Static verification for memory safety of Linux kernel drivers*

A.A. Vasilyev <vasilyev@ispras.ru>

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

Abstract. Memory errors in Linux kernel drivers are a kind of serious bugs that can lead to dangerous consequences but such errors are hard to detect. This article describes static verification that aims at finding all errors under certain assumptions. Static verification of industrial projects such as the Linux kernel requires additional effort. Limitations of current tools for static verification disallow to analyze the Linux kernel as a whole, so we use a simplified automatically generated environment model. This model introduces inaccuracy, but provides ability for verification. In addition, we allow absent definitions for some functions which results in incomplete ANSI C programs. The current work proposes an approach to reveal issues with memory usage in such incomplete programs. Our static verification technique is based on Symbolic Memory Graphs (SMG) with extensions aiming to reduce a false alarm rate. We introduced an on-demand memory conception for simplification of kernel API models and implemented this conception in static verification tool CPAchecker. Also, we changed precision of a CPAchecker memory model from bytes to bits and supported structure alignment similar to the GCC compiler. We implemented the predicate extension for SMG to improve accuracy of the analysis. We verified of Linux kernel 4.11.6 and 4.16.10 with help of the Klever verification framework with CPAchecker as a verification engine. Manual analysis of warnings produced by Klever revealed 78 real bugs in drivers. We have made patches to fix 33 of them.

Keywords: shape analysis; static verification; symbolic memory graphs; memory model.

DOI: 10.15514/ISPRAS-2018-30(6)-8

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

1. Introduction

Operating system kernels are often written in the C programming language. This language is portable and effective, but unfortunately it is not memory safe. Memory issues can lead to vulnerabilities or unpredictable failures. Common methods such as testing are unable to find all problems. A probable solution to get an evidence of

* The research was supported by RFBR grant 18-01-00426

satisfiability of safety properties is formal methods and there are results of comprehensive formal verification of the seL4 microkernel [1]. However formal methods generally require a whole program and a complete model of its environment to produce an appropriate verdict. For example, Microsoft developed Static Driver Verifier (SDV) [2] to improve Microsoft Windows stability. SDV contains models of the kernel and drivers' environment, and over 60 API usage rules.

The Linux kernel is important open source software. There are many research and industrial projects for improving kernel quality by verification, testing, bug hunting, fuzzing and error reports. Coverity [3], Saturn [4], DDVerify [5], Coccinelle [6], Linux Driver Verification [7] are projects which work on improving Linux stability.

This article considers operating system kernel drivers with automatically generated environment models as a target for approbation of a memory verification technology. Main contributions of the paper are connected with extensions of an existed static memory verification approach to be able to perform Linux kernel drivers verification, which are described in Section 4.

2. Linux driver verification

The Linux kernel represents an industrial code base with more than 10 million lines of drivers' code. A distinctive feature of Linux is instability of internal interfaces. A high speed of changes with a distributed development process requires an efficient bug finding strategy.

The research of faults in Linux operating system drivers divides errors into typical and specific [8]. Specific faults in drivers are described as connected with hardware and not applicable to other drivers. Typical faults can be specified by some rule which is true for all or some group of drivers. Typical faults are further divided into:

- Linux specific faults, which correspond to rules of correct usage of the Linux kernel API;
- races and deadlocks, which are related with parallel execution;
- generic problems, which are common for C programs such as null pointer dereference, integer overflow, etc.

Authors show that 29.2% of typical errors fixed in stable branches of the Linux kernel are generic problems. Statistics of memory problems corresponding to all generic faults is shown in Table. 1.

Table. 1. Ratio of memory problems corresponding to all generic faults

Type	Percentage
NULL pointer dereference	30.4%
Resource: memory leak, double free, use after free	23.5%
Buffer overflow	7.8%

Uninitialized: uninitialized pointer free, write to unallocated memory	5.9%
Total	67.6%

This information shows that the main part of generic faults match memory errors. We suggest to improve situation with memory safety of the Linux kernel with help of static verification.

The Linux Driver Verification project (LDV) [7, 9, 10] aims at performing automatic static driver verification and reporting detected problems. It provides a static verification framework called Klever [11] for Linux kernel verification including automated environment model generation [12, 13], rules of correct kernel API usage, interfaces for storing and visualization of verification results [14]. As a verification engine *Klever* includes the CPAchecker [15] verification tool.

