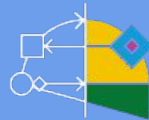




SBMF 2023
26th Brazilian Symposium on
Formal Methods



AFRITS 2023
Workshop on automated Formal
Reasoning for Trustworthy AI systems
Manaus - Brazil

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

Janisley Oliveira de Sousa

Email: janisley.sousa@sidia.com



ESBMC



Outline

1. **Linux Kernel: An Overview**
2. **Linux Kernel Drivers**
3. **Security Vulnerabilities in Kernel Drivers**
4. **Memory Safety Issues**
5. **Deep Dive into Kernel-specific Vulnerabilities Fixed**
6. **Introduction to Formal Software Verification**
7. **Introduction to ESBMC and LSVerifier**
8. **Hands-on: Applying Bounded Model Checking to exploit Linux Device Drivers**
9. **Plus.**

1. Linux Kernel – Introduction

- The **complexity of software development is constantly increasing**, especially in the area of system-level programming.
- The **Linux kernel** stands out among the **top operating systems** due to its open-source status and wide use.
- But because of its **extensive code base** and importance in **reconciling hardware** and **software elements**, it is a key area of **reliability and security concern**.
- The **importance of kernel verification** for the industry is highlighted by the growing integration of Linux-based systems in crucial applications, from **mobile systems, automobile control units to even medical equipment**.
- The techniques under the field of **formal methods, static analysis and bounded model checking**, are developed and utilized to solve the concern of the potential vulnerabilities in Linux kernel programs.
- **Formal verification methods provide an alternative by providing mathematical evidence of program correctness**. EMBMC has distinguished itself as a useful tool for comprehensive analysis.

Linux Kernel Drivers

An Overview.

- Linux kernel was created by Linus Torvalds in the early 1990s and functions as the brains behind the Linux operating system.



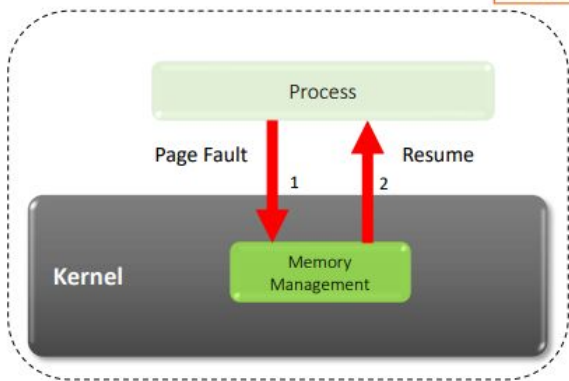
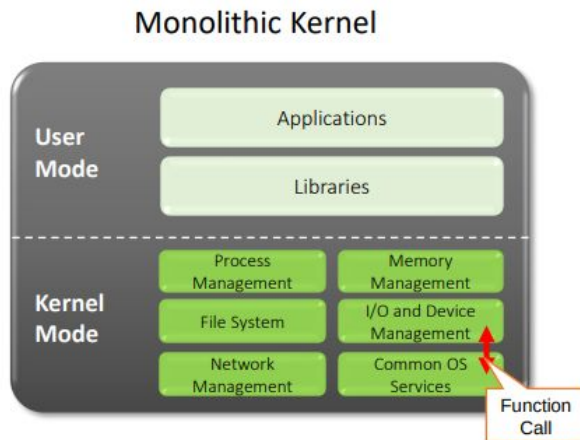
- The official site is <http://www.kernel.org>.
 - Linus Torvalds Announces Linux Kernel 6.7.
-

Linux Kernel Drivers

An Overview.

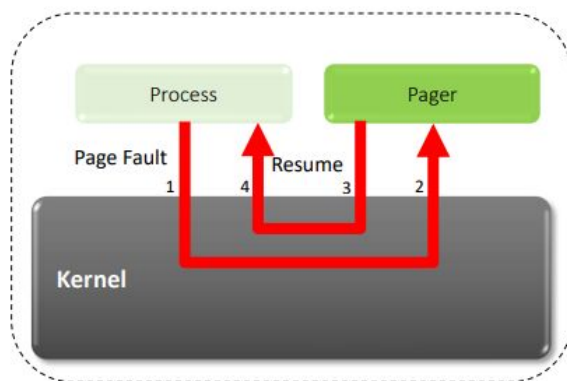
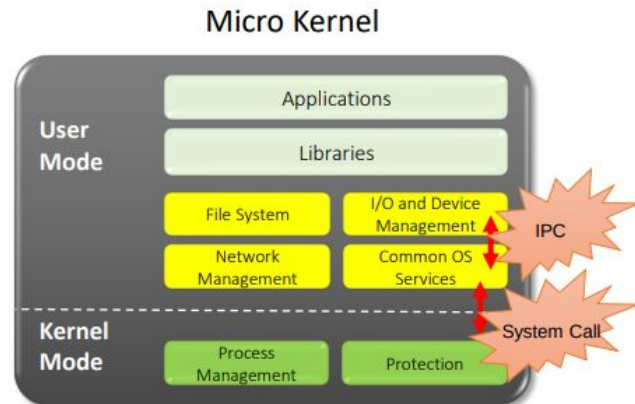
- The Linux kernel is a
 - Free and Open-Source Software (FOSS).
 - Monolithic Kernel.
 - Unix-like operating system kernel.
- The Linux kernel is released under the GNU General Public License version 2 (GPLv2)
 - with some firmware images released under various non-free licenses.
- Day-to-day development discussions take place on the Linux kernel mailing list (LKML)
 - <http://vger.kernel.org/vger-lists.html>

1. Linux Kernel - Design : Monolithic Kernel



Two Context Switches

**Monolithic
kernel
achieves
higher
performance**

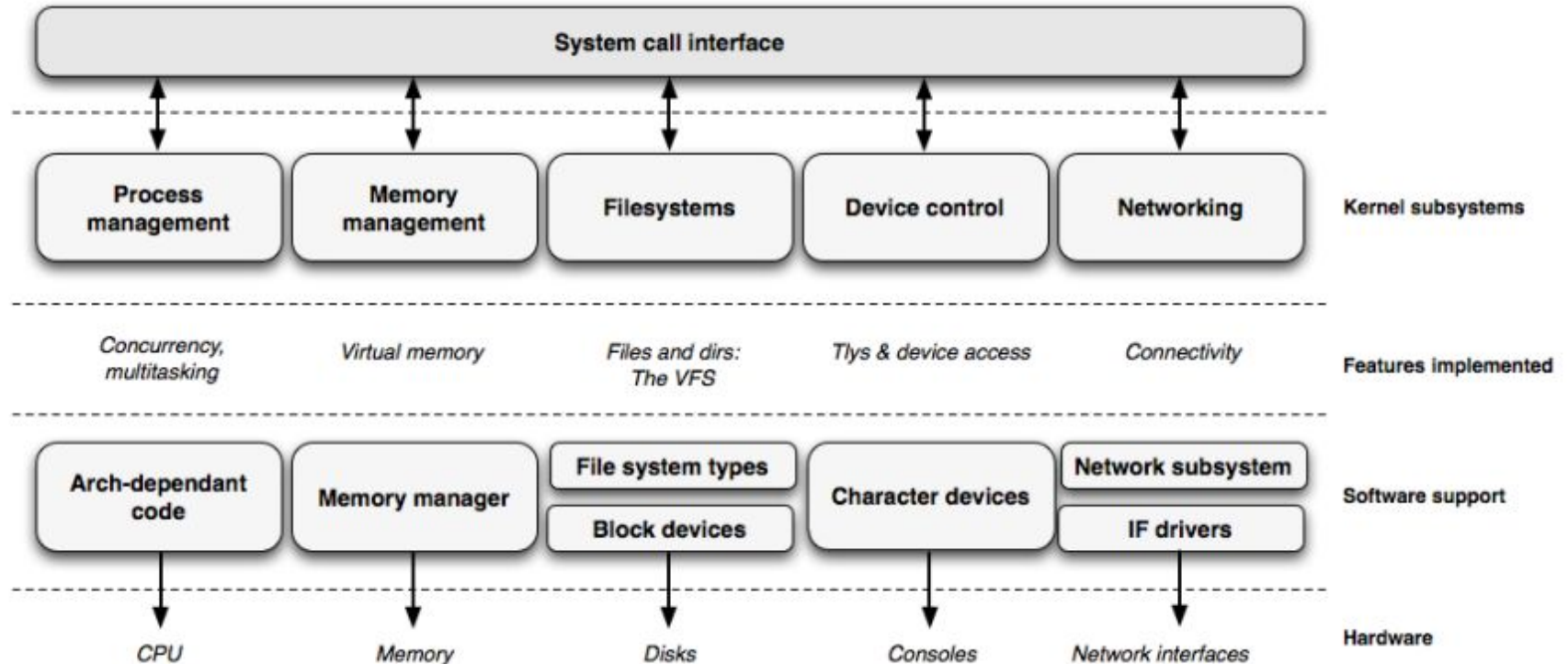


Four Context Switches

1. Linux Kernel – Design : Unique Features

- Loadable kernel module
 - Loadable kernel module is an object file whose code can be linked with kernel at runtime.
 - Linux achieves modularity and extensibility via loadable kernel module.
 - Most kernel functions can be compiled as module at kernel build time.
 - Device drivers, network protocols, file systems, etc.
- Process/thread model
 - Unique data structures and system calls for supporting multithreading.
- Preemptive kernel support
 - A process running in kernel mode can be replaced by another process in the middle of a kernel function.
 - Can reduce the dispatch latency of the user mode processes.
 - Linux kernel became preemptive from v2.6 : It is now possible to preempt a task at any point.

