No Crash, No Exploit: Automated Verification of Embedded Kernels

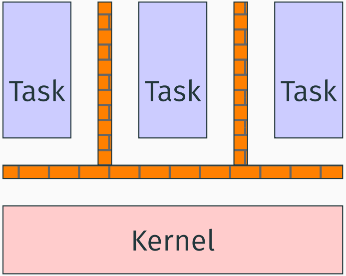
Olivier Nicole [1,2] Matthieu Lemerre [1] Sébastien Bardin [1] Xavier Rival [2]

RTAS 2021

[1] Université Paris-Saclay, CEA List

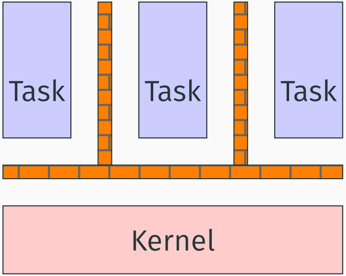
[2] ENS, PSL University / Inria

How to protect an OS kernel against its worst defects?



Worst possible bugs for an OS kernel:

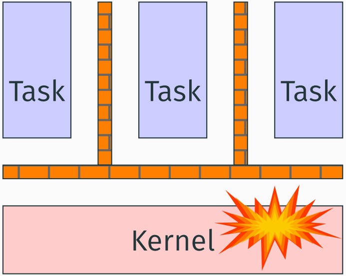
How to protect an OS kernel against its worst defects?



Worst possible bugs for an OS kernel:

- Runtime errors Division by zero, illegal memory access...

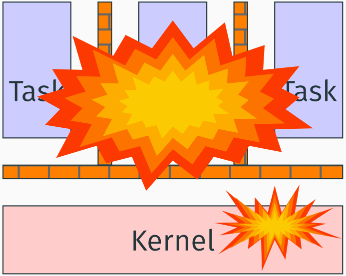
How to protect an OS kernel against its worst defects?



Worst possible bugs for an OS kernel:

- Runtime errors   Division by zero, illegal memory access...
  The kernel **crashes**

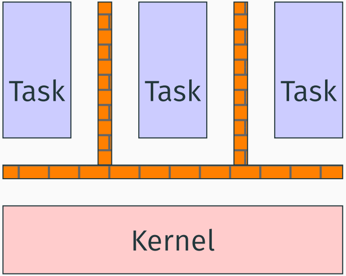
How to protect an OS kernel against its worst defects?



Worst possible bugs for an OS kernel:

- Runtime errors   Division by zero, illegal memory access...
  The kernel **crashes** $\implies$ the whole system crashes

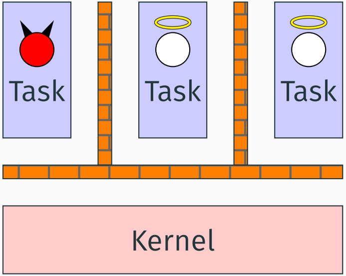
How to protect an OS kernel against its worst defects?



Worst possible bugs for an OS kernel:

- Runtime errors   Division by zero, illegal memory access...
  The kernel **crashes** $\Longrightarrow$ the whole system crashes

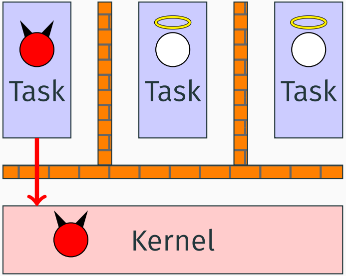
How to protect an OS kernel against its worst defects?



Worst possible bugs for an OS kernel:

- Runtime errors   Division by zero, illegal memory access...
    The kernel **crashes** $\implies$ the whole system crashes
- Privilege escalation

How to protect an OS kernel against its worst defects?



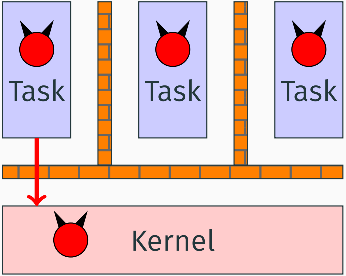
Worst possible bugs for an OS kernel:

- Runtime errors   Division by zero, illegal memory access...

  The kernel **crashes** $\Longrightarrow$ the whole system crashes
- Privilege escalation

  Kernel protections are **bypassed**

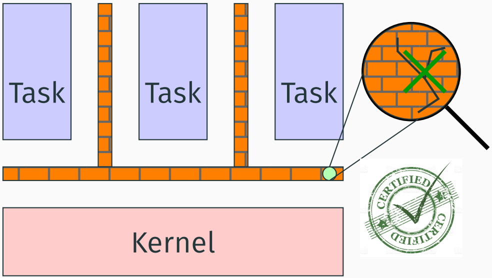
How to protect an OS kernel against its worst defects?



Worst possible bugs for an OS kernel:

- Runtime errors Division by zero, illegal memory access...
  The kernel **crashes** $\Longrightarrow$ the whole system crashes
- Privilege escalation
  Kernel protections are **bypassed** $\Longrightarrow$ the whole system is compromised

How to protect an OS kernel against its worst defects?