In this work, we added several extensions into the CPAchecker verification tool for memory safety verification and improved *Klever* environment models to check memory safety for drivers of the Linux kernel. We have made experimental evaluation on drivers of Linux kernel 4.11.6 and 4.16.10, analyzed all memory safety problems reported by the verification tool and classified them into bugs and false alarms. We prepared bug reports and fixes to the newest kernel versions. Regarding false alarms, we conclude that automatic environment generation heavily affects verification results and requires further improvement.

3. Symbolic memory graphs

The symbolic memory graph (SMG) algorithm [16] is a kind of shape analysis. It works with directional graph representation of a memory state. Nodes are used for symbolic values, memory regions and abstracted structures representation. Edges show references between nodes and are divided into *point-to edges* for pointers and *has-value edges*. Each edge and node in SMG has a set of *labels* representing size, offset and allocation status. One symbolic memory graph with abstractions can represent several memory states called concrete memory images. Set of all concrete memory images for SMG *G* is denoted as *MI(G)*.

Our SMG implementation in CPAchecker keeps mapping between global, stack variables and memory regions. Also, it tracks mapping between symbolic and concrete values. A memory graph is modified in correspondence with analyzed source code.

Detailed description of operations on SMG can be found at [16]. Here we provide a brief overview.

3.1. Read/write data reinterpretation

This operation emulates memory modification with validity checks.

Modifications: A level of details for a memory model allows to take into account such low level interpretation as unions and provide facility for reinterpretation values even on the same offset with different types.

Algorithm supports partial values overwrite if memory for corresponding field intersects. For example:

```
1   union {
2       int i;
3       char c;
4   } u;
5   u.i = 10;
6   u.c = 'A';
```

After line 5 union *u* will contain integer value 10 with size 4 byte, but after line 6 from this union we are able to read 1 byte char 'A' or an undefined 4 byte integer value.

Checks: For these operations, the algorithm performs checks against null pointer dereference and read/write within object bounds.

3.2. Join of SMGs

This operation is central one for abstraction and decision whether a current memory state is covered by another one and vice versa, so the algorithm can drop one of the states. It takes as input 2 SMGs G_1, G_2, compares their concrete memory images and produces join status with summarization SMG G. If $MI(G_1) \square MI(G_2)$ and $MI(G_1) \square MI(G_2)$ then SMGs are semantically incomparable and their join is undefined.

Algorithm travels through pair of SMGs and tries to join nodes. It is possible if nodes have same sizes, validity, and special conditions for join with abstract lists. Abstract lists are joinable if they have same head, previous and next fields offsets, a join result will have a number of elements equal to minimum from originals. Also, a result of a join region with an abstract list become an abstract list. It is possible to insert an empty list abstraction at any correct position in a graph to increase opportunity of correct join.

3.3. Summarizing sequences of objects to list abstraction

This operation comes from the shape analysis theory. Ideas for different abstractions could be found in Sagiv work [17]. SMG uses single and double linked lists as abstractions.

The algorithm discovers sequences of neighboring objects which could be considered as list entry candidates and then sequentially adds them into one abstract list and increases its size. An abstraction size is considered as number of elements necessarily present in the abstraction.

3.4. Abstract list materialization

Materialization is an operation for unfolding the abstraction to memory regions on write/read from abstracted regions.

3.5. Checking equality and inequality of values and pointers

The algorithm supports incomplete checking for equality and inequality of values and pointers. In some cases, it can fail with different point-to edges from one abstracted region.

The tool performs stack variables cleaning on function exit and checking for dangling pointers to allocated memory, which helps identify memory leak errors.

Let's consider analysis of a simple example:

```
void main() {
    void *array;
    long b = 2;
    long c = 3;
    array = calloc(1, 16);
    memcpy(&array[4], &b, 4);
    memcpy(&array[5], &c, 4);
}
```

Steps of the algorithm are shown in figs 1-6 below.

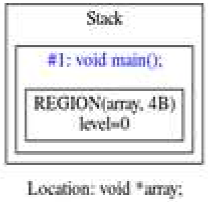

Fig. 1. Modification: allocate the 4 byte memory region on stack for pointer array

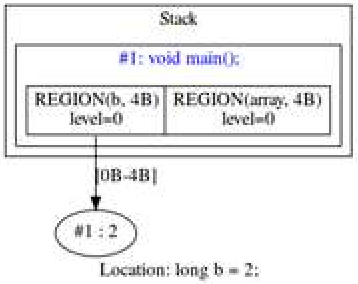

Fig. 2. Modification: allocate the 4 byte memory region on stack for variable b and assign it a new value #1 with explicit value 2

Check: a memory region size is sufficient for the assigned value.