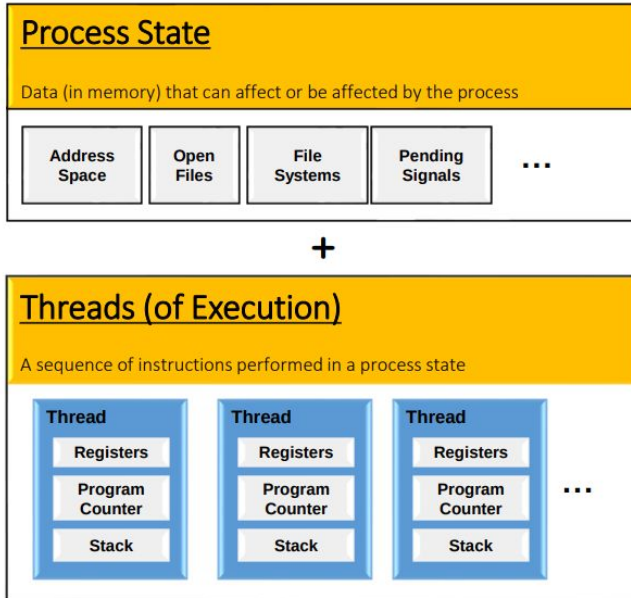
1. Linux Kernel – Architecture : Kernel Subsystems



1. Linux Kernel – Process (is a program in the midst of execution)

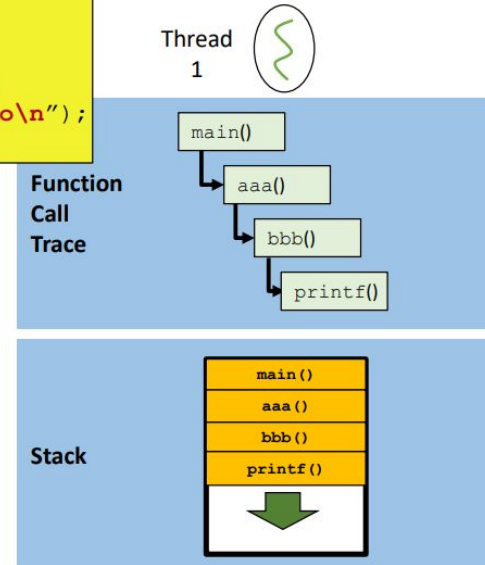
- Processes are, however, more than just the executing program code.
- They also include:
 - A set of resources such as open files & pending signals.
 - Internal kernel data.
 - Processor state & a memory address space with one or more memory mappings.
 - One or more threads of execution.
 - A data section containing global variables.
- A program itself is not a process.
- A process is an active program and related resources.
- Two or more processes can exist that are executing the same program.
- Two or more processes can exist that share various resources, such as open files or an address space.

1. Linux Kernel – Process creation

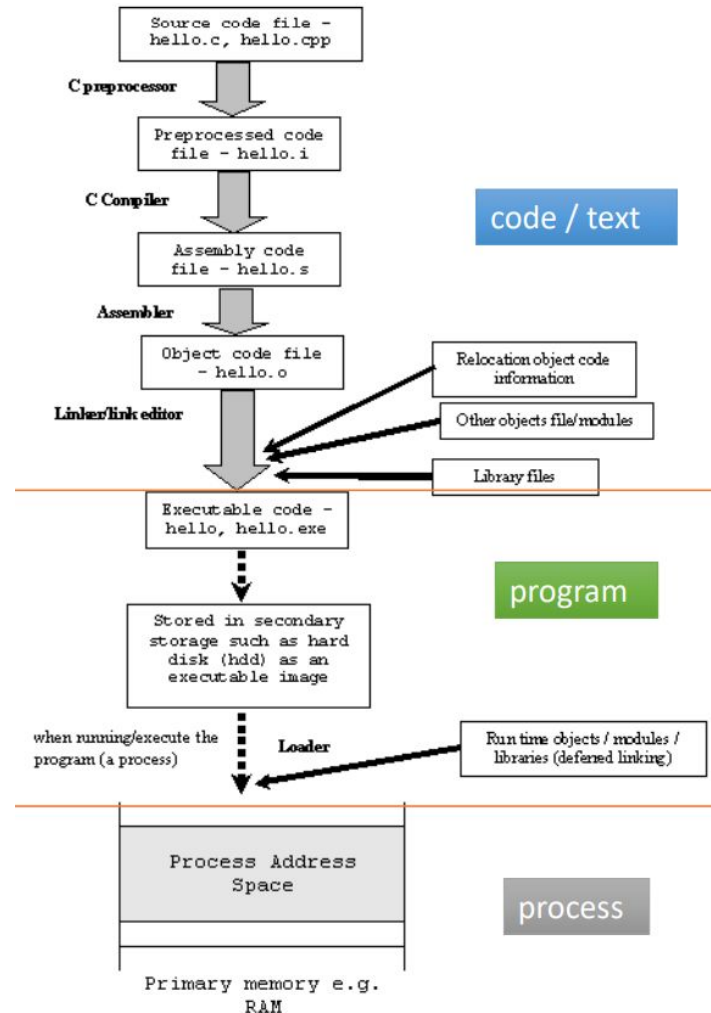


Example Program

```
int main(void) {  
    aaa();  
    return 0;  
}  
  
void aaa(void) {  
    bbb();  
}  
  
void bbb(void) {  
    printf("hello\n");  
}
```

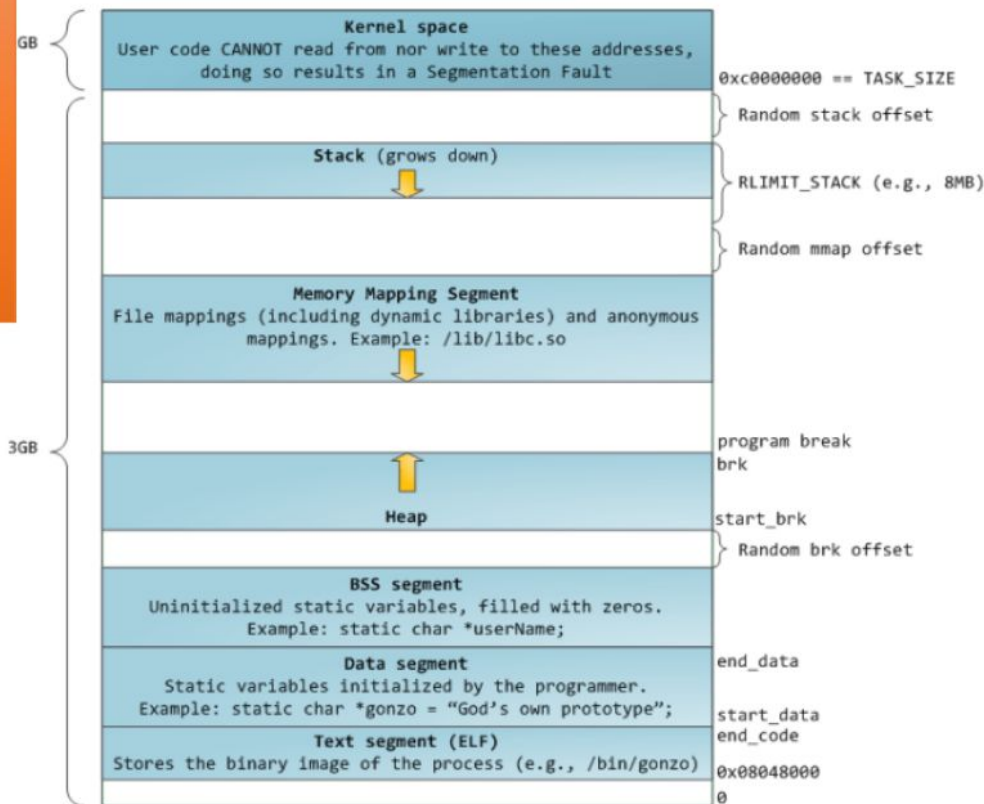


1. Linux Kernel – Journey of a Code to a Program to a Process

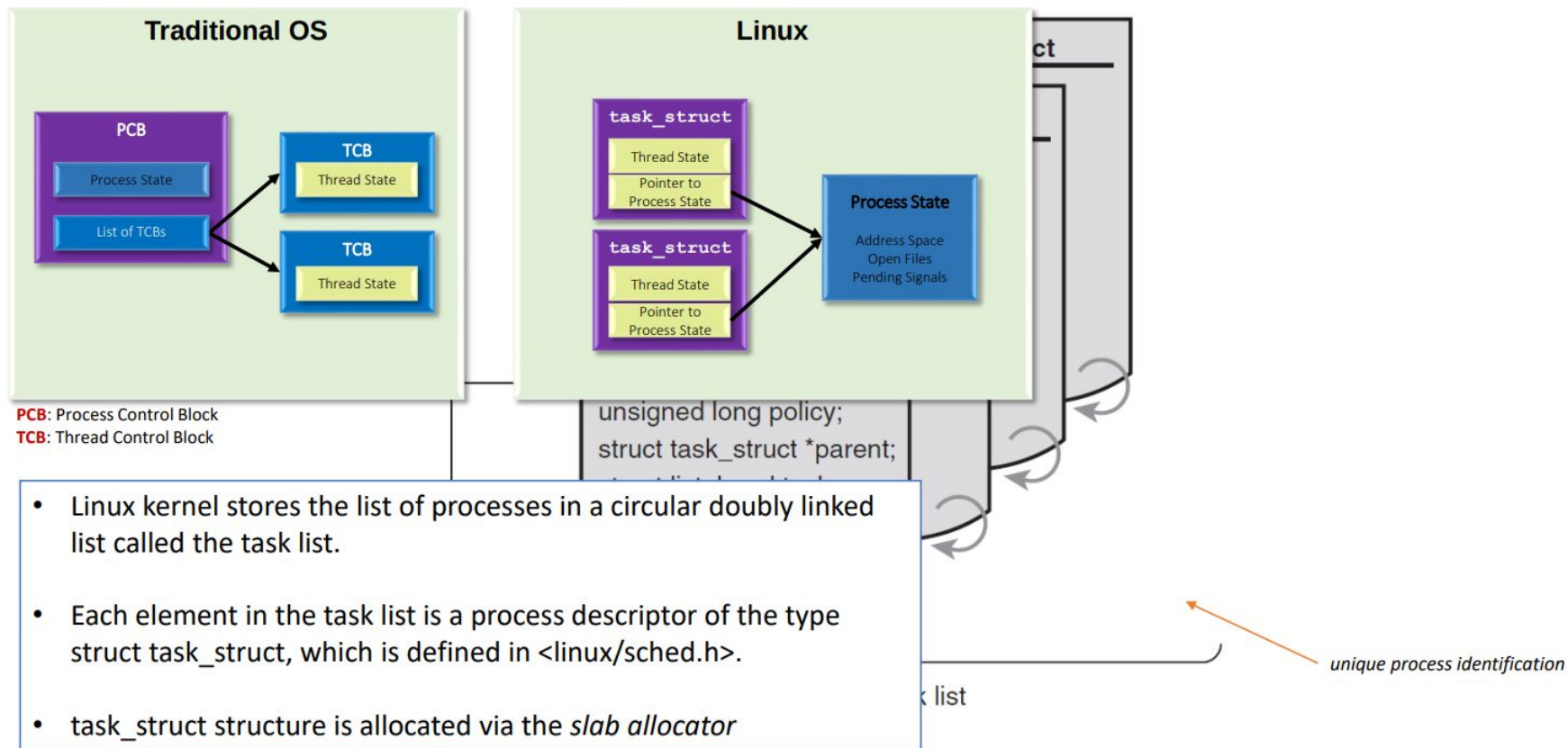


1. Linux Kernel – Journey of a Code to a Program to a Process

A process is a program
in the midst of
execution



1. Linux Kernel – Journey of a Code to a Program to a Process



- Linux kernel stores the list of processes in a circular doubly linked list called the task list.
- Each element in the task list is a process descriptor of the type `struct task_struct`, which is defined in `<linux/sched.h>`.
- `task_struct` structure is allocated via the *slab allocator*

1. Linux Kernel – Code size

- > 70,000 files.
- > 26,370,000 lines.
- > 5000 developers.
- > 550 companies.
- Every single day:
 - 10,000 lines added.
 - 2,700 lines removed.
 - 2,000 lines modified.

