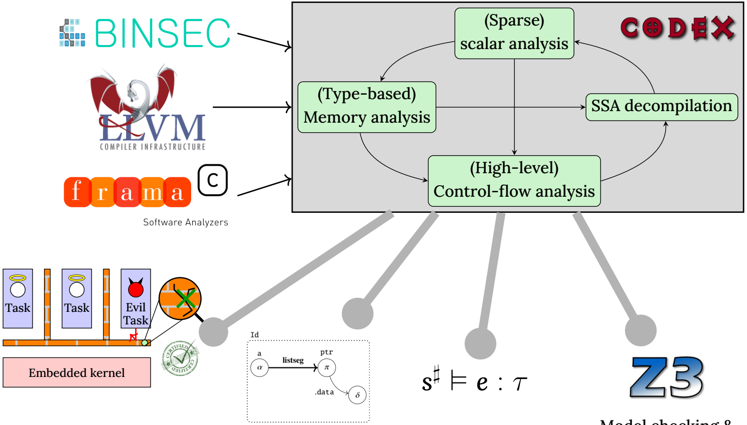
No Crash, No Exploit: Automated Verification of Embedded Kernels

O. Nicole, M. Lemerre, S. Bardin, X. Rival

Coap seminar, 9 novembre 2023

Context: Abstract Interpretation with BINSEC/Codex



(Sparse) scalar analysis

(Type-based) Memory analysis

SSA decompilation

(High-level) Control-flow analysis

$s^\sharp \models e : \tau$

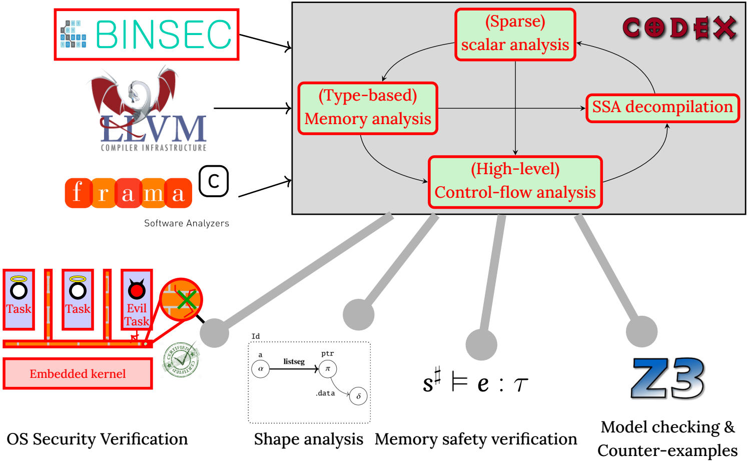
OS Security Verification

Shape analysis

Memory safety verification

Model checking & Counter-examples

Context: Abstract Interpretation with BINSEC/Codex



(Sparse) scalar analysis

(Type-based) Memory analysis

SSA decompilation

(High-level) Control-flow analysis

$s^\sharp \models e : \tau$

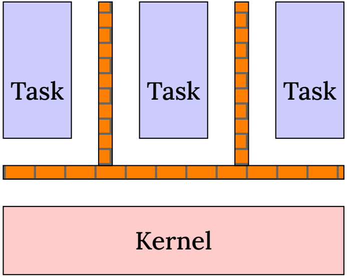
OS Security Verification

Shape analysis

Memory safety verification

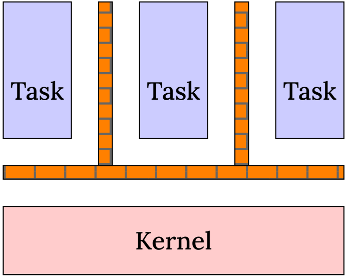
Model checking & Counter-examples

How to protect an OS kernel against its worst defects?



Worst possible bugs for an OS kernel:

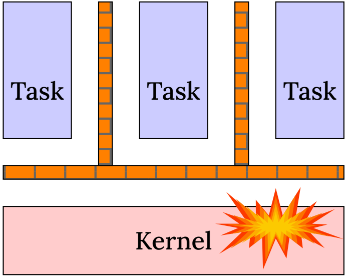
How to protect an OS kernel against its worst defects?



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?