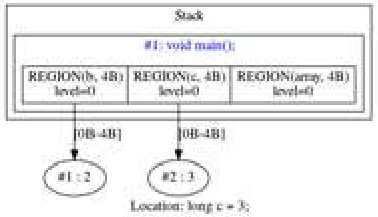

Fig. 3. Modification: allocate the 4 byte memory region on stack for variable c and assign it a new value #2 with explicit value 3

Check: a memory region size is sufficient for the assigned value

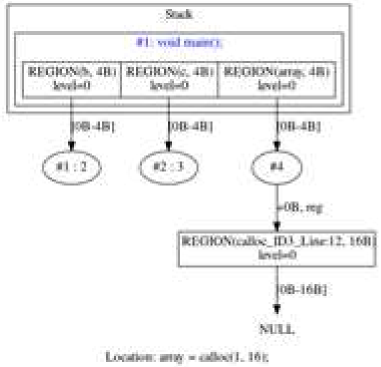

Fig. 4. Modification: allocate the 16 byte memory region on heap (mark it by tag calloc_ID3), fill it by NULL values, and assign to array a new point-to-value #4 which points to 0 offset of region calloc_ID3

Check: a region memory size is sufficient for the assigned value.

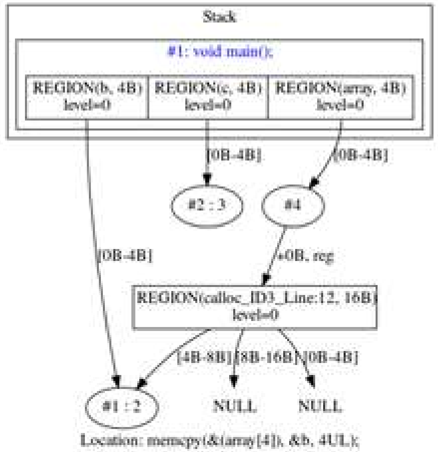

Fig. 5. Modification: assign 4 byte value #1 by offset 4 of region calloc_ID3

Check: dereference and assignment are done within allocated memory.

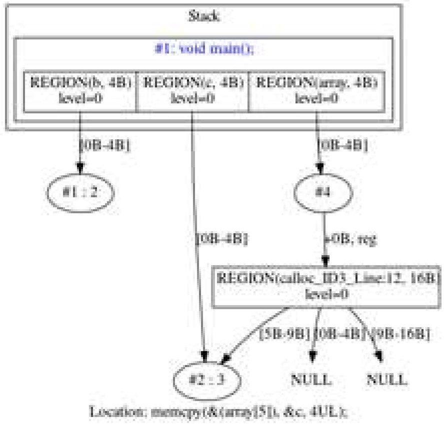

Fig. 6.Modification: assign 4 byte value #2 by offset 5 of region calloc_ID3, remove intersecting values, so value at offset 4 of region calloc_ID3 is not defined

Check: dereference and assignment are done within allocated memory.

4. Extensions for SMG

4.1. Bit precise model

The Linux kernel operates on structures with bit fields. We implemented bit fields in CPAchecker and switched SMG operations granularity from byte to bit precision. Also, we simulate structure alignment corresponding to GCC compiler memory usage.

4.2. Predicate extension

We implemented tracking of predicates over symbolic and concrete values stored in a memory graph. This feature allows filtering infeasible paths. On branching we perform a predicate satisfiability check to decide which branch is feasible. In addition, this method allows us to extend memory region over-read and overwrite checks for arrays using an error predicate check on a data reinterpretation operation.

4.3. On-demand memory

We consider the Linux kernel as trusted code and drivers as untrusted code in following sense: all structures provided to drivers by the kernel core are controlled by the kernel. We assume that the kernel recursively initializes all structure/union fields so drivers do not require to manage these structures. We supported the current point of view as the *on-demand memory (ODM)* concept within CPAchecker.

Allocation of *ODM* is made by special function *void* ext_allocation()*. A returned pointer allows any recursive dereference by any offset and distinguishes values by list of offsets and pointers from the original pointer. Additionally, any explicitly allocated memory which is reachable from on-demand memory is considered as automatically freed on program exit.