Longterm release kernels

Version	Maintainer	Released	Projected EOL
6.1	Greg Kroah-Hartman & Sasha Levin	2022-12-11	Dec, 2026
5.15	Greg Kroah-Hartman & Sasha Levin	2021-10-31	Oct, 2026
5.10	Greg Kroah-Hartman & Sasha Levin	2020-12-13	Dec, 2026
5.4	Greg Kroah-Hartman & Sasha Levin	2019-11-24	Dec, 2025
4.19	Greg Kroah-Hartman & Sasha Levin	2018-10-22	Dec, 2024
4.14	Greg Kroah-Hartman & Sasha Levin	2017-11-12	Jan, 2024

Linux Device Driver

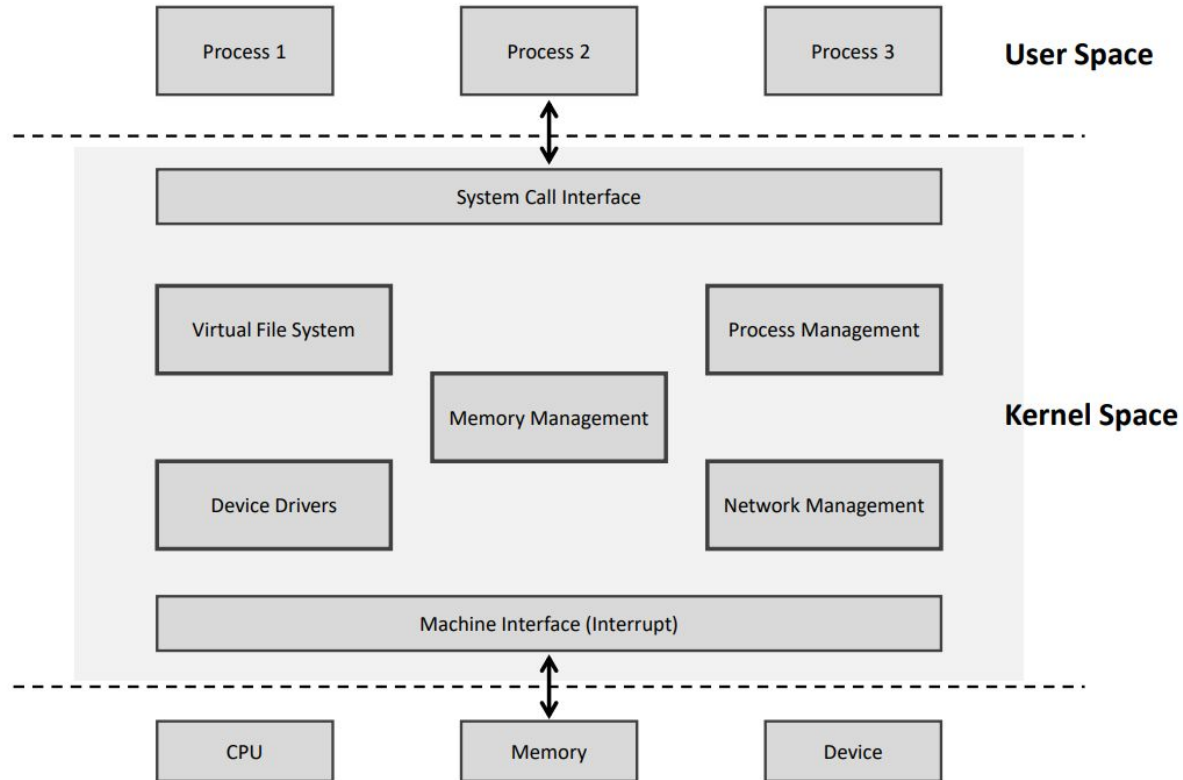
Modules for each HW
components.

- Drivers are used to help the hardware devices interact with the operating system.
- In windows, all the devices and drivers are grouped together in a single console called device manager.
- In Linux, even the hardware devices are treated like ordinary files, which makes it easier for the software to interact with the device drivers. When a device is connected to the system, a device file is created in /dev directory.

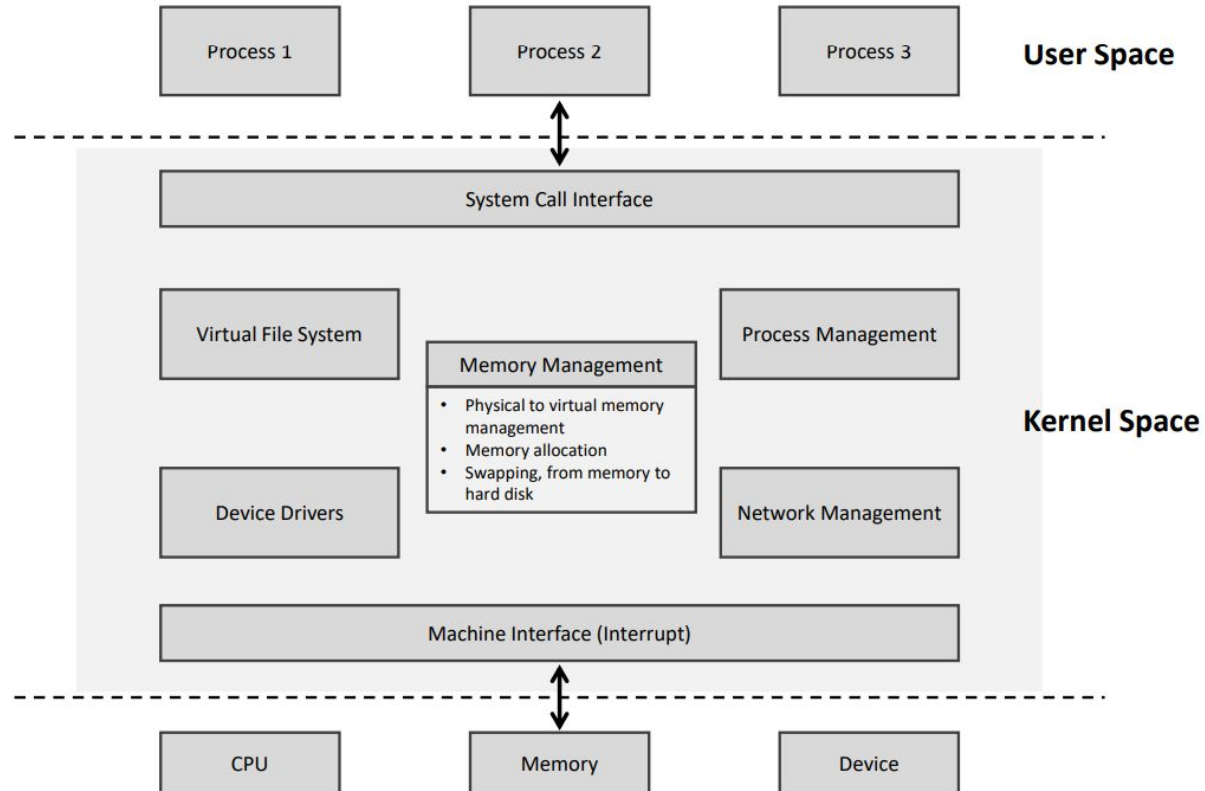
2. Linux Device Drivers

- Loadable kernel module
 - Loadable kernel module is an object file whose code can be linked with kernel at runtime.
 - Linux achieves modularity and extensibility via loadable kernel module.
 - Most kernel functions can be compiled as module at kernel build time.
 - Device drivers, network protocols, file systems, etc.
- Process/thread model
 - Unique data structures and system calls for supporting multithreading.
- Preemptive kernel support
 - A process running in kernel mode can be replaced by another process in the middle of a kernel function.
 - Can reduce the dispatch latency of the user mode processes.
 - Linux kernel became preemptive from v2.6 : It is now possible to preempt a task at any point.