Worst possible bugs for an OS kernel:

- Runtime errors Division by zero, illegal memory access...

  The kernel **crashes** $\Longrightarrow$ the whole system crashes
- Privilege escalation

  Kernel protections are **bypassed** $\Longrightarrow$ the whole system is compromised

Only way to guarantee their absence: formal methods.

Goals

We want a verification of

- absence of run-time errors **(ARTE)**, and
- absence of privilege escalation **(APE)**

that is:

- **Automated**
- **Comprehensive**
- **Generic**
- **Practical**

Automated

```
int max_seq(int* p, int n) {
  int res =*p;
  //@ ghost int e = 0;
  /*@ loop invariant \forall integer j; 0 <= j < i ==> res >=\at(p[j],Pre);
      loop invariant \valid(\at(p,Pre)+e) && \at(p,Pre)[e] == res;
      loop invariant 0 <= i <= n;
      loop invariant p == \at(p,Pre)+i;
      loop invariant 0 <= e < n; */
  for(int i = 0; i < n; i++) {
    if(res <*p) {
      res =*p;
      //@ghost e = i;
    }
    p++;
  }
  return res;
}
```

- Avoid manual annotations

Comprehensive

```
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 ();
}
```

- Check all the code (including boot and assembly sections)
- End-to-end verification, without trusting the compiler

Generic

$$\forall \text{ tasks}, (\text{kernel} \oplus \text{tasks}) \vDash \text{APE}, \text{ARTE}$$

- Verify kernel **independently from the tasks**
- No fundamental restriction (e.g. **allow unbounded loops**)

Practical

- Works on real-world, existing kernels without modification.

Contributions

BINSEC/CODEX, a static analyzer to verify **APE** and **ARTE** on **embedded kernels**.

- **Automated**
  - Abstract interpretation on the **system loop** to **infer** kernel invariants
  - APE is an implicit property (**no specification needed**)
- **Comprehensive**
  - **Machine code** verification on the kernel executable
- **Generic**
  - **Parameterized** verification (i.e. independent from the applications)
  - Using a **type-based** memory analysis
- **Practical**
  - **Different treatment** of boot code and runtime code
  - Comprehensive evaluation on challenging case studies
    unmodified version of ASTERIOS RTK, 96 variants of EducRTOS

Positioning wrt. the verification technique

Interactive proof

- seL4 [SOSP'09]
- CertiKOS [OSDI'16]

Deductive verification

- Verve [PLDI'10]
- Komodo [SOSP'17]

Proves strong properties, but requires huge **expertise** and **effort**.

"Push-button" verification

- PROSPER [CCS'13]
- Serval [SOSP'19]
- Phidias [EuroSys'20]

- Still require to write hundreds of kernel invariants
- Only support **bounded loops** (no priority scheduling)
- Requires a **fixed memory layout** (depends on the number of tasks)

Positioning wrt. the verification technique

Interactive proof

- seL4 [SOSP'09]
- CertiKOS [OSDI'16]

Deductive verification

- Verve [PLDI'10]
- Komodo [SOSP'17]

Proves strong properties, but requires huge **expertise** and **effort**.

"Push-button" verification

- PROSPER [CCS'13]
- Serval [SOSP'19]
- Phidias [EuroSys'20]

- Still require to write hundreds of kernel invariants
- Only support **bounded loops** (no priority scheduling)
- Requires a **fixed memory layout** (depends on the number of tasks)

Us: Abstract interpretation

- ASTERIOS

- Infers all invariants
- Handles unbounded loops
- Handles parameterized verification
- Low annotation burden (e.g. 58 lines)

Verification principle

Abstract interpretation basics

Abstract each **numeric variable** by an **interval**.

```
int i = 100;
int x = 0;
while(i > 1) {
  i--;
}
int x = 42 / i;
```

Abstract each **numeric variable** by an **interval**.

```
int i = 100;
int x = 0;
while(i > 1) {
  i--;
}
int x = 42 / i;
```

$i \in \{100\}$

Abstract interpretation basics

Abstract each **numeric variable** by an **interval**.

```
int i = 100;
int x = 0;
while(i > 1) {
  i--;
}
int x = 42 / i;
```

$i \in \{100\}$

$i \in \{100\}, x \in \{0\}$

Abstract interpretation basics

Abstract each **numeric variable** by an **interval**.

```
int i = 100;
int x = 0;
while(i > 1) {
  i--;
}
int x = 42 / i;
```

$i \in \{100\}$

$i \in \{100\}$, $x \in \{0\}$

$i \in \{100\}$, $x \in \{0\}$

Abstract interpretation basics

Abstract each **numeric variable** by an **interval**.

```
int i = 100;
int x = 0;
while(i > 1) {
  i--;
}
int x = 42 / i;
```