SMG implementation of ODM is done by special labels on memory regions and following behavior rules:

- any first dereference (read/write/free) of ODM pointers assumes that they are not NULL, ODM function pointers are an address to a pure function which returns nondeterministic value for non-pointer return value types or a pointer to ODM for pointer return value types;
- read memory:
  - read without previous read or write:
    - ✓ valid for any offset;
    - ✓ returns nondeterministic values for non-pointer types and a pointer to ODM for pointer types;
  - read after write:
    - ✓ valid for any offset;
    - ✓ returns values that were written by write;
  - read after read:
    - ✓ valid for any offset;
    - ✓ returns the same values that were read previously;
  - read after free is not valid.
- write memory:
  - write:
    - ✓ valid for any offset;
    - ✓ store new values in memory;
  - write after free is not valid.
  - free memory:
  - pointers to ODM are not subjected for memory leaks;
  - pointers to regular memory which are contained in ODM are not subjected for memory leaks;
  - free of any ODM offset is valid;
  - double free of ODM with the same offset is not valid;
  - read or write of freed ODM is not valid.

5. Configurable Program Analysis

The theory of SMG is implemented as Configurable Program Analysis (CPA) [18] within CPAchecker under the name SMGCPA.

Common CPA has *abstract domain*, *transfer*, *merge* and *stop* operators:

- *abstract domain* describes abstract states which represent sets of concrete states of the program;
- *transfer* gets one state and a control flow operation as input and returns all states which appears after applying the operation on the original state;
- *merge* takes 2 states as input and tries to combine them into one;
- *stop* identifies when one state is covered by others and decides whether it is required to continue analysis with a current state.

CPAchecker allows to combine different CPAs into one composite CPA. It works with a composite state which includes states of each involved CPAs. *Merge* produces a Cartesian product of separate analyses *merge* results.

SMGCPA fits into CPA conception with the following operators:

- abstract domain has SMG states as abstractions;
- transfer performs SMG transformations corresponding to a current control flow operation;
- merge tries to join SMGs from states and returns new SMG if join is successful;
- stop checks whether $MI(G1) \subseteq MI(G2)$ or a state has memory issues.

6. Experimental results

Experiments were performed with the help of *Klever* static verification framework [11], that is a part of LDV project [7]. *Klever* automatically generates environment models for each separate driver.

We checked memory safety for drivers of Linux 4.11.6 and Linux 4.16.10.

Table 2 and 3 present results of experiments on 6224 and 5215 generated verification tasks for Linux 4.11.6 and 4.16.10 respectively. We used the 15 minutes CPU time limit for each verification task. We performed manual analysis of 561 Unsafe verdicts for Linux 4.11.6 and 266 Unsafe verdicts for Linux 4.16.10 and classified 49 Unsafes as real memory bugs and 512 as false alarms for Linux 4.11.6 and 29 real bugs and 237 false alarms for Linux 4.16.10.

Table 2. Evaluation on drivers of Linux 4.11.6

Safe	1560		
Unknown	4023	Timeouts	2594
		Others	1429
Unsafe	641	Bugs	49
		False alarms	512
		Without marks	80

Table 3. Evaluation on drivers of Linux 4.16.10

Safe	2093		
Unknown	2830	Timeouts	1293
		Others	1537
Unsafe	292	Bugs	29
		False alarms	237
		Without marks	26

Causes of false alarms (512 on 4.11.6 and 237 on 4.16.10) are the following.

- Imprecise environment models (258 + 96);

Automatically generated environment models could mistakenly provide wrong driver initialization and cleanup. Also, some emulated functions are imprecise for correct proof of memory safety.

- Absent function (139 + 58);

Current environment models do not contain functions imported from other drivers. This leads to false alarms if undefined functions are important for memory safety properties.

- Require predicate SMG (83 + 43);

These false alarms are connected mainly with arithmetic operations on unknown values. We expect that some common patterns used in software could be emulated by additional predicates description, e.g. bitwise AND on unsigned values provide result value less or equal to operands and this is common check for array dereference in the Linux kernel.

- SMG problems (13 + 32);

Problems with analysis such as missed values after merge and wrong assumptions about loop invariants.

- Verification task generator problems (10 + 5);

The verification task generator omits information about packed pragma for structures at final source files. Sometimes it provides less allocation sizes than unpacked structure sizes.

- Unknown allocation sizes (9 + 3);

If SMG can not derive explicit values for allocation sizes it uses a predefined value, which may be less than required.

The list of reported bugs is presented in Table 4. Not all bugs were reported, because some of them were detected in old unsupported drivers or were already fixed.