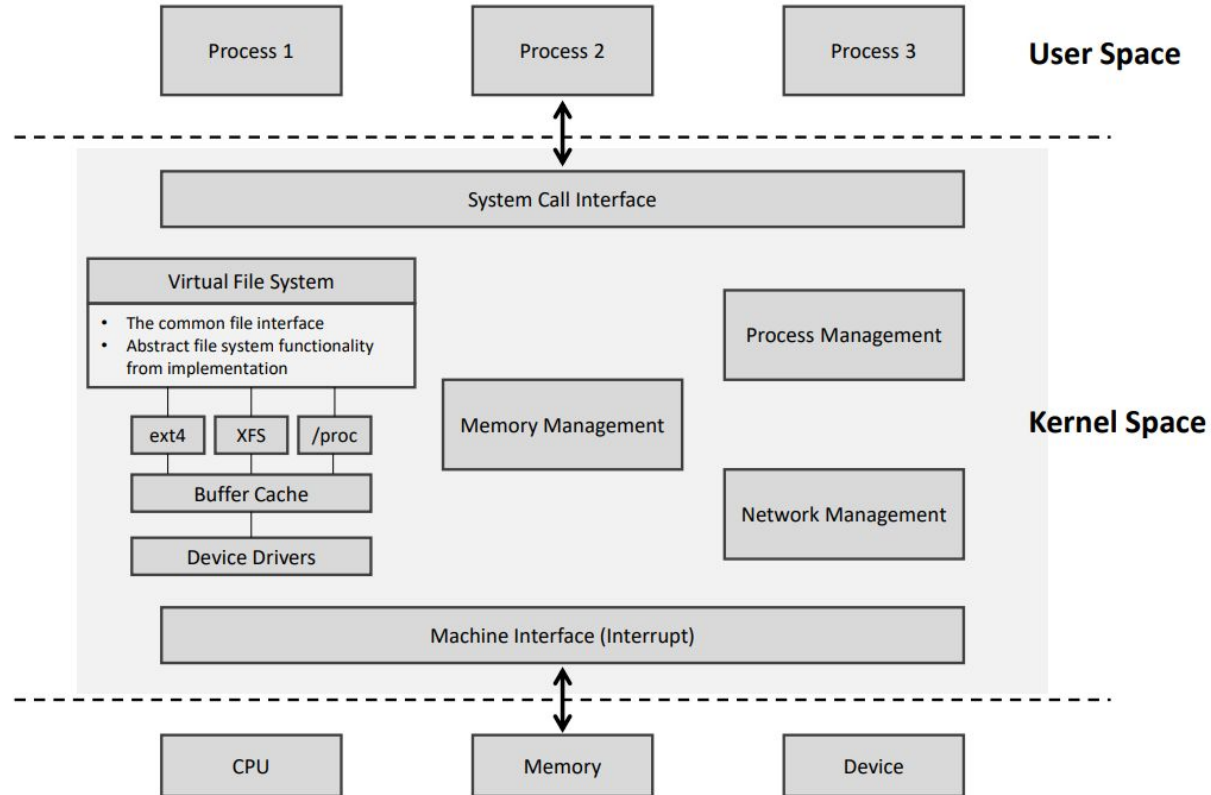
2. Linux Device Drivers



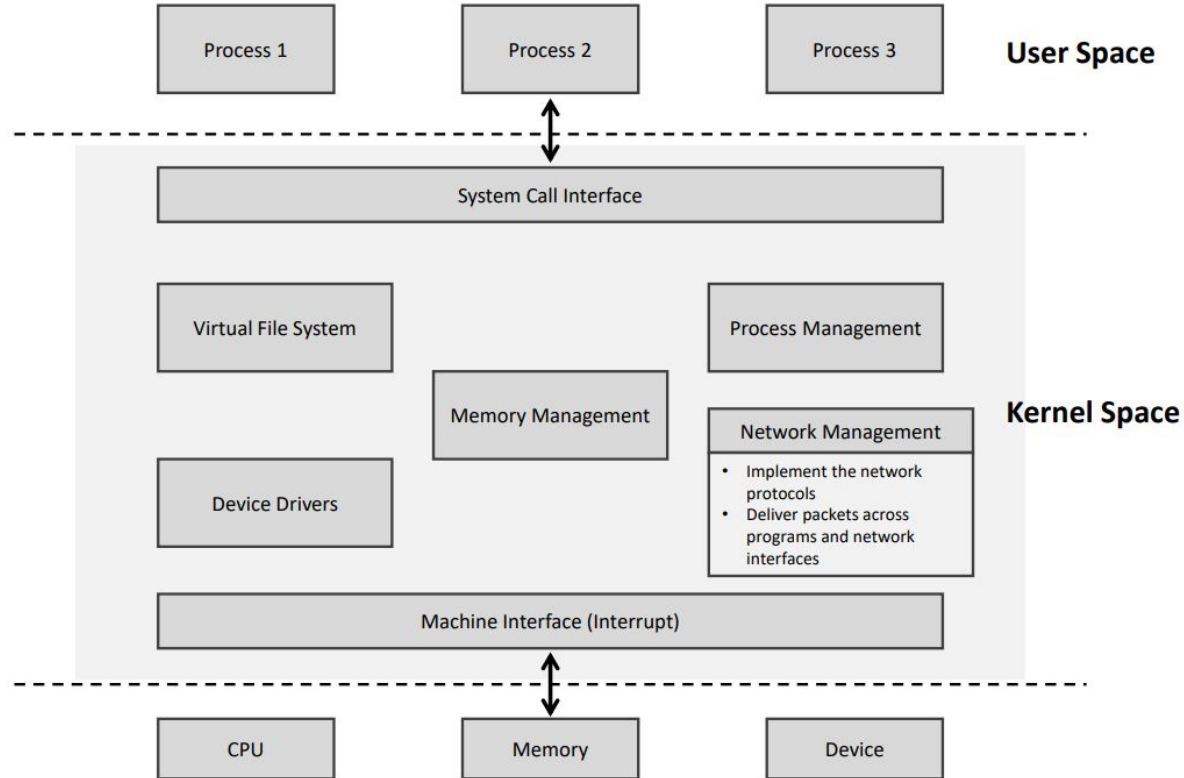
2. Linux Device Drivers



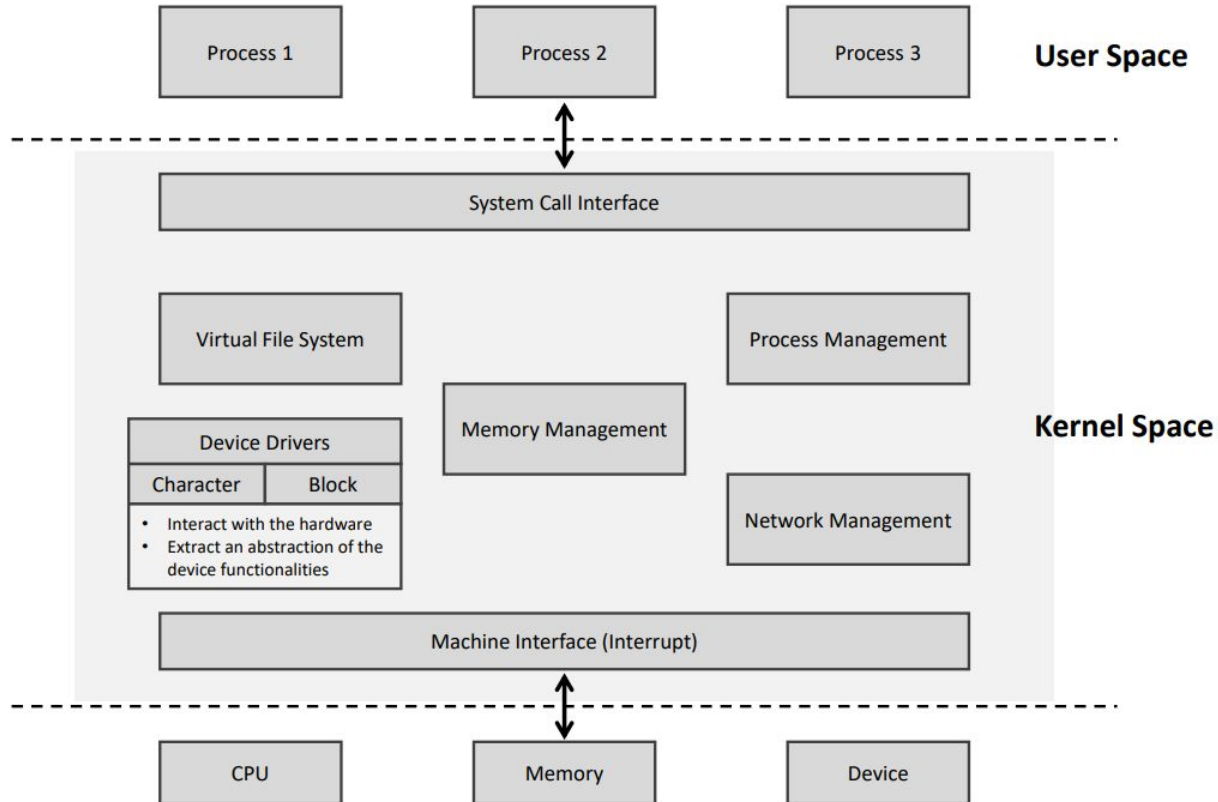
2. Linux Device Drivers



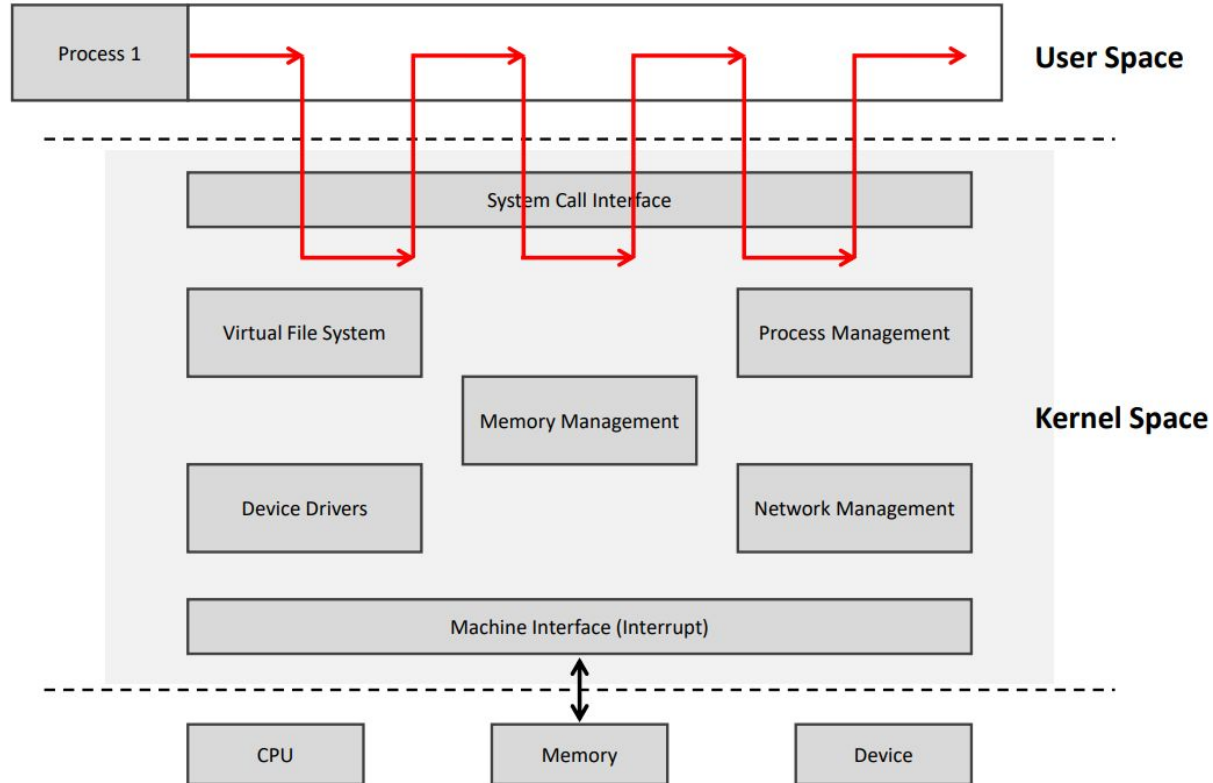
2. Linux Device Drivers



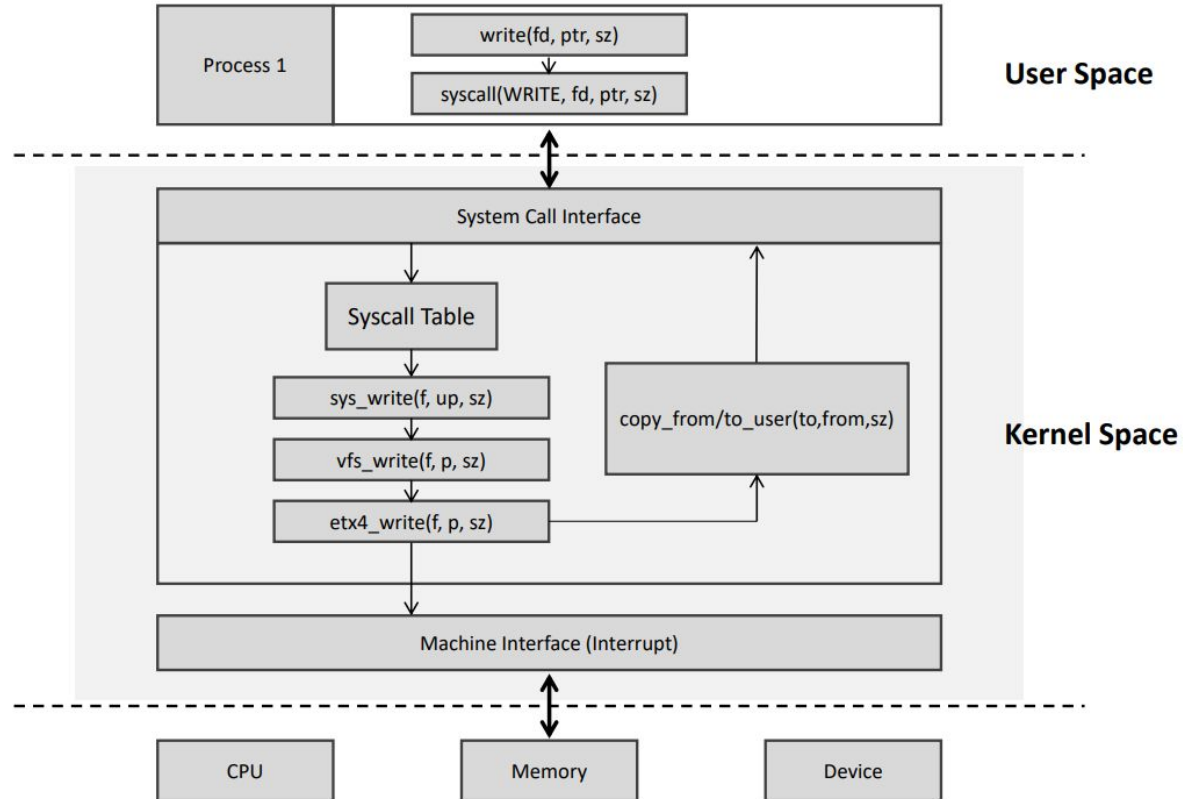
2. Linux Device Drivers



2. Linux Device Drivers



2. Linux Device Drivers



Security Vulnerabilities in Kernel Drivers

Overview of research area challenges.

- Security vulnerabilities in kernel drivers can be a serious concern as they can provide attackers with a way to gain privileged access to a system.
- Kernel drivers are an essential part of an operating system, and vulnerabilities in them can potentially lead to privilege escalation, denial of service, or even remote code execution.
- Previous research such as ScrewedDrivers and POPKORN utilized symbolic execution for automating the discovery of vulnerable drivers.

3. Security vulnerabilities in kernel drivers

- List of common vulnerabilities:
 - **Buffer Overflows:** Kernel drivers that handle user inputs may be vulnerable to buffer overflows.
 - **Use-After-Free:** It occurs when a driver references memory after it has been freed.
 - **Null Pointer Dereference:** It occurs when a kernel driver attempts to access or manipulate data through a null pointer.
 - **Privilege Escalation:** Some drivers may have vulnerabilities that allow an attacker to escalate their privileges from a low-privileged user to the kernel level.
 - **Uninitialized Memory:** Kernel drivers sometimes use uninitialized memory.
 - **Race Conditions:** It occurs when multiple threads or processes access shared resources concurrently without proper synchronization.
 - **Input Validation Flaws:** Failure to validate and sanitize input properly can lead to various security issues.
 - **Inadequate Access Controls:** Drivers should enforce access controls to prevent unauthorized access.
 - **Resource Leaks:** Failing to release resources properly can lead to resource leaks.
 - **Information Disclosure:** Kernel drivers may inadvertently leak sensitive information to user space.
 - **Denial of Service (DoS):** Vulnerabilities that lead to kernel crashes or system instability can be exploited for DoS attacks.

3. Security vulnerabilities in kernel drivers

Vulnerability	Mem. corruption	Policy violation	DoS	Info. disclosure	Misc.
Missing pointer check	6	0	1	2	0
Missing permission check	0	15	3	0	1
Buffer overflow	13	1	1	2	0
Integer overflow	12	0	5	3	0
Uninitialized data	0	0	1	28	0
Null dereference	0	0	20	0	0
Divide by zero	0	0	4	0	0
Infinite loop	0	0	3	0	0
Data race / deadlock	1	0	7	0	0
Memory mismanagement	0	0	10	0	0
Miscellaneous	0	0	5	2	1
Total	32	16	60	37	2

Figure 1: Vulnerabilities (rows) vs. possible exploits (columns). Some vulnerabilities allow for more than one kind of exploit, but vulnerabilities that lead to memory corruption are not counted under other exploits.

Vulnerability	Total	core	drivers	net	fs	sound
Missing pointer check	8	4	3	1	0	0
Missing permission check	17	3	1	2	11	0
Buffer overflow	15	3	1	5	4	2
Integer overflow	19	4	4	8	2	1
Uninitialized data	29	7	13	5	2	2
Null dereference	20	9	3	7	1	0
Divide by zero	4	2	0	0	1	1
Infinite loop	3	1	1	1	0	0
Data race / deadlock	8	5	1	1	1	0
Memory mismanagement	10	7	1	1	0	1
Miscellaneous	8	2	0	4	2	0
Total	141	47	28	35	24	7

Figure 2: Vulnerabilities (rows) vs. locations (columns).



3. Security vulnerabilities in kernel drivers - CVE list

- To mitigate these vulnerabilities, it's essential to follow best practices in kernel driver development:
 - Perform thorough code review and testing.
 - Use memory-safe programming languages and libraries.
 - Apply proper input validation and bounds checking.
 - Employ static analysis tools to detect vulnerabilities early in development.
 - Keep drivers up to date with security patches.
 - Follow the principle of least privilege when designing access controls.
 - Regularly audit and monitor driver behavior for suspicious activities.
- Linux kernel vulnerabilities published on the CVE list:



Memory Safety Issues

Kernel Linux