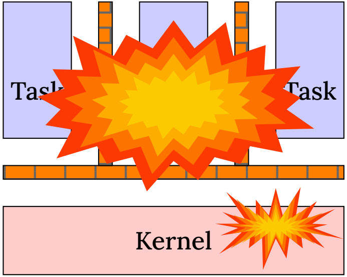


Worst possible bugs for an OS kernel:

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

  The kernel **crashes**

How to protect an OS kernel against its worst defects?

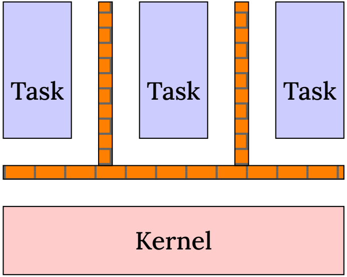


Worst possible bugs for an OS kernel:

- **Runtime errors** Division by zero, illegal memory access…

  The kernel **crashes** $\Longrightarrow$ the whole system crashes

How to protect an OS kernel against its worst defects?

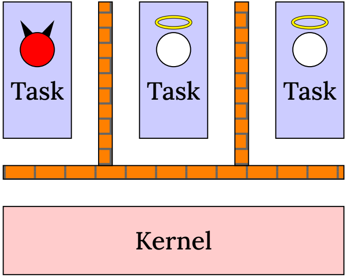


Worst possible bugs for an OS kernel:

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

  The kernel **crashes** ⟹ the whole system crashes

How to protect an OS kernel against its worst defects?



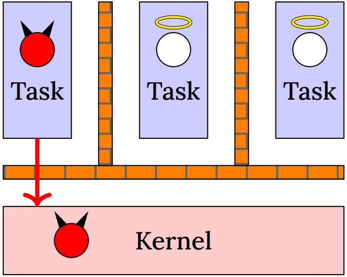
Worst possible bugs for an OS kernel:

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

  The kernel **crashes** $\Longrightarrow$ the whole system crashes

- **Privilege escalation**

How to protect an OS kernel against its worst defects?



Worst possible bugs for an OS kernel:

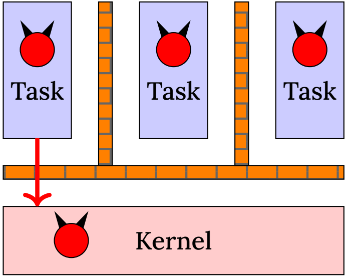
- **Runtime errors** Division by zero, illegal memory access...

  The kernel **crashes** ⟹ 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:

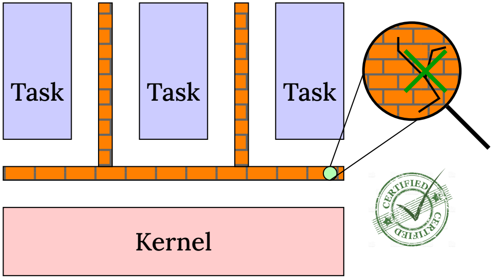
- **Runtime errors** Division by zero, illegal memory access...

  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

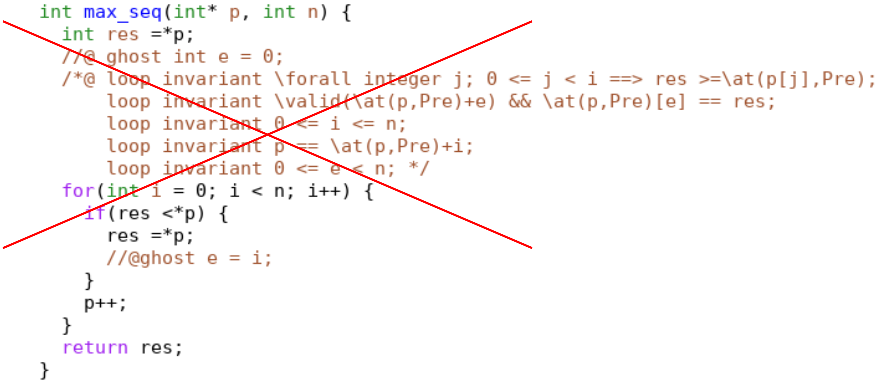
We want a verification of

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

that is:

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

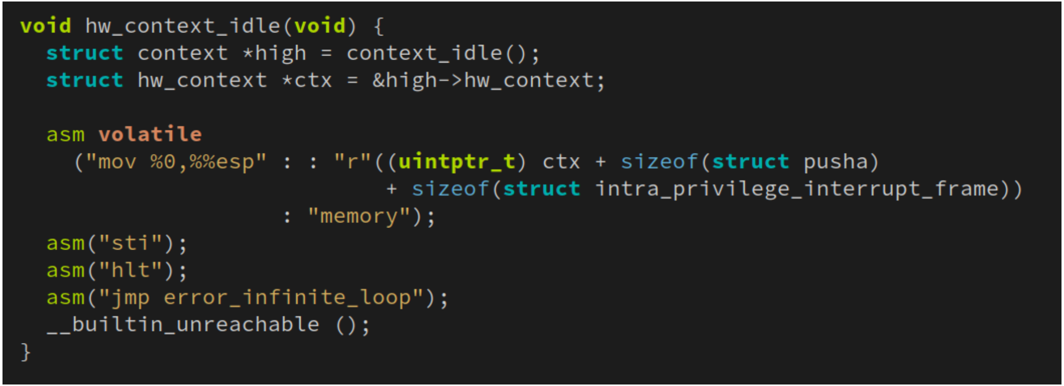
Automated



```
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

$$\forall \text{ tasks}, \left(\text{kernel} \oplus \text{tasks}\right) \models \text{APE, ARTE}$$

- 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)

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 interpretation basics

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

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

$i \in \{100\}$

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;
```

$i \in \{100\}$

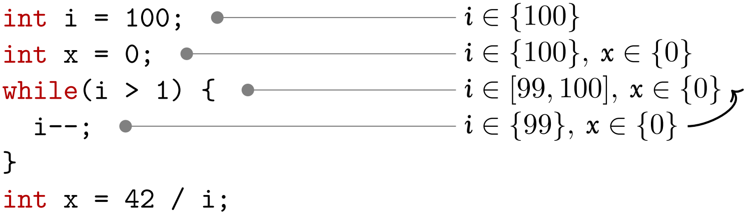
$i \in \{100\},\ x \in \{0\}$

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

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

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;
```

$i \in \{100\}$

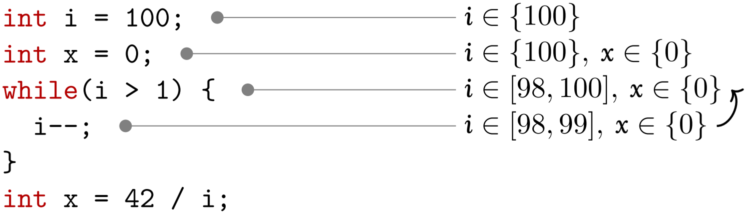
$i \in \{100\}, \; x \in \{0\}$

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

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

Abstract interpretation basics

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



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

$i \in \{100\}$

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

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

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

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;
```

