [Online]. Available at: https://web.git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/log/?qt=grep&q=Found+by+Linux+Verification+Center+(linuxtesting.org)+with+SVACE, accessed 29.09.2025.

[27]. E.M. Novikov. An approach to implementation of aspect-oriented programming for C. Programming and Computer Software, vol. 39, no. 4. 07.2013. pp. 194–206.
[28]. I.S. Zakharov, V.S. Mutilin, and A.V. Khoroshilov. Pattern-based environment modeling for static verification of Linux kernel modules. Programming and Computer Software, vol. 41, no. 3. 05.2015. pp. 183–195. DOI: 10.1134/S036176881503007X.
[29]. I. Zakharov and E. Novikov. Compositional Environment Modelling for Verification of GNU C Programs. 2018 Ivannikov ISPRAS Open Conference. 2018. pp. 39–44. DOI: 10.1109/ISPRAS.2018.00013.
[30]. P.S. Andrianov. Analysis of Correct Synchronization of Operating System Components. Programming and Computer Software, vol. 46, no. 8. 01.12.2020. pp. 712–730. DOI: 10.1134/S0361768820080022.
[31]. Predator. [Online]. Available at: https://www.fit.vut.cz/research/group/verifit/public/tools/predator/, accessed 29.09.2025.
[32]. Klever 4.0.1. [Online]. Available at: https://github.com/ldv-klever/klever/tree/v4.0.1/, accessed 18.03.2025.
[33]. CPAchecker 702bc1a. [Online]. Available at: https://github.com/ldv-klever/cpachecker/commit/702bc1a36f663d0e1bac13e6c6752e61828e6ac8, accessed 21.03.2025.

Информация об авторах / Information about authors

Екатерина Михайловна ОРЛОВА – студентка магистратуры факультета вычислительной математики и кибернетики МГУ, лаборант Института системного программирования РАН. Сфера научных интересов: статический анализ и верификация ядра Linux.

Ekaterina Mikhaylovna ORLOVA – Master's student at the Faculty of Computational Mathematics and Cybernetics of Lomonosov Moscow State University (MSU), lab assistant at the Institute for System Programming of the RAS. Research interests: static analysis and verification of the Linux kernel.

Антон Александрович ВАСИЛЬЕВ – младший научный сотрудник Института системного программирования им. В.П. Иванникова РАН. Сфера научных интересов: статическая верификация и анализ программ.

Anton Aleksandrovich VASILYEV – junior researcher at the Ivannikov Institute for System Programming of the RAS. Research interests: static verification, software model checking, static program analysis.

Олег Максимович ПЕТРОВ – аспирант и стажёр-исследователь Института системного программирования им. В.П. Иванникова РАН. Сфера научных интересов: статическая верификация и анализ исходного кода программ, delta debugging.

Oleg Maximovich PETROV – postgraduate student and intern researcher at the Ivannikov Institute for System Programming of the RAS. His research interests include software model checking, static program analysis, delta debugging.