- **Software validation and verification techniques** are essential tools for developing robust systems with **high dependability** and **reliability** requirements.
- **Memory errors** in C software written in **unsafe programming languages** represent one of the main problems in **computer security**.
- The **Common Weakness Enumeration (CWE)** community identified a lot of vulnerabilities regarding the **C programming** language in **third-party drivers** used on the **kernel linux**.

4. Memory safety Issues

- The **C programming language** is widely used to develop **critical software**, such as operating systems, drivers, and encryption libraries. However, it **lacks protection mechanisms**, leaving memory and resource management's responsibility in the **developers' hands**.
- **Large software systems** are frequently composed of a myriad of elements declared in **several source files**, usually divided into **various directories**.
- To **handle large pieces of software** with many files, a scenario typically found in open-source applications, it is necessary to **verify each one** at once and then change the current entry point when required.

4. Memory safety Issues - The CWE Top 13

#	ID	Name
1	CWE-787	Out-of-bounds Write
2	CWE-79	Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')
3	CWE-89	Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')
4	CWE-20	Improper Input Validation
5	CWE-125	Out-of-bounds Read
6	CWE-78	Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')
7	CWE-416	Use After Free
8	CWE-22	Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')
9	CWE-352	Cross-Site Request Forgery (CSRF)
10	CWE-434	Unrestricted Upload of File with Dangerous Type
11	CWE-476	NULL Pointer Dereference
12	CWE-502	Deserialization of Untrusted Data
13	CWE-190	Integer Overflow or Wraparound

More info: https://cwe.mitre.org/top25/archive/2023/2023_top25_list.html

Kernel-specific Vulnerabilities Fixed

Some examples

- Kernel Secure fixes:
 - Happen at least once a week.
 - Look like any other bug fix.
 - Never specifically called out as a security fix.
 - Some bug fixes are not known to be security issues until years later.
 - No differentiation between bug types.
 - A bug is a bug is a bug.

5. Same examples of kernel linux vulnerabilities fixed

- Loads of netfilter bugs.
- Loads of keyring issues. Improved SELinux policies.
- A few “malicious network packet” attacks.
- Heaps of sound core exploits.
- Signal handling root exploit.
- Read-only kernel pages being writable.
- Malicious USB device attacks.
- Many, many more.

Linux kernel stable release bug fixes per month



5. Same examples of kernel linux vulnerabilities fixed

- **CVE-2021-26708**: This vulnerability in the Linux kernel, particularly in the "af_packet.c" in versions prior to 5.11.4, allowed local users to cause a denial of service or possibly execute arbitrary code. It was a use-after-free issue related to the way packet sockets were handled.
- **CVE-2019-19083**: Found in Linux kernel versions up to 5.3.13, this vulnerability involved a use-after-free in the "drivers/net/wireless/ath/ath9k/htc_hst.c" leading to potential remote code execution. It was mainly triggered by improper handling of certain error conditions.
- **CVE-2020-25670, CVE-2020-25671, CVE-2020-25672, CVE-2020-25673**: These were a series of vulnerabilities in the Linux kernel's "nf_tables" component, impacting versions before 5.9. They could lead to denial of service or privilege escalation due to improper handling of packet data.
- **CVE-2020-8835**: This vulnerability in the Linux kernel before 5.5.11 was due to an issue in the "bpf" subsystem that allowed an out-of-bounds write, potentially leading to code execution or denial of service.
- **CVE-2017-1000251**: Affecting the Linux kernel's Bluetooth stack (BlueZ), this vulnerability in versions before 4.13.1 allowed a stack buffer overflow, leading to arbitrary code execution by a physically proximate attacker.
- **CVE-2018-5953**: This was a buffer overflow vulnerability in the "vmxnet3" virtual network device driver of VMware's ESXi, Workstation, and Fusion products, potentially allowing a guest to execute code on the host.

Formal Verification Methods

A brief analysis.