Table 4. Bugs in Linux 4.11.6 reported to Linux Kernel Mailing List (https://lkml.org/lkml)

Message ID	Subject
2017/8/1/615	Buffer overread in pv88090-regulator.ko
2017/8/10/693	hwmon:(stts751) buffer overread on wrong chip
2017/8/10/597	dmaengine: qcom_hidma: avoid freeing an uninitialized pointer
2017/8/15/322	ASoC: samsung: i2s: Null pointer dereference on samsung_i2s_remove
2017/8/10/535	i2c: use release_mem_region instead of release_resource
2017/8/16/493	mtd: plat-ram: Replace manual resource management by devm
2017/8/11/366	mISDN: Fix null pointer dereference at mISDN_FsmNew
2017/8/10/522	parport: use release_mem_region instead of release_resource
2017/8/11/368	video: fbdev: udlfb: Fix use after free on dlfb_usb_probe error path
2017/8/10/550	dvb-usb: Add memory free on error path in dw2102_probe()
2017/8/16/345	udc: Memory leak on error path and use after free

Table 5. Bugs in Linux 4.16.10 reported to Linux Kernel Mailing List (https://lkml.org/lkml)

Message ID	Subject
2018/7/6/412	uwb: hwa-rc: fix memory leak at probe
2018/7/18/551	media: dm1105: Limit number of cards to avoid buffer over read
2018/7/23/964	media: dw2102: Fix memleak on sequence of probes
2018/7/6/389	video: goldfishfb: fix memory leak on driver remove
2018/7/23/944	firmware: vpd: Fix section enabled flag on vpd_section_destroy
2018/7/27/764	misc: ti-st: Fix memory leak in the error path of probe()
2018/7/27/503	media: vimc: Remove redundant free
2018/7/23/949	gpio: ml-ioh: Fix buffer underwrite on probe error path
2018/7/27/769	can: ems_usb: Fix memory leak on ems_usb_disconnect
2018/7/27/661	regulator: tps65217: Fix NULL pointer dereference on probe
2018/7/27/655	scsi: 3ware: fix return 0 on the error path of probe
2018/7/27/772	net: mdio-mux: bcm-iproc: fix wrong getter and setter pair
2018/7/23/1020	HID: intel_ish-hid: tx_buf memory leak on probe/remove
2018/8/6/572	pinctrl: axp209: Fix NULL pointer dereference after allocation
2018/7/27/508	media: davinci: vpif_display: Mix memory leak on probe error path
2018/7/27/512	drm: qxl: Fix error handling at qxl_device_init
2018/7/27/727	fmc: Fix memory leak and NULL pointer dereference
2018/7/27/755	drm: qxl: Fix NULL pointer dereference at qxl_alloc_client_monitors_config
2018/6/9/253	staging: rts5208: add error handling into rtsx_probe
2018/7/27/644	tty: rocket: Fix possible buffer overwrite on register_PCI
2018/8/6/615	serial: mxs-auart: Fix potential infinite loop
2018/8/7/292	usb: gadget: fotg210-udc: Fix memory leak of fotg210->ep[i]

Let's consider the bug 2017/8/15/322 from Table 4 discovered in the Samsung I2S Controller driver within Linux 4.11.6 for which our patch was applied in 4.14-rc1.

```
1229 static int samsung_i2s_probe(struct platform_device *pdev)
1230 {
1231     struct i2s_dai *pri_dai, *sec_dai = NULL;
```

Fig. 7. (a) probe function

Klever provides a full error trace from an entry point to a error occurrence for the Unsafe verdict. The parts of the error trace for the Samsung I2S Controller driver are shown in fig. 7.

Fig. 7 (a) shows a part of the error trace with the declaration of the variable *struct i2s_dai *pri_dai* in function *samsung_i2s_probe()*. In the same function in fig. 7 (b) *pri_dai* is initialized by function *i2s_alloc_dai()* (line 1246), and field *sec_dai* becomes NULL (line 1095).

The third part of the error trace in fig. 7.(c) shows that *sec_dai* initialization is skipped by condition in line 1319 (*quirks & QUIRK_SEC_DAI*) triggered by device capabilities, so *pri_dai* is remained equal to NULL.

In the fig. 7, (d) we see that the structure *pri_dai* becomes stored at *driver_data* by *dev_set_drvdata()* in line 1363 and then extracted by *dev_get_drvdata()* in line 1382 of *samsung_i2s_remove()*. Next the driver assigns *sec_dai* in line 1383 and then perform dereference of *sec_dai* in line 1386 without check for NULL, which leads to NULL pointer dereference.