$i \in \{100\}$

$i \in \{100\}$, $x \in \{0\}$

$i \in \{100\}$, $x \in \{0\}$

$i \in \{99\}$, $x \in \{0\}$

Abstract interpretation basics

Abstract each **numeric variable** by an **interval**.

```
int i = 100;
int x = 0;
while(i > 1) {
  i--;
}
int x = 42 / i;
```

$i \in \{100\}$

$i \in \{100\}$, $x \in \{0\}$

$i \in [99, 100]$, $x \in \{0\}$

$i \in \{99\}$, $x \in \{0\}$

Abstract interpretation basics

Abstract each **numeric variable** by an **interval**.

```
int i = 100;
int x = 0;
while(i > 1) {
  i--;
}
int x = 42 / i;
```

- `int i = 100;` — $i \in \{100\}$
- `int x = 0;` — $i \in \{100\}$, $x \in \{0\}$
- `while(i > 1) {` — $i \in [99, 100]$, $x \in \{0\}$
- `i--;` — $i \in [98, 99]$, $x \in \{0\}$

Abstract interpretation basics

Abstract each **numeric variable** by an **interval**.

```
int i = 100;
int x = 0;
while(i > 1) {
  i--;
}
int x = 42 / i;
```

$i \in \{100\}$

$i \in \{100\},\ x \in \{0\}$

$i \in [98, 100],\ x \in \{0\}$

$i \in [98, 99],\ x \in \{0\}$

Abstract interpretation basics

Abstract each **numeric variable** by an **interval**.

```
int i = 100;
int x = 0;
while(i > 1) {
  i--;
}
int x = 42 / i;
```

$i \in \{100\}$

$i \in \{100\},\ x \in \{0\}$

$i \in [98, 100],\ x \in \{0\}$

$i \in [97, 99],\ x \in \{0\}$

Abstract interpretation basics

Abstract each **numeric variable** by an **interval**.

```
int i = 100;
int x = 0;
while(i > 1) {
  i--;
}
int x = 42 / i;
```

$i \in \{100\}$

$i \in \{100\}$, $x \in \{0\}$

$i \in [2, 100]$, $x \in \{0\}$

$i \in [1, 99]$, $x \in \{0\}$

Abstract interpretation basics

Abstract each **numeric variable** by an **interval**.

```
int i = 100;
int x = 0;
while(i > 1) {
  i--;
}
int x = 42 / i;
```

$i \in \{100\}$

$i \in \{100\}$, $x \in \{0\}$

$i \in [2, 100]$, $x \in \{0\}$

$i \in [1, 99]$, $x \in \{0\}$

$i \in \{1\}$, $x \in \{0\}$

Abstract interpretation basics

Abstract each **numeric variable** by an **interval**.

`int i = 100;`	$i \in \{100\}$
`int x = 0;`	$i \in \{100\},\ x \in \{0\}$
`while(i > 1) {`	$i \in [2, 100],\ x \in \{0\}$
`  i--;`	$i \in [1, 99],\ x \in \{0\}$
`}`	$i \in \{1\},\ x \in \{0\}$
`int x = 42 / i;`	$i \in \{1\},\ x \in \{42\}$

Abstract interpretation basics

Abstract each **numeric variable** by an **interval**.

Code	Abstract state
`int i = 100;`	$i \in \{100\}$
`int x = 0;`	$i \in \{100\},\ x \in \{0\}$
`while(i > 1) {`	$i \in [2, 100],\ x \in \{0\}$
`  i--;`	$i \in [1, 99],\ x \in \{0\}$
`}`	$i \in \{1\},\ x \in \{0\}$
`int x = 42 / i;`	$i \in \{1\},\ x \in \{42\}$

invariant

Abstract interpretation basics

Abstract each **numeric variable** by an **interval**.

```
int i = 100;
int x = 0;
while(i > 1) {
  i--;
}
int x = 42 / i;
```

$i \in \{100\}$

$i \in \{100\},\ x \in \{0\}$

$i \in [2, 100],\ x \in \{0\}$

$i \in [1, 99],\ x \in \{0\}$

$i \in \{1\},\ x \in \{0\}$

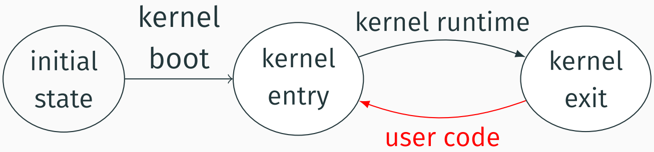
$i \in \{1\},\ x \in \{42\}$

invariant

- Abstract interpretation can **prove** properties. Here: no division by zero.
- No specification required for this property (it is **implicit**)

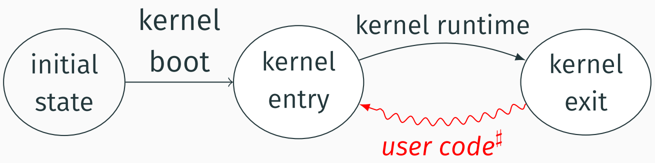
Absence of run-time errors (ARTE) is an implicit property.

The system loop



Alternation of **user code** and **kernel runtime**.

The system loop: Empowering the attacker



Alternation of **user code** and **kernel runtime**.