$i \in \{100\}$

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

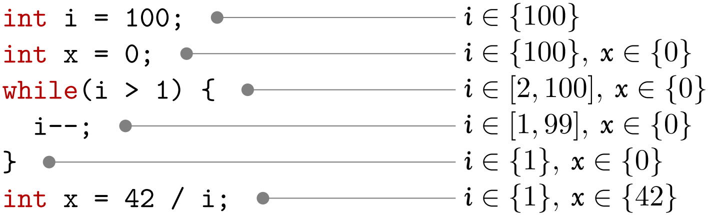
$i \in [2, 100]$, $x \in \{0\}$

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

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

Abstract interpretation basics

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



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

$i \in \{100\}$

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

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

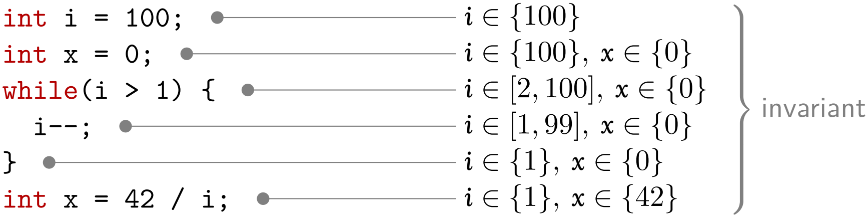
$i \in [1, 99],\ x \in \{0\}$

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

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

Abstract interpretation basics

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



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

$i \in \{100\}$

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

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

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

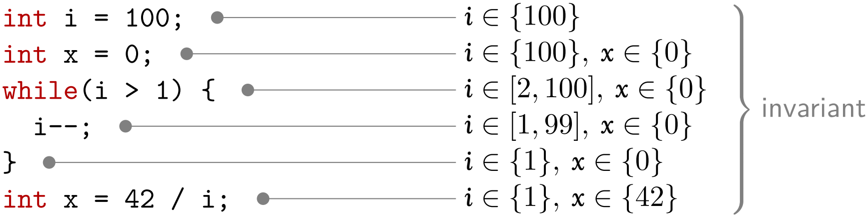
$i \in \{1\}$, $x \in \{0\}$

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

invariant

Abstract interpretation basics

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



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

$i \in \{100\}$

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

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

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

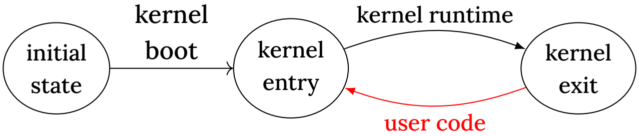
$i \in \{1\},\ x \in \{0\}$

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

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

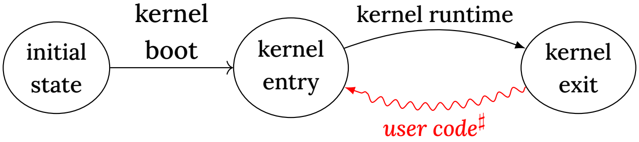
> Absence of run-time errors (**ARTE**) is an implicit property.

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

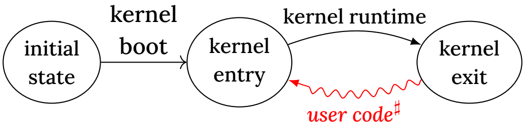
If the system satisfies a non-trivial invariant,
then no privilege escalation is possible on that system.

Proof.

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

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

Example kernel



initial state

kernel boot

kernel entry

kernel runtime

kernel exit

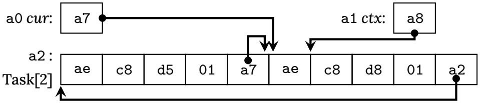
user code$^\sharp$

```
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

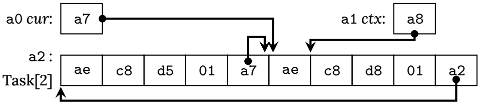
Initial state:



```
Task *cur; Context *ctx;

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

Example in-context analysis

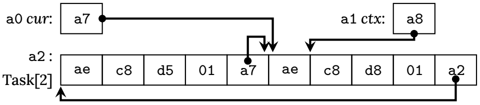
Initial state:



```
Task *cur; Context *ctx;

runtime() {  ●──────────  cur ∈ {0xa7}, ctx ∈ {0xa8}
  save_context();
  /* Schedule next task */
  cur = cur→next;
  ctx = &cur→ctx;
  load_protection();
  load_context();
}
```

Example in-context analysis

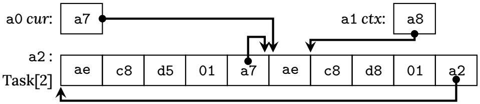
Initial state:



```
Task *cur; Context *ctx;

runtime() {   ●——————————————————  cur ∈ {0xa7}, ctx ∈ {0xa8}
  save_context();   ●————————————  cur ∈ {0xa7}, ctx ∈ {0xa8}
  /* Schedule next task */
  cur = cur→next;
  ctx = &cur→ctx;
  load_protection();
  load_context();
}
```

Example in-context analysis

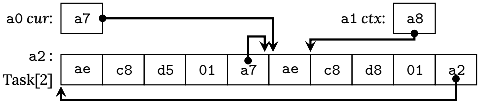
Initial state:



```
Task *cur; Context *ctx;

runtime() {           ●───────────────  cur ∈ {0xa7}, ctx ∈ {0xa8}
  save_context();     ●─────────────  cur ∈ {0xa7}, ctx ∈ {0xa8}
  /* Schedule next task */
  cur = cur→next;     ●─────────────  cur ∈ {0xa2}, ctx ∈ {0xa8}