- When applied to systems as complex as the Linux kernel, even formal verification methods like ESBMC have difficulties. Because of the kernel's size and the vast range of hardware configurations it supports, verification tools **face scalability and complexity challenges**.
- This **workshop** is based on the **Bounded Model Checking (BMC)**, and more peculiarly, the Bounded Model Checker **ESBMC (the Efficient SMT-based Context-Bounded Model Checker)**.

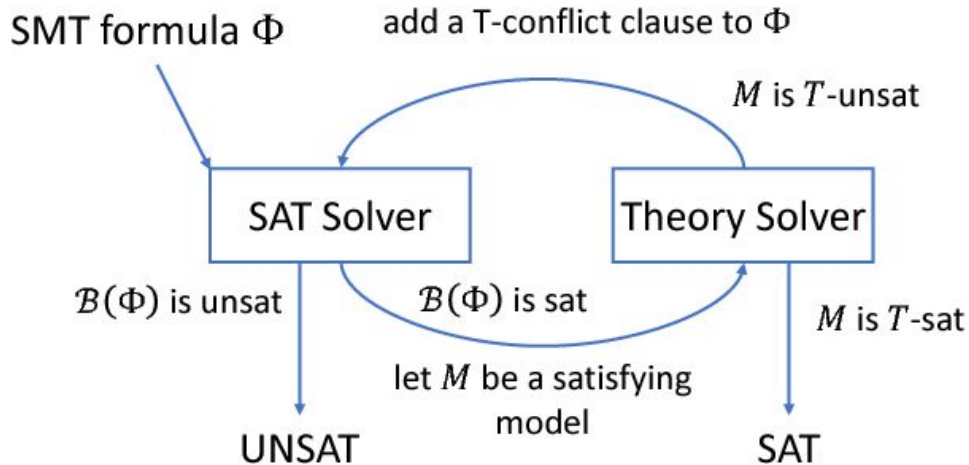
6. Static Analysis

- Static analysis refers to the set of methods used to evaluate a codebase for potential errors, vulnerabilities, or deviations.
- Static analysis tools can identify many common coding problems automatically before a program is released.
- They **examine** the text of a **program statically**, without attempting to execute it.
- They can examine either a program's source code or a compiled form of the program to equal benefit, although the problem of decoding the latter can be difficult
- C code static analyzer can detect spinlock usage in the Linux kernel. Misuse of spinlocks is difficult to detect and rather widespread, resulting in runtime deadlocks in the Linux operating system kernel on multiprocessor architectures.



6. Formal Software Verification - SMT Solvers

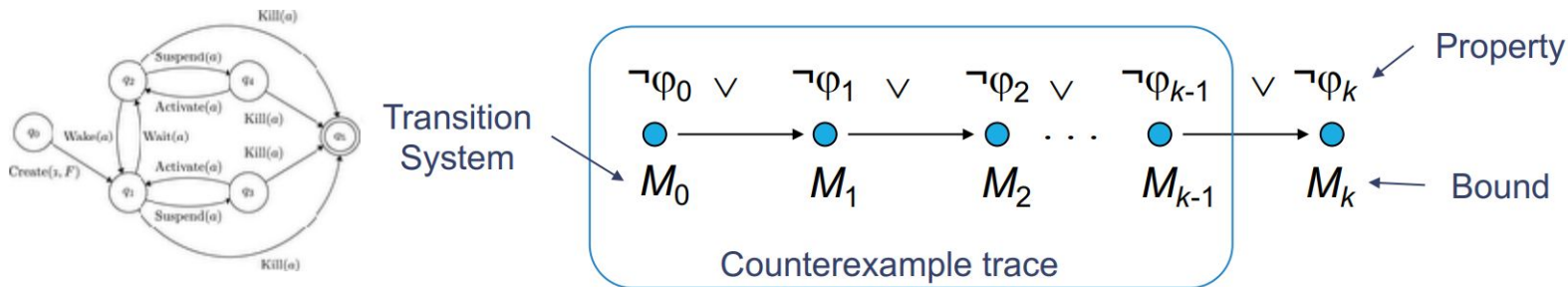
- Solvers aim to **determine if given formulas are satisfiable**. These formulas are broadly categorized into **Boolean satisfiability (SAT)** and **Satisfiability Modulo Theories (SMT)**.
- In the context of SAT, the main query is if a particular **Boolean formula can be validated**, and tools designed for this purpose are known as **SAT solvers**.
- One limitation of SAT is that the process of **translating higher-level system designs to Boolean logic** can be **resource-intensive**.
- This challenge led to the development of SMT solvers, which offer verification mechanisms capable of understanding more abstract levels while still harnessing the efficiency and automation typical of Boolean engines.



6. Bounded Model Checking (BMC) Approach

- **Basic Idea:** given a transition system **M**, check negation of a given property **φ** up to given depth **k**.

$$\text{BMC}_{\Phi}(k) = I(s_1) \wedge \left(\bigwedge_{i=1}^{k-1} T(s_i, s_{i+1}) \right) \wedge \left(\bigvee_{i=1}^k \neg \phi(s_i) \right)$$



- ESBMC is a context-bounded model checker, leveraging satisfiability modulo theories, designed for the verification of both single-threaded and multi-threaded C/C++ applications.
- Bounded model checkers “slice” the state space in depth.
- It is aimed to find bugs and can only prove correctness if all states are reachable within the bound.
- Exhaustively explores all executions.
- Can be bounded to limit number of iterations and context-switch.
- Report errors as traces.

LSVerifier

Open-Source Tool

Apache 2.0

<https://github.com/janisley/LSVerifier>

Best tool paper SBSeg 2023



A novel verification tool combining **input-code analysis** and **BMC technique** to detect **software vulnerabilities** for **Open-Source C software** projects.

7. LSVerifier Team



Mr. Janisley Oliveira
(UFAM/SIDIA)



Ph.D. Bruno Farias
(Manchester)



M.Sc. Thales Silva
(UFAM)



Dr. Eddie Batista
(UFAM/TPV)



Dr. Lucas Cordeiro
(UFAM/Manchester)