The bug can be reproduced on Samsung s3c6410-i2s and exynos7-i2s1 devices by inserting and removing driver module *sound/soc/samsung/i2s.ko*, because the condition in line 1319 is false for *i2sv3_dai_type* and *i2sv5_dai_type_i2s1* (see lines 1454 and 1477 in *sound/soc/samsung/i2s.c*).

```
pri_dai = i2s_alloc_dai(pdev, 0);
    struct i2s_dai *i2s ;
    i2s = (struct i2s_dai *)tmp;
    assume(((unsigned long)i2s) != ((unsigned long)((struct i
    i2s->pdev = pdev;
    i2s->pri_dai = (struct i2s_dai *)0;
    i2s->sec_dai = (struct i2s_dai *)0;
    i2s->i2s_dai_drv.symmetric_rates = 1U;
    i2s->i2s_dai_drv.probe = &samsung_i2s_dai_probe;
    i2s->i2s_dai_drv.remove = &samsung_i2s_dai_remove;
    i2s->i2s_dai_drv.ops = &samsung_i2s_dai_ops;
    i2s->i2s_dai_drv.suspend = &i2s_suspend;
```

Fig. 7. (b) pri_dai initialization

```
    if (quirks & QUIRK_SEC_DAI) {
        sec_dai = i2s_alloc_dai(pdev, true);
        if (!sec_dai) {
            dev_err(&pdev->dev, "Unable to alloc I2S_sec\n");
            ret = -ENOMEM;
            goto err_disable_clk;
        }

        sec_dai->lock = &pri_dai->spinlock;
        sec_dai->variant_regs = pri_dai->variant_regs;
        sec_dai->dma_playback.addr = regs_base + I2STXDS;
        sec_dai->dma_playback.chan_name = "tx-sec";

        if (!np) {
            sec_dai->dma_playback.filter_data = i2s_pdata->dma_
            sec_dai->filter = i2s_pdata->dma_filter;
        }

        sec_dai->dma_playback.addr_width = 4;
        sec_dai->addr = pri_dai->addr;
        sec_dai->clk = pri_dai->clk;
        sec_dai->quirks = quirks;
        sec_dai->idma_playback.addr = idma_addr;
        sec_dai->pri_dai = pri_dai;
        pri_dai->sec_dai = sec_dai;

        ret = samsung_asoc_dma_platform_register(&pdev->dev,
                    sec_dai->filter, "tx-sec", NULL);
        if (ret < 0)
            goto err_disable_clk;
```

Fig. 7. (c) skipped pri_dai initialization


```
assume(ret >= 0);
assume((quirks &2U) == 0U);
assume(((unsigned long)i2s_pdata) == ((unsigned long)((struct
dev_set_drvdata(& pdev->dev, (void *)pri_dai);
dev_set_drvdata(&pdev->dev, (void *)pri_dai);
    dev->driver_data = data;
    return;

pm_runtime_set_active(&pdev->dev);
pm_runtime_enable(&pdev->dev);
ret = i2s_register_clock_provider(pdev);
assume(ret == 0);
return 0;

ldv_2_probed_default = ldv_post_probe(ldv_2_probed_default);

Remove device from the system. Invoke callback remove from platform
samsung_i2s_remove(arg1);
samsung_i2s_remove(ldv_2_resource_platform_device);
    struct i2s_dai *pri_dai ;
    struct i2s_dai *sec_dai ;
    pri_dai = (struct i2s_dai *)tmp;
    pri_dai = (struct i2s_dai *)dev_get_drvdata((struct device
        return ((void *)dev->driver_data);
        return ((void *)dev->driver_data);

    sec_dai = pri_dai->sec_dai;
    pri_dai->sec_dai = (struct i2s_dai *)0;
    NULL pointer dereference on write
    sec_dai->pri_dai = (struct i2s_dai *)0;
```

Fig. 7. (d) dev_set_drvdata/dev_get_drvdata and NULL pointer dereference

7. Conclusions and future work

We have presented the approach to find memory errors in Linux kernel drivers using static verification. Whereas the Linux kernel is widely tested, our experiments show that it is possible to find memory bugs in Linux kernel drivers with help of our static verification method.

We expect to reduce the false alarm rate by introducing a more precise predicate extension. Further efforts will be aimed at reducing the number of timeouts.

References

[1]. G. Klein, J. Andronick, K. Elphinstone, T. Murray, T. Sewell, R. Kolanski, and G. Heiser, Comprehensive formal verification of an os microkernel. ACM Transactions on Computer Systems, vol. 32, no. 1, 2014, pp. 2:1–2:70.

[2]. T. Ball, E. Bounimova, B. Cook, V. Levin, J. Lichtenberg, C. McGarvey, B. Ondrusek, S. K. Rajamani, and A. Ustuner, Thorough static analysis of device drivers. SIGOPS Operating Systems Review, vol. 40, no. 4, 2006, pp. 73–85.