The user code is unknown

$\Longrightarrow$ We abstract it by "arbitrary sequences of instructions" (whose execution is permitted by the hardware).

Main hardware protection mechanisms

- Memory protection
- Hardware privilege level

Absence of Privilege Escalation is an implicit property

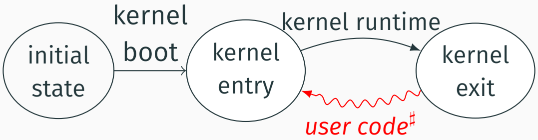
Theorem

If the system satisfies a non-trivial invariant,
then no privilege escalation is possible on that system.

Proof.

If the systems fails to self-protect, the empowered attacker can reach any state. $\square$

$\Longrightarrow$ APE can be verified without writing a specification.

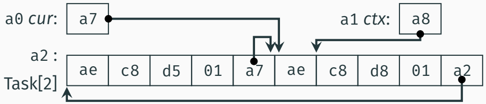



```
Task *cur; Context *ctx;

runtime() {
  save_context();
  /* Schedule next task */
  cur = cur->next;
  ctx = &cur->ctx;
  load_protection();
  load_context();
}
```

```
struct Context { Int8 pc, sp, flags; };

struct Task {
  Memory_table * mem_table;
  Context ctx;
  Task * next;
};
```

Example in-context analysis

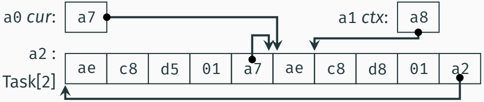
Initial state:




```
Task *cur; Context *ctx;

runtime() {
  save_context();
  /* Schedule next task */
  cur = cur→next;
  ctx = &cur→ctx;
  load_protection();
  load_context();
}
```

Example in-context analysis

Initial state:

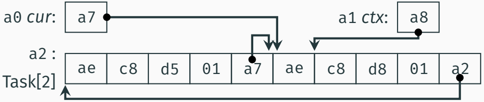



```
Task *cur; Context *ctx;

runtime() {
  save_context();
  /* Schedule next task */
  cur = cur→next;
  ctx = &cur→ctx;
  load_protection();
  load_context();
}
```

$\mathtt{cur} \in \{0xa7\},\ \mathtt{ctx} \in \{0xa8\}$

Example in-context analysis

Initial state:



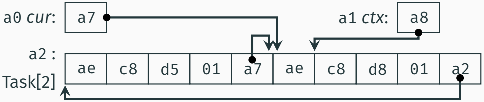

```
Task *cur; Context *ctx;

runtime() {
  save_context();
  /* Schedule next task */
  cur = cur→next;
  ctx = &cur→ctx;
  load_protection();
  load_context();
}
```

$\mathtt{cur} \in \{0xa7\}$, $\mathtt{ctx} \in \{0xa8\}$

$\mathtt{cur} \in \{0xa7\}$, $\mathtt{ctx} \in \{0xa8\}$

Example in-context analysis

Initial state:




```
Task *cur; Context *ctx;

runtime() {
  save_context();
  /* Schedule next task */
  cur = cur→next;
  ctx = &cur→ctx;
  load_protection();
  load_context();
}
```

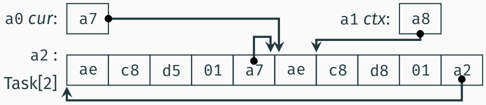
runtime() { — $\texttt{cur} \in \{0xa7\}$, $\texttt{ctx} \in \{0xa8\}$

save_context(); — $\texttt{cur} \in \{0xa7\}$, $\texttt{ctx} \in \{0xa8\}$

cur = cur→next; — $\texttt{cur} \in \{0xa2\}$, $\texttt{ctx} \in \{0xa8\}$

Example in-context analysis

Initial state:




```
Task *cur; Context *ctx;

runtime() {
  save_context();
  /* Schedule next task */
  cur = cur→next;
  ctx = &cur→ctx;
  load_protection();
  load_context();
}
```

runtime() { — $\mathtt{cur} \in \{0xa7\}$, $\mathtt{ctx} \in \{0xa8\}$

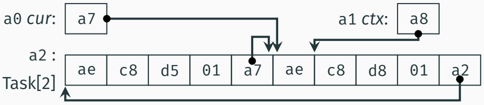
save_context(); — $\mathtt{cur} \in \{0xa7\}$, $\mathtt{ctx} \in \{0xa8\}$

cur = cur→next; — $\mathtt{cur} \in \{0xa2\}$, $\mathtt{ctx} \in \{0xa8\}$

ctx = &cur→ctx; — $\mathtt{cur} \in \{0xa2\}$, $\mathtt{ctx} \in \{0xa3\}$

Example in-context analysis

Initial state:

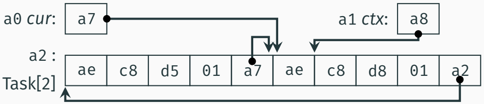



```
Task *cur; Context *ctx;
```

Code	State
`runtime() {`	cur $\in \{0xa7\}$, ctx $\in \{0xa8\}$
`  save_context();`	cur $\in \{0xa7\}$, ctx $\in \{0xa8\}$
`  /* Schedule next task */`	
`  cur = cur->next;`	cur $\in \{0xa2\}$, ctx $\in \{0xa8\}$
`  ctx = &cur->ctx;`	cur $\in \{0xa2\}$, ctx $\in \{0xa3\}$
`  load_protection();`	cur $\in \{0xa2\}$, ctx $\in \{0xa3\}$
`  load_context();`	and kernel is protected
`}`	

Example in-context analysis

Initial state:



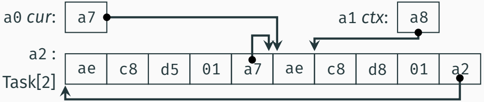

```
Task *cur; Context *ctx;

runtime() {
  save_context();
  /* Schedule next task */
  cur = cur→next;
  ctx = &cur→ctx;
  load_protection();
  load_context();
}
```

`runtime() {`	$\texttt{cur} \in \{0xa7\}$, $\texttt{ctx} \in \{0xa8\}$
`save_context();`	$\texttt{cur} \in \{0xa7\}$, $\texttt{ctx} \in \{0xa8\}$
`cur = cur→next;`	$\texttt{cur} \in \{0xa2\}$, $\texttt{ctx} \in \{0xa8\}$
`ctx = &cur→ctx;`	$\texttt{cur} \in \{0xa2\}$, $\texttt{ctx} \in \{0xa3\}$
`load_protection();`	$\texttt{cur} \in \{0xa2\}$, $\texttt{ctx} \in \{0xa3\}$ and kernel is protected

user code$^\sharp$

Example in-context analysis

Initial state:




```
Task *cur; Context *ctx;

runtime() {
  save_context();
  /* Schedule next task */
  cur = cur→next;
  ctx = &cur→ctx;
  load_protection();
  load_context();
}
```

$\texttt{cur} \in \{0xa7, 0xa2\}$, $\texttt{ctx} \in \{0xa8, 0xa3\}$

$\texttt{cur} \in \{0xa7\}$, $\texttt{ctx} \in \{0xa8\}$

$\texttt{cur} \in \{0xa2\}$, $\texttt{ctx} \in \{0xa8\}$

$\texttt{cur} \in \{0xa2\}$, $\texttt{ctx} \in \{0xa3\}$

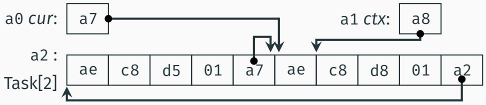
$\texttt{cur} \in \{0xa2\}$, $\texttt{ctx} \in \{0xa3\}$

and kernel is protected

user code$^\sharp$

Example in-context analysis

Initial state:




```
Task *cur; Context *ctx;

runtime() {
  save_context();
  /* Schedule next task */
  cur = cur→next;
  ctx = &cur→ctx;
  load_protection();
  load_context();
}
```

runtime() { — $\texttt{cur} \in \{0xa7, 0xa2\},\ \texttt{ctx} \in \{0xa8, 0xa3\}$

save_context(); — $\texttt{cur} \in \{0xa7, 0xa2\},\ \texttt{ctx} \in \{0xa8, 0xa3\}$

cur = cur→next; — $\texttt{cur} \in \{0xa2\},\ \texttt{ctx} \in \{0xa8\}$

ctx = &cur→ctx; — $\texttt{cur} \in \{0xa2\},\ \texttt{ctx} \in \{0xa3\}$

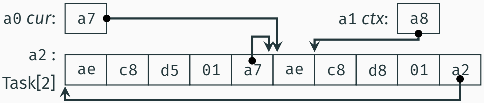
load_protection(); — $\texttt{cur} \in \{0xa2\},\ \texttt{ctx} \in \{0xa3\}$

and kernel is protected

user code$^{\sharp}$

Example in-context analysis

Initial state:



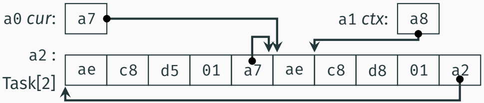

```
Task *cur; Context *ctx;
```

Code	State
`runtime() {`	$\texttt{cur} \in \{0xa7, 0xa2\}$, $\texttt{ctx} \in \{0xa8, 0xa3\}$
`  save_context();`	$\texttt{cur} \in \{0xa7, 0xa2\}$, $\texttt{ctx} \in \{0xa8, 0xa3\}$
`  /* Schedule next task */`	
`  cur = cur→next;`	$\texttt{cur} \in \{0xa2, 0xa7\}$, $\texttt{ctx} \in \{0xa3, 0xa8\}$
`  ctx = &cur→ctx;`	$\texttt{cur} \in \{0xa2\}$, $\texttt{ctx} \in \{0xa3\}$
`  load_protection();`	$\texttt{cur} \in \{0xa2\}$, $\texttt{ctx} \in \{0xa3\}$
`  load_context();`	and kernel is protected
`}`	

user code$^\sharp$

Example in-context analysis

Initial state:



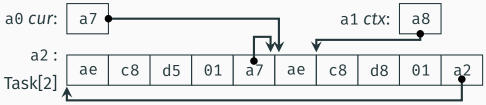

```
Task *cur; Context *ctx;
```

`runtime() {`	$\mathtt{cur} \in \{0xa7, 0xa2\},\ \mathtt{ctx} \in \{0xa8, 0xa3\}$
`  save_context();`	$\mathtt{cur} \in \{0xa7, 0xa2\},\ \mathtt{ctx} \in \{0xa8, 0xa3\}$
`  /* Schedule next task */`	
`  cur = cur→next;`	$\mathtt{cur} \in \{0xa2, 0xa7\},\ \mathtt{ctx} \in \{0xa3, 0xa8\}$
`  ctx = &cur→ctx;`	$\mathtt{cur} \in \{0xa2, 0xa7\},\ \mathtt{ctx} \in \{0xa3, 0xa8\}$
`  load_protection();`	$\mathtt{cur} \in \{0xa2\},\ \mathtt{ctx} \in \{0xa3\}$
`  load_context();`	and kernel is protected
`}`	

user code$^\sharp$

Example in-context analysis

Initial state:




```
Task *cur; Context *ctx;
```

`runtime() {`	$\texttt{cur} \in \{0xa7, 0xa2\}, \ \texttt{ctx} \in \{0xa8, 0xa3\}$
`  save_context();`	$\texttt{cur} \in \{0xa7, 0xa2\}, \ \texttt{ctx} \in \{0xa8, 0xa3\}$
`  /* Schedule next task */`	
`  cur = cur→next;`	$\texttt{cur} \in \{0xa2, 0xa7\}, \ \texttt{ctx} \in \{0xa3, 0xa8\}$
`  ctx = &cur→ctx;`	$\texttt{cur} \in \{0xa2, 0xa7\}, \ \texttt{ctx} \in \{0xa3, 0xa8\}$
`  load_protection();`	$\texttt{cur} \in \{0xa2, 0xa7\}, \ \texttt{ctx} \in \{0xa3, 0xa8\}$
`  load_context();`	and kernel is protected
`}`	

user code$^\sharp$

Example in-context analysis

> BINSEC/CODEX can verify APE and ARTE of such small kernels with 0 lines of annotations.

Abstractions we use:

- **Control flow:** Incremental CFG recovery
- **Values:** Non-relational numeric domains with symbolic relational information
- **Memory:** Byte-level memory manipulation
- **Concurrency:** Flow-insensitive abstraction of shared memory zones

Parameterized analysis

Shortcomings of in-context analyses

The method is:

- **Not generic:** Cannot analyze kernel independently from the applications
- **Not scalable:** 1000 tasks $\implies$ 1000 addresses to enumerate.

Key idea

Part of memory needs to be **summarized**.

We summarize **task data** using **types**.

Type system: a few examples

Types refined with **predicates**.

```
type Flags = Int8 with
  (self & PRIVILEGED) == 0

type Context = struct {
  Int8 pc; Int8 sp;
  Flags flags;
}

type Task = struct {
  Memory_table* mem_table;
  Context ctx;
  Task* next;
}
```

Each type t has an **interpretation** $(\!|t|\!)$ as a set of values.

E.g.

$$(\!|\mathtt{Task*}|\!) = \{0xa2, 0xa7\}$$

$$(\!|\mathtt{Flags}|\!) = \{x \mid x \,\&\, \mathtt{PRIVILEGED} = 0\}$$

Type system: a few examples

Types refined with **predicates**.

```
type Flags = Int8 with
  (self & PRIVILEGED) == 0

type Context = struct {
  Int8 pc; Int8 sp;
  Flags flags;
}

type Task = struct {
  Memory_table* mem_table;
  Context ctx;
  Task* next;
}
```

Each type t has an **interpretation** $(\!|t|\!)$ as a set of values.

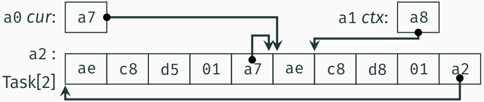
E.g.

$$(\!|\texttt{Task*}|\!) = \{0xa2, 0xa7\}$$

$$(\!|\texttt{Flags}|\!) = \{x \mid x \mathbin{\&} \texttt{PRIVILEGED} = 0\}$$

Example parameterized analysis

Initial state:

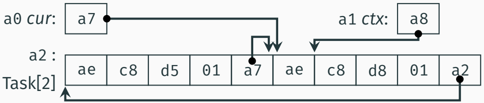


$$(\!|\texttt{Task*}|\!) = \{0xa2, 0xa7\}$$
$$(\!|\texttt{Context*}|\!) = \{0xa3, 0xa8\}$$

```
Task *cur; Context *ctx;

runtime() {
  save_context();
  /* Schedule next task */
  cur = cur→next;
  ctx = &cur→ctx;
  load_protection();
  load_context();
}
```