ESBMC



UFAM



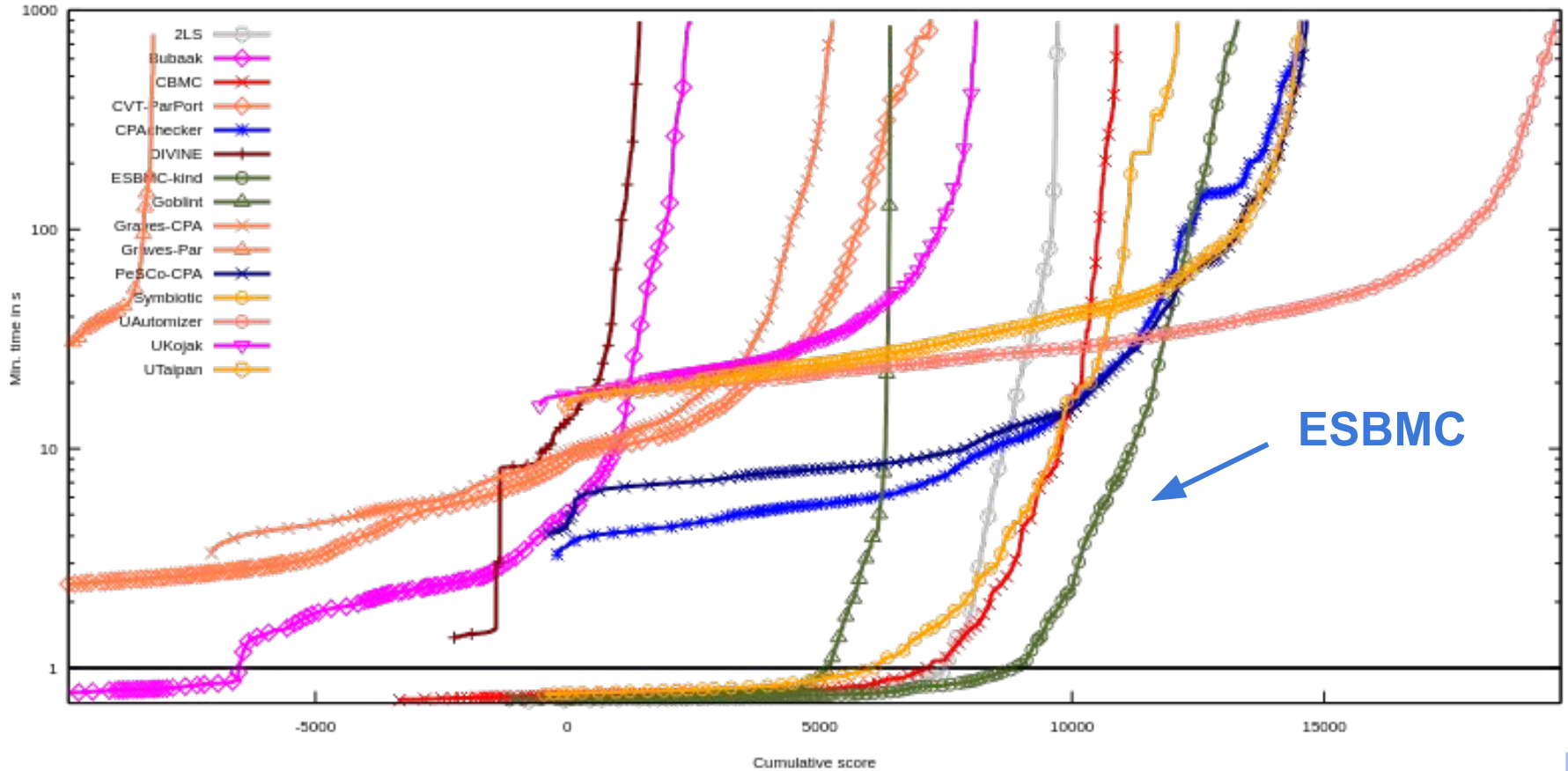
The University of Manchester

Verification Algorithm

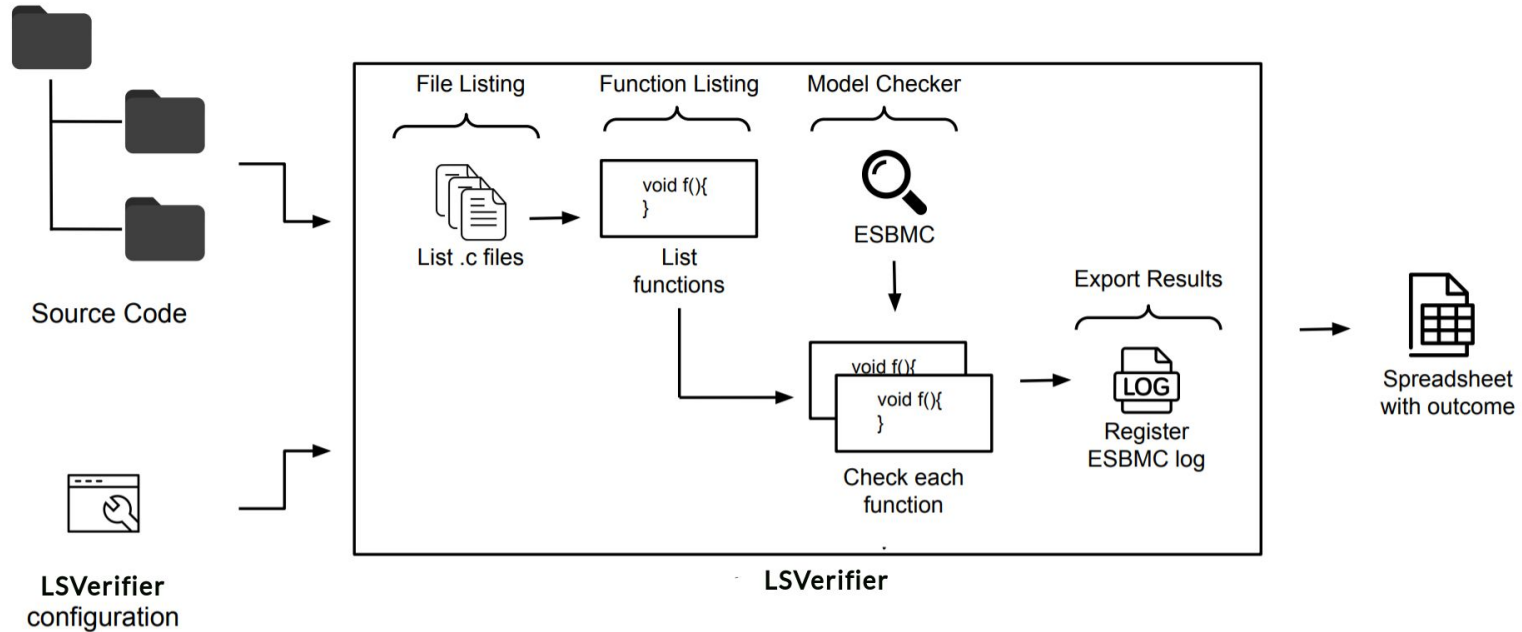
Description of the proposed approach.

- **CTAGS** is used to list all functions, variables, marcos, etc. in a C file.
- Generate a **AST (Abstract Syntax Tree)** with all classified data.
- Generate a **CFG (Control Flot Chat)**.
- ESBMC uses **Boolector** as **SMT solver** by default when none is specified in command line.
- **ESBMC**'s module is used to convert C programs into GOTO ones.
- Used a **State Machine** to analysed all data processed.
- LSVerifier is implemented in **Python** and ESBMC module is implemented in **C/C++** .
- The ESBMC module is used as **binary**.

7. SV-COMP 2023 - ESBMC module



Large Systems Verifier (LSVerifier) Architecture



- The Tool takes a **source-code directory**, a **software**, and **dependencies configuration** as inputs. It then lists all .c files and iterates through them to verify each function.
- The verification outcomes are compiled into a **report (logs, CSV files)**, which is returned as the final output.

LSVerifier – Property Verification Process

- More artefacts:

```
1 /usr/include/glib-2.0/  
2 /usr/lib/x86_64-linux-gnu/glib-2.0/include/  
3 extcap/  
4 plugins/epan/ethercat/  
5 plugins/epan/falco_bridge/  
6 plugins/epan/wimaxmacphy/  
7 randpkt_core/  
8 writecap/  
9 epan/crypt/  
10 ...
```

Project dependencies example (dep.txt).

Counterexample:

State 2 file main.c line 51 function vector_func thread 0

Violated property:

file main.c line 51 function vector_func
array bounds violated: array 'i' upper bound

Property violation log.

```
void vector_func(char c[]) {  
    c[2] = 'a';  
    int i[1];  
    i[2] = 1;  
}
```

Software with bug.

```
void vector_func(char c[]) {  
    c[2] = 'a';  
    int i[1];  
    i[0] = 1;  
}
```

Issue fixed.

LSVerifier – Property Verification Process

- Property verification for an entire project:

```
$ lsverifier -v -r -f -e "--unwind 1 --no-unwinding-assertions" -l dep.txt
```

- Property verification for specific .c files:

```
$ lsverifier -v -r -f -fl main.c
```

- Property verification for a specific path:

```
$ lsverifier -v -r -f -l dep.txt -d project-root/
```

- Property verification by specific class of vulnerability:

```
$ lsverifier -v -r -f -p memory-leak-check,overflow-check,deadlock-check,data-races-check
```

More details: <https://github.com/janisley/LSVerifier/blob/main/README.md>

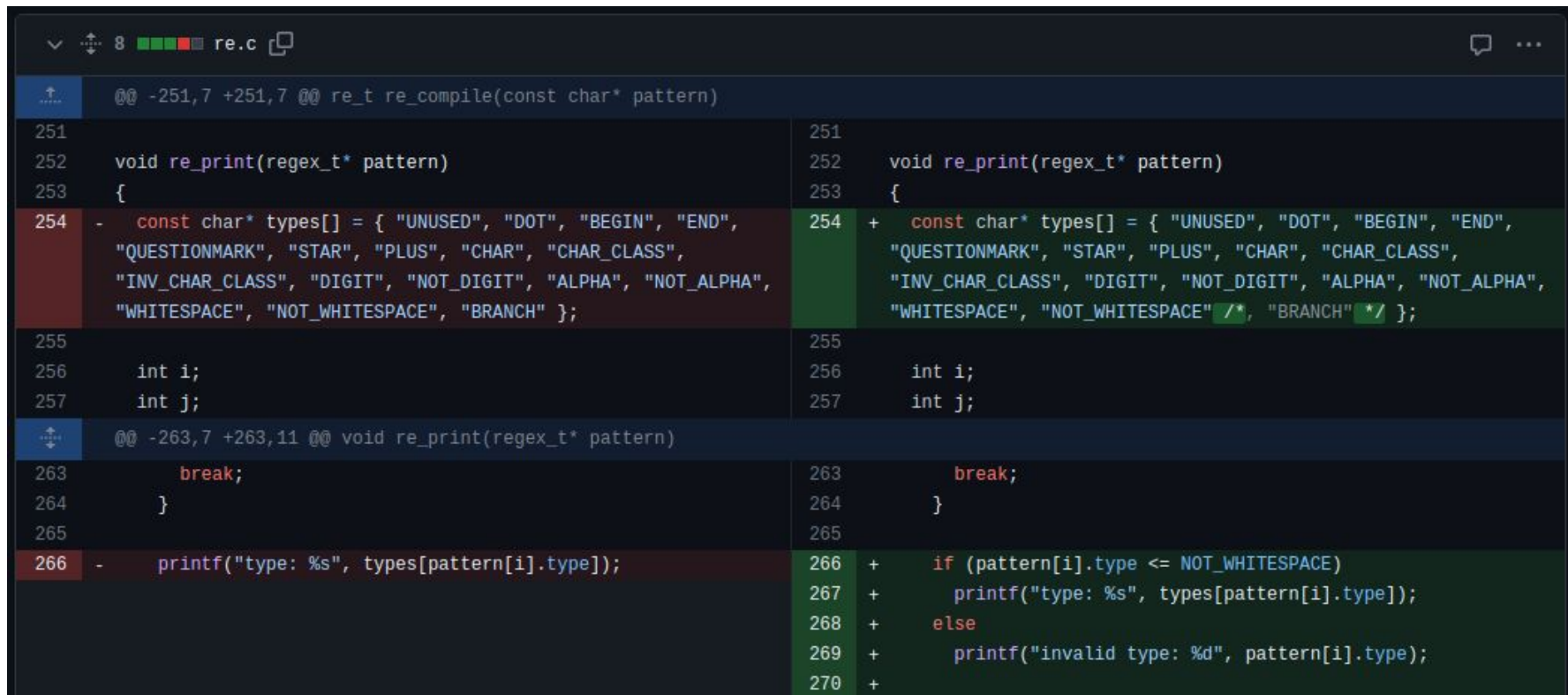
Experimental Results: Properties violated and CWE categories

Table 1. Dataset analysis using LSVerifier tool.

Software	Property violations	Files analyzed	Functions verified	Overall time	Peak memory usage
VIM	5	184	8804	406.02 s	36.46 MB
RUFUS	186	142	1575	101.59 s	32.6 MB
OpenSSH	337	286	3033	490.33 s	15.32 MB
Wireshark	122	2194	108824	39413.97 s	119.52 MB
PuTTY	2019	244	4575	91448.89 s	53.79 MB

- Dataset: https://github.com/janislley/LSVerifier_Benchmarks
- Issues reported and fixed:
 - RUFUS: <https://github.com/pbatard/rufus/issues/1856> (CWE-119)
 - Wireshark: <https://gitlab.com/wireshark/wireshark/-/issues/17897> (CWE-416)
- RUFUS presented property violations such as array out of bounds, with 3 issues opened and 1 fixed regarding imported libraries.
- The Wireshark's property violations, which are related to array out of bounds and invalid pointers, were due to errors in the NPL third-party library.