[3]. D. Engler and M. Musuvathi. Static analysis versus software model checking for bug finding. Lecture Notes in Computer Science, vol. 2937, 2004, pp. 191–210.
[4]. Saturn. Precise and Scalable Software Analysis. Available at: http://saturn.stanford.edu/, accessed 01.12.2018.
[5]. T. Witkowski, N. Blanc, D. Kroening, and G. Weissenbacher. Model checking concurrent Linux device drivers. In Proceedings of the 22nd IEEE/ACM Int. Conference on Automated Software Engineering, 2007, pp. 501–504.
[6]. N. Palix, G. Thomas, S. Saha, C. Calvès, J. Lawall, and G. Muller. Faults in Linux: Ten years later. In Proceedings of the 16th Int. Conference on Architectural Support for Programming Languages and Operating Systems, 2011, pp. 305–318.
[7]. Linux driver verification project. Available at: http://linuxtesting.org/ldv, accessed 01.12.2018.
[8]. V. Mutilin, E. Novikov, and A. Khoroshilov. Analysis of typical faults in Linux operating system drivers. Trudy ISP RAN/Proc. ISP RAS, vol. 22, 2012, pp. 349–374 (in Russian). DOI: 10.15514/ISPRAS-2012-22-19.
[9]. A. Khoroshilov, V. Mutilin, A. Petrenko, and V. Zakharov. Establishing Linux driver verification process, Lecture Notes in Computer Science, vol. 5947, pp. 165–176, 2010.
[10]. I. Zakharov, M. Mandrykin, V. Mutilin, E. Novikov, A. Petrenko, and A. Khoroshilov. Configurable toolset for static verification of operating systems kernel modules. Programming and Computer Software, vol. 41, no. 1, 2015, pp. 49–64.
[11]. Klever verification framework. Available at: https://forge.ispras.ru/projects/klever, accessed 01.12.2018.
[12]. 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, 2015, pp. 183–195.
[13]. A. Khoroshilov, V. Mutilin, E. Novikov, and I. Zakharov. Modeling environment for static verification of linux kernel modules. Lecture Notes in Computer Science, vol. 8974, 2015, pp. 400–414.
[14]. E. Novikov and I. Zakharov. Towards automated static verification of GNU C programs. Lecture Notes in Computer Science, vol. 10742, 2018, pp. 402–416.
[15]. D. Beyer and M. Keremoglu. CPAchecker: A tool for configurable software verification. Lecture Notes in Computer Science, vol. 6806, 2011, pp. 184–190.
[16]. K. Dudka, P. Peringer, and T. Vojnar. Byte-precise verification of low-level list manipulation. Lecture Notes in Computer Science, vol. 7935, 2013, pp. 215–237.
[17]. R. Wilhelm, S. Sagiv, and T. W. Reps. Shape analysis. Lecture Notes in Computer Science, vol. 1781, 2000, pp. 1–17.
[18]. D. Beyer, T. A. Henzinger, and G. Théoduloz. Configurable software verification: concretizing the convergence of model checking and program analysis. Lecture Notes in Computer Science, vol. 4590, 2007, pp. 504–518.

Статическая верификация ошибок использования памяти в модулях ядра ОС Linux

А.А. Васильев <vasilyev@ispras.ru>
Институт системного программирования им. В.П. Иванникова РАН, 109004, Россия, г. Москва, ул. А. Солженицына, д. 25.

Abstract. Ошибки использования памяти в модулях ядра операционной системы Linux сложно обнаружить, но они могут привести к серьезным последствиям. В данной статье мы описываем метод статической верификации, позволяющий обнаруживать все ошибки в рамках предположений метода. Статическая верификация крупных пректов таких, как ядро ОС Linux, требуют дополнительных усилий. Современные инструменты статической верификации не позволяют анализировать ядро как единое целое, поэтому мы используем упрощенную автоматически генерируемую модель окружения. Эта модель вносит некоторую неточность, но позволяет проводить статическую верификацию. Также мы допускаем отсутствие тела некоторых функций, что приводит к неполным программам, написанных на языке ANSI C. В данной работе предлагается подход к обнаружению ошибок использования памяти в таких неполных программах. Наша техника статической верификации основана на теории символических графов памяти и ее расширении для снижения количества ложных срабатываний. Мы ввели концепцию памяти по требованию для упрощения моделей интерфейсов ядра ОС и реализовали ее в фреймворке статической верификации CPAchecker. Также мы изменили точность модели памяти CPAchecker с байтов на поддержку отдельных битов и добавили поддержку выравнивания структур, аналогичное использованному в компиляторе. Для повышения точности анализа мы реализовали предикатное расширение состояния символического графа памяти. Мы провели проверку модулей ядра ОС Linux для версий 4.11.6 и 4.16.10 с помощью фреймворка статической верификации Klever с инструментом верификации CPAchecker, что позволило проанализировать 6224 и 5215 модулей соответствующих версий. Ручной анализ предупреждений от фреймворка Klever выявил 78 реальных ошибок в модулях ядра. Мы сделали патчи для исправления 33 из них.