Example parameterized analysis

Initial state:



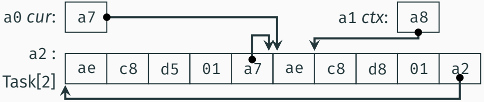
$$(\!|\texttt{Task}*|\!) = \{0xa2, 0xa7\}$$
$$(\!|\texttt{Context}*|\!) = \{0xa3, 0xa8\}$$

```
Task *cur; Context *ctx;

runtime() {
  save_context();
  /* Schedule next task */
  cur = cur→next;
  ctx = &cur→ctx;
  load_protection();
  load_context();
}
```

$\texttt{cur} \in (\!|\texttt{Task}*|\!),\ \texttt{ctx} \in (\!|\texttt{Context}*|\!)$

Example parameterized analysis

Initial state:



$$(\!|\texttt{Task}*|\!) = \{0xa2, 0xa7\}$$
$$(\!|\texttt{Context}*|\!) = \{0xa3, 0xa8\}$$

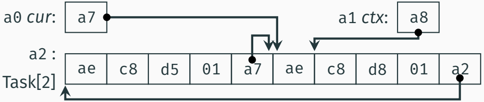
```
Task *cur; Context *ctx;

runtime() {
  save_context();
  /* Schedule next task */
  cur = cur→next;
  ctx = &cur→ctx;
  load_protection();
  load_context();
}
```

`runtime() {`: $\texttt{cur} \in (\!|\texttt{Task}*|\!),\ \texttt{ctx} \in (\!|\texttt{Context}*|\!)$

`save_context();`: $\texttt{cur} \in (\!|\texttt{Task}*|\!),\ \texttt{ctx} \in (\!|\texttt{Context}*|\!)$

Example parameterized analysis

Initial state:



$$(\!|\texttt{Task}*|\!) = \{0xa2, 0xa7\}$$
$$(\!|\texttt{Context}*|\!) = \{0xa3, 0xa8\}$$

```
Task *cur; Context *ctx;

runtime() {
  save_context();
  /* Schedule next task */
  cur = cur→next;
  ctx = &cur→ctx;
  load_protection();
  load_context();
}
```

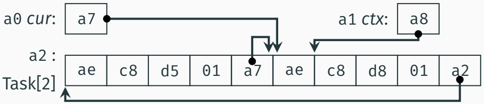
At `runtime() {`: $\texttt{cur} \in (\!|\texttt{Task}*|\!),\ \texttt{ctx} \in (\!|\texttt{Context}*|\!)$

At `save_context();`: $\texttt{cur} \in (\!|\texttt{Task}*|\!),\ \texttt{ctx} \in (\!|\texttt{Context}*|\!)$

At `cur = cur→next;`: $\texttt{cur} \in (\!|\texttt{Task}*|\!),\ \texttt{ctx} \in (\!|\texttt{Context}*|\!)$

Example parameterized analysis

Initial state:



$$(\!|\texttt{Task}*|\!) = \{0xa2, 0xa7\}$$
$$(\!|\texttt{Context}*|\!) = \{0xa3, 0xa8\}$$

```
Task *cur; Context *ctx;

runtime() {
  save_context();
  /* Schedule next task */
  cur = cur→next;
  ctx = &cur→ctx;
  load_protection();
  load_context();
}
```

`runtime() {` — $\texttt{cur} \in (\!|\texttt{Task}*|\!),\ \texttt{ctx} \in (\!|\texttt{Context}*|\!)$

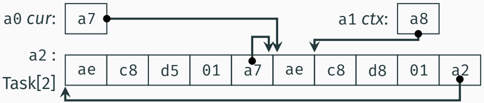
`save_context();` — $\texttt{cur} \in (\!|\texttt{Task}*|\!),\ \texttt{ctx} \in (\!|\texttt{Context}*|\!)$

`cur = cur→next;` — $\texttt{cur} \in (\!|\texttt{Task}*|\!),\ \texttt{ctx} \in (\!|\texttt{Context}*|\!)$

`ctx = &cur→ctx;` — $\texttt{cur} \in (\!|\texttt{Task}*|\!),\ \texttt{ctx} \in (\!|\texttt{Context}*|\!)$

Example parameterized analysis

Initial state:



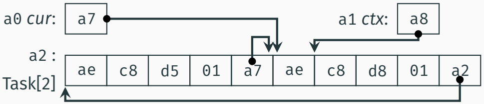
$$(\!|\texttt{Task}*|\!) = \{0xa2, 0xa7\}$$
$$(\!|\texttt{Context}*|\!) = \{0xa3, 0xa8\}$$

```
Task *cur; Context *ctx;
```