Experimental Results: Properties violated on RUFUS



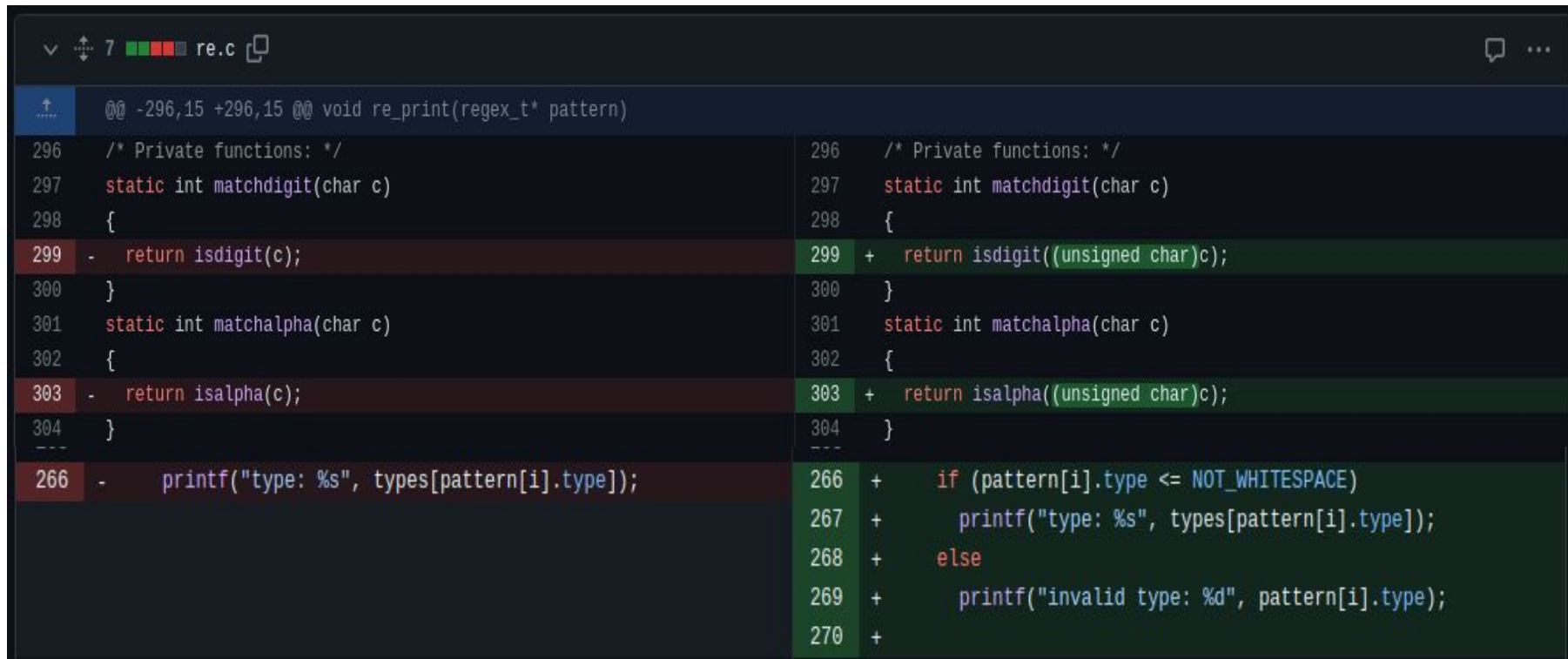
```
@@ -251,7 +251,7 @@ re_t re_compile(const char* pattern)
251
252 void re_print(regex_t* pattern)
253 {
254 - const char* types[] = { "UNUSED", "DOT", "BEGIN", "END",
+ const char* types[] = { "UNUSED", "DOT", "BEGIN", "END",
"QUESTIONMARK", "STAR", "PLUS", "CHAR", "CHAR_CLASS",
"INV_CHAR_CLASS", "DIGIT", "NOT_DIGIT", "ALPHA", "NOT_ALPHA",
"WHITESPACE", "NOT_WHITESPACE", "BRANCH" };
255
256 int i;
257 int j;
@@ -263,7 +263,11 @@ void re_print(regex_t* pattern)
263 break;
264 }
265
266 - printf("type: %s", types[pattern[i].type]);
266 + if (pattern[i].type <= NOT_WHITESPACE)
267 + printf("type: %s", types[pattern[i].type]);
268 + else
269 + printf("invalid type: %d", pattern[i].type);
270 +
```

Array bounds violated: array `types' upper bound fix (**CWE-119**).

This issue was **fixed** and others **9 fixes** were provide for another parts in re.c

in **tyne-regex third-party library**. More details: <https://github.com/kokke/tiny-regex-c/pull/78>

Experimental Results: Properties violated on RUFUS



```

 7  re.c
@@ -296,15 +296,15 @@ void re_print(regex_t* pattern)
296  /* Private functions: */
297  static int matchdigit(char c)
298  {
299  - return isdigit(c);
300  }
301  static int matchalpha(char c)
302  {
303  - return isalpha(c);
304  }
---
266  - printf("type: %s", types[pattern[i].type]);
---
296  /* Private functions: */
297  static int matchdigit(char c)
298  {
299  + return isdigit((unsigned char)c);
300  }
301  static int matchalpha(char c)
302  {
303  + return isalpha((unsigned char)c);
304  }
---
266  + if (pattern[i].type <= NOT_WHITESPACE)
267  +     printf("type: %s", types[pattern[i].type]);
268  + else
269  +     printf("invalid type: %d", pattern[i].type);
270  +

```

Array bounds violated: array `types` upper bound fix (**CWE-119**).

This issue was **fixed** and others **9 fixes** were provide for another parts in re.c

in **tyne-regex third-party library**. More details: <https://github.com/kokke/tiny-regex-c/pull/78>

Experimental Results: Properties violated on Wireshark

tools/npl

h ast.h

+0 -419

c npl.c

+0 -1993

parser.l

+0 -1429

h xmem.h

+0 -26

 **Tools: Remove NPL.**
Gerald Combs authored 1 year ago

6e48f973

Remove tools/npl. It doesn't appear to be used and hasn't had any activity for many years. Ping #17897.

tools/npl/ast.h deleted

100644 → 0

+0 -419

- **Dereference failure (invalid pointer and Null pointer)** issues (**CWE-416**) were found in the **NPL** third-party library.
- The fix involved removing this library, as it is no longer used in Wireshark.
- More details: https://gitlab.com/wireshark/wireshark/-/merge_requests/6021

Hands-On 1

Using LSVerifier

`pip install lsverifier`

https://github.com/janisley/AFRiTS_2023.git

Hands On 01

```
janislley@jos:Hands_On_01$ lsverifier -r -f -fl main.c
[LSVerifier] Loading configuration settings
[LSVerifier] Running ESBMC model checker
Checking main.c: 100%| 1/1

#####
Summary:

Files Verified: 1
Functions Verified: 9
Counterexamples: 2

Overall time: 0.76s
Peak Memory Usage: 0.48MB
#####

[LSVerifier] Verification completed
janislley@jos:Hands_On_01$ lsverifier -r -f -fl main_memory.c -p memory-leak-check,overflow-check,deadlock-check,data-races-check
[LSVerifier] Loading configuration settings
[LSVerifier] Running ESBMC model checker
Checking main_memory.c: 100%| 1/1

#####
Summary:

Files Verified: 1
Functions Verified: 8
Counterexamples: 3

Overall time: 2642.07s
Peak Memory Usage: 2.0MB
#####

[LSVerifier] Verification completed
```

Hands-On 2

Using Bounded Model Checking to exploit Linux Device Drivers

`git clone https://github.com/janisley/AFRiTS_2023.git`

Hands On 02

```
janislley@jos:Hands_On_02$ lsverifier -r -f -fl cve-2021-3156.c
[LSVerifier] Loading configuration settings
[LSVerifier] Running ESBMC model checker
Checking cve-2021-3156.c: 100%| 1/1

#####
Summary:

Files Verified: 1
Functions Verified: 5
Counterexamples: 2

Overall time: 0.79s
Peak Memory Usage: 0.48MB
#####

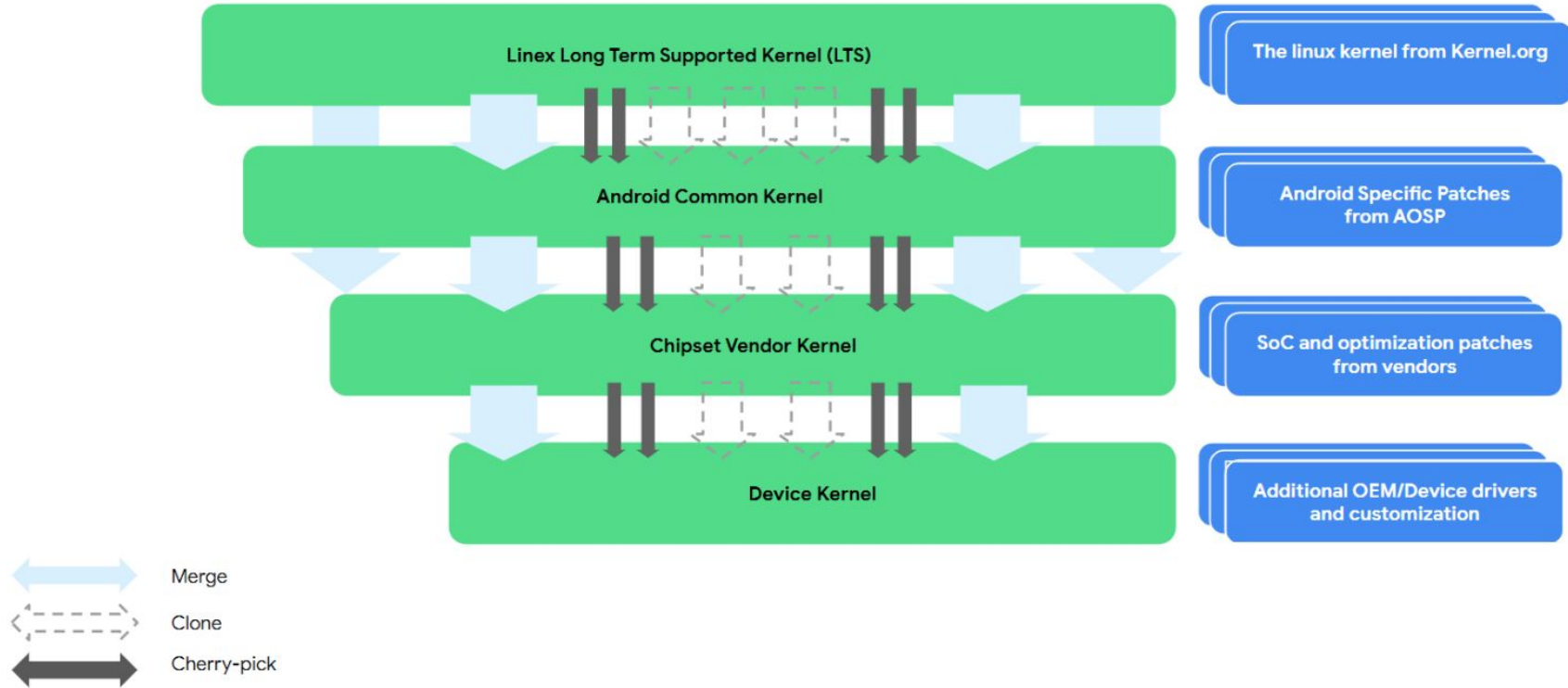
[LSVerifier] Verification completed
```

Google - Android

Project Zero

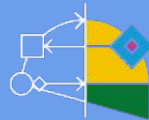


Google Project Zero





SBMF 2023
26th Brazilian Symposium on
Formal Methods



AFRITS 2023
Workshop on automated Formal
Reasoning for Trustworthy AI systems
Manaus - Brazil

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

Thank You | Obrigado

Email: janisley.sousa@sidia.com



ESBMC