Ключевые слова: анализ рекурсивных структур данных; статическая верификация; символические графы памяти; модели памяти.

DOI: 10.15514/ISPRAS-2018-30(6)-8

Для цитирования: Васильев А.А. Статическая верификация ошибок использования памяти в модулях ядра ОС Linux. Труды ИСП РАН, том 30, вып. 6, 2018 г., стр. 143-160. DOI: 10.15514/ISPRAS-2018-30(6)-8

Список литературы

[1]. G. Klein, J. Andronick, K. Elphinstone, T. Murray, T. Sewell, R. Kolanski, and G. Heiser, Comprehensive formal verification of an os microkernel. ACM Transactions on Computer Systems, vol. 32, no. 1, 2014, pp. 2:1–2:70.
[2]. T. Ball, E. Bounimova, B. Cook, V. Levin, J. Lichtenberg, C. McGarvey, B. Ondrusek, S. K. Rajamani, and A. Ustuner, Thorough static analysis of device drivers. SIGOPS Operating Systems Review, vol. 40, no. 4, 2006, pp. 73–85.

[3]. D. Engler and M. Musuvathi. Static analysis versus software model checking for bug finding. Lecture Notes in Computer Science, vol. 2937, 2004, pp. 191–210.
[4]. Saturn. Precise and Scalable Software Analysis. Available at: http://saturn.stanford.edu/, accessed 01.12.2018.
[5]. T. Witkowski, N. Blanc, D. Kroening, and G. Weissenbacher. Model checking concurrent Linux device drivers. In Proceedings of the 22nd IEEE/ACM Int. Conference on Automated Software Engineering, 2007, pp. 501–504.
[6]. N. Palix, G. Thomas, S. Saha, C. Calvès, J. Lawall, and G. Muller. Faults in Linux: Ten years later. In Proceedings of the 16th Int. Conference on Architectural Support for Programming Languages and Operating Systems, 2011, pp. 305–318.
[7]. Linux driver verification project. Available at: http://linuxtesting.org/ldv, accessed 01.12.2018.
[8]. В.С. Мутилин, Е.М. Новиков, А.В. Хорошилов, Анализ типовых ошибок в драйверах операционной системы Linux. Труды ИСП РАН, том 22, 2012, стр. 349-374. DOI: 10.15514/ISPRAS-2012-22-19.
[9]. A. Khoroshilov, V. Mutilin, A. Petrenko, and V. Zakharov. Establishing Linux driver verification process, Lecture Notes in Computer Science, vol. 5947, pp. 165–176, 2010.
[10]. I. Zakharov, M. Mandrykin, V. Mutilin, E. Novikov, A. Petrenko, and A. Khoroshilov. Configurable toolset for static verification of operating systems kernel modules. Programming and Computer Software, vol. 41, no. 1, 2015, pp. 49–64.
[11]. Klever verification framework. Available at: https://forge.ispras.ru/projects/klever, accessed 01.12.2018.
[12]. 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, 2015, pp. 183–195.
[13]. A. Khoroshilov, V. Mutilin, E. Novikov, and I. Zakharov. Modeling environment for static verification of linux kernel modules. Lecture Notes in Computer Science, vol. 8974, 2015, pp. 400–414.
[14]. E. Novikov and I. Zakharov. Towards automated static verification of GNU C programs. Lecture Notes in Computer Science, vol. 10742, 2018, pp. 402–416.
[15]. D. Beyer and M. Keremoglu. CPAchecker: A tool for configurable software verification. Lecture Notes in Computer Science, vol. 6806, 2011, pp. 184–190.
[16]. K. Dudka, P. Peringer, and T. Vojnar. Byte-precise verification of low-level list manipulation. Lecture Notes in Computer Science, vol. 7935, 2013, pp. 215–237.
[17]. R. Wilhelm, S. Sagiv, and T. W. Reps. Shape analysis. Lecture Notes in Computer Science, vol. 1781, 2000, pp. 1–17.
[18]. D. Beyer, T. A. Henzinger, and G. Théoduloz. Configurable software verification: concretizing the convergence of model checking and program analysis. Lecture Notes in Computer Science, vol. 4590, 2007, pp. 504–518.