`runtime() {`	$\texttt{cur} \in (\!|\texttt{Task}*|\!)$, $\texttt{ctx} \in (\!|\texttt{Context}*|\!)$
`  save_context();`	$\texttt{cur} \in (\!|\texttt{Task}*|\!)$, $\texttt{ctx} \in (\!|\texttt{Context}*|\!)$
`  /* Schedule next task */`	
`  cur = cur→next;`	$\texttt{cur} \in (\!|\texttt{Task}*|\!)$, $\texttt{ctx} \in (\!|\texttt{Context}*|\!)$
`  ctx = &cur→ctx;`	$\texttt{cur} \in (\!|\texttt{Task}*|\!)$, $\texttt{ctx} \in (\!|\texttt{Context}*|\!)$
`  load_protection();`	$\texttt{cur} \in (\!|\texttt{Task}*|\!)$, $\texttt{ctx} \in (\!|\texttt{Context}*|\!)$
`  load_context();`	and kernel is protected
`}`	

Example parameterized analysis

Initial state:



$$\llparenthesis \texttt{Task}* \rrparenthesis = \{0xa2, 0xa7\}$$
$$\llparenthesis \texttt{Context}* \rrparenthesis = \{0xa3, 0xa8\}$$

```
Task *cur; Context *ctx;
```

`runtime() {`	$\texttt{cur} \in \llparenthesis \texttt{Task}* \rrparenthesis$, $\texttt{ctx} \in \llparenthesis \texttt{Context}* \rrparenthesis$
`  save_context();`	$\texttt{cur} \in \llparenthesis \texttt{Task}* \rrparenthesis$, $\texttt{ctx} \in \llparenthesis \texttt{Context}* \rrparenthesis$
`  /* Schedule next task */`	
`  cur = cur→next;`	$\texttt{cur} \in \llparenthesis \texttt{Task}* \rrparenthesis$, $\texttt{ctx} \in \llparenthesis \texttt{Context}* \rrparenthesis$
`  ctx = &cur→ctx;`	$\texttt{cur} \in \llparenthesis \texttt{Task}* \rrparenthesis$, $\texttt{ctx} \in \llparenthesis \texttt{Context}* \rrparenthesis$
`  load_protection();`	$\texttt{cur} \in \llparenthesis \texttt{Task}* \rrparenthesis$, $\texttt{ctx} \in \llparenthesis \texttt{Context}* \rrparenthesis$
`  load_context();`	and kernel is protected
`}`	

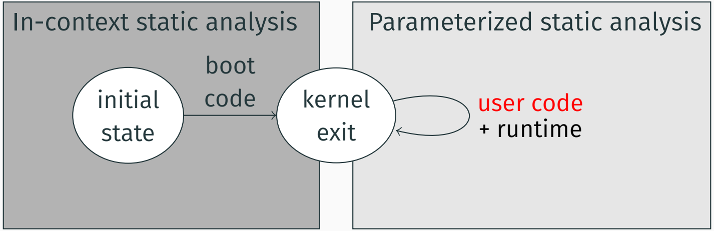
user code$^\sharp$

Differentiated handling of boot and runtime code

- Type-based analysis verifies the **preservation** of the invariant
- But the boot code **establishes** that invariant

Based on this, we

1. Perform a **parameterized** analysis of the **runtime**
2. And an **in-context** analysis of the boot code
3. Check that the state after boot matches the invariant.



Experimental evaluation

Experimental evaluation: Real-life effectiveness

Case study 1: ASTERIOS

- Industrial microkernel used in industrial settings
- Version: port to an **ARM** quad-core
- 329 functions, ~10,000 instructions
- Protection using **page tables**.

2 versions

- BETA version: 1 vulnerability
- V1 version: vulnerability fixed

Specific = 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.

Experimental evaluation: Genericity

Case study 2: 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.

Varying parameters:

- compiler (GCC/Clang), optimization flags
- scheduling algorithm (EDF/FP) dynamic thread creation (on/off)

...

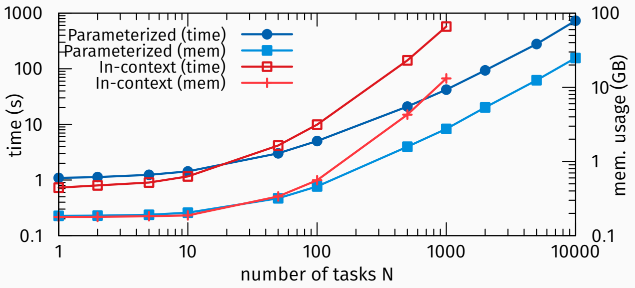
Verification time: from 1.6 s to 73 s.

14 lines of annotations.

Experimental evaluation: Automation and Scalability

We compare

- **fully automated in-context** analysis vs **parameterized** analysis (12 lines of annotations)
- for a simple variant of EducRTOS
- with varying numbers of tasks.



Time and space complexity of **parameterized** analysis is **almost linear**
In-context verification is **quadratic**

Conclusion

BINSEC/CODEX formally verifies embedded kernels (absence of run-time error and absence of privilege escalation)

- from the executable
- with a low annotation burden.

We address existing limitations:

- We allow **parameterized** verification
- We handle **unbounded loops** (necessary for RT scheduling)
- We **infer** the kernel invariants (instead of only checking them)

$\implies$ Key enabler for more automated verification of larger systems.

`https://binsec.github.io/`