Automatically Proving Microkernel Security

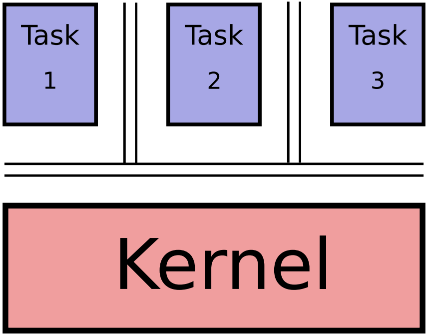
Olivier Nicole

Co-supervisors: Matthieu Lemerre[1], Xavier Rival[2]

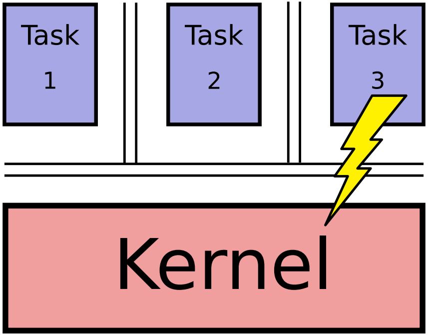
[1]CEA List

[2]ENS

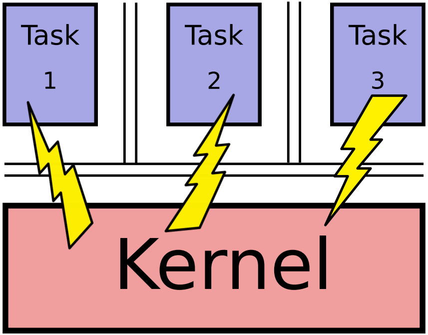
RESSI, Dec. 2020



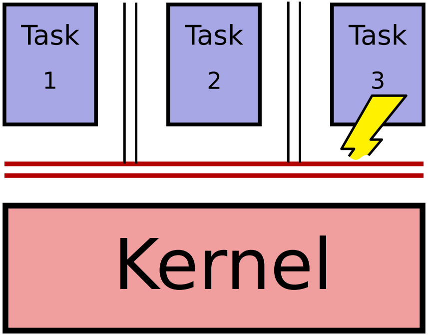
Kernels are critical to security.



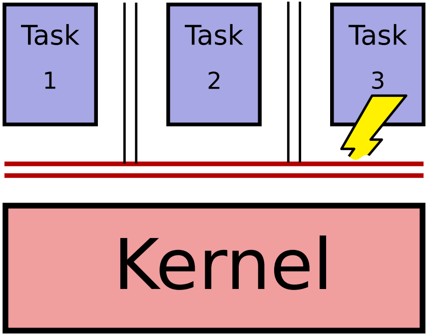
Kernels are critical to security.



Kernels are critical to security.



Kernels are critical to security.



Kernels are critical to security.

Our goal:

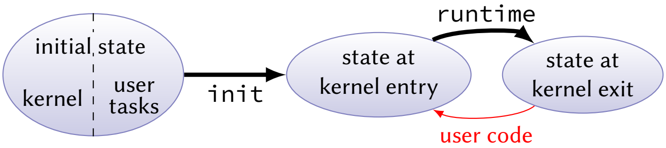
- prove absence of privilege escalation (APE).
- prove absence of run-time errors (ARTE).

Automatically, with few annotations, from the machine code

Scope

We focus on embedded kernels.

- No dynamic allocation



Protection mechanisms

- page tables, or
- segments (32-bit x86)

Specify (address range, permissions) pairs.
To verify: at kernel exit, the memory protection state is correct.
Requires analyzing all the kernel's code.

Challenge 1: machine code analysis

```c
void hw_context_idle(void) {
  struct context *high = context_idle();
  struct hw_context *ctx = &high->hw_context;

  asm volatile
    ("mov %0,%%esp" : : "r"((uintptr_t) ctx + sizeof(struct pusha)
                          + sizeof(struct intra_privilege_interrupt_frame))
                    : "memory");
  asm("sti");
  asm("hlt");
  asm("jmp error_infinite_loop");
  __builtin_unreachable ();
}
```

Challenge 1: machine code analysis

- no control flow
- no types, no memory partitioning
- masks, legitimate overflows, low-level comparisons, etc.

Challenge 1: machine code analysis

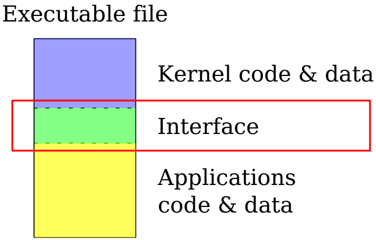
- no control flow
- no types, no memory partitioning
- masks, legitimate overflows, low-level comparisons, etc.

Contribution

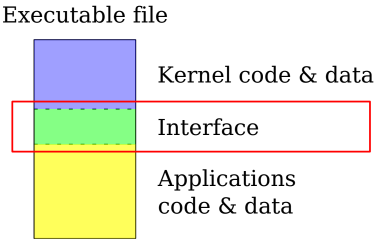
BINSEC/CODEX, a static analyzer based on **abstract interpretation**.

- Computes CFG on-the-fly
- Rich numerical abstractions + symbolic information

Challenge 2: parameterized kernels



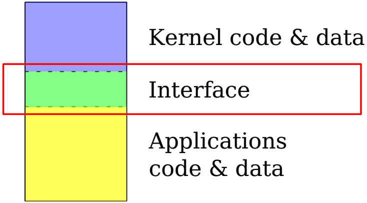
Challenge 2: parameterized kernels




```
type Flags = Int8 with (self & PRIVILEGED) == 0;
type Context = struct { Int8 pc; Int8 sp; Flags flags; };

type Segment = struct {
  Int8 base;
  Int8 size_and_rights;
} with self.base ≥ kernel_last_addr

type Memory_Table = struct { Segment code; Segment data; }

type Thread = struct {
  Memory_Table ∗mt;
  Context ctx;
  Thread ∗next; }

type Interface = struct {
    Thread[nb_thread]∗ threads;
    (Int8 with self = nb_threads) threads_length; }
```

Challenge 2: parameterized kernels

Executable file



Contribution

A type-based memory analysis

```
type Flags = Int8 with (self & PRIVILEGED) == 0;
type Context = struct { Int8 pc; Int8 sp; Flags flags; };

type Segment = struct {
  Int8 base;
  Int8 size_and_rights;
} with self.base ≥ kernel_last_addr

type Memory_Table = struct { Segment code; Segment data; }

type Thread = struct {
  Memory_Table *mt;
  Context ctx;
  Thread *next; }

type Interface = struct {
   Thread[nb_thread]* threads;
   (Int8 with self = nb_threads) threads_length; }
```

Experimental evaluation

Case study 1: EducRTOS

- Small academic OS developed for teaching purposes
- Both separation kernel and real-time OS, dynamic thread creation
- 1,200 x86 instructions.
- Protection by segmentation.

Proved APE and ARTE on 96 variants of EducRTOS. Varying parameters:

- compiler (GCC/Clang), optimization flags
- scheduling algorithm, dynamic thread creation (enabled/disabled)...

Verification time: from 1.6 s to 73 s.
14 lines of annotations.

Experimental evaluation

Case study 2: AnonymOS

- Industrial microkernel used in industrial settings
- Multicore
- 329 functions, ~10,000 instructions
- Protection using page tables.

2 versions

- BETA version: 1 vulnerability
- v1 version: vulnerability fixed

Specific = *Generic* + restriction on stack sizes

		Generic annotations		*Specific* annotations	
# shape annotations	generated	1057			
	manual	57 (5.11%)		58 (5.20%)	
Kernel version		BETA	v1	BETA	v1
invariant computation	status	✓	✓	✓	✓
	time (s)	647	417	599	406
# alarms in runtime		1 **true error** 2 false alarms	1 false alarm	1 **true error** 1 false alarm	**0** ✓
user tasks checking	status	✓	✓	✓	✓
	time (s)	32	29	31	30
Proves APE?		N/A	∼	N/A	✓

Proved APE and ARTE in 430 s.
58 lines of annotations.

Summary and perspectives

- Verification of APE and ARTE on kernels is possible
- from the binary
- unbounded loops
- small number of manual annotations.

Submission under review: O. Nicole, M. Lemerre, S. Bardin, X. Rival, “No Crash, No Exploit: Automated Verification of Embedded Kernels”, *arXiv:2011.15065*

Perspectives

- Non-interference between tasks
- Analysis of other software
- Efficient verification of memory safety (70 % of vulnerabilities[1])
- Mixed-language analysis, off-the-shelf component analysis

[1] M. Miller, Trends, challenges, and strategic shifts in the software vulnerability mitigation landscape, *BlueHat IL*, 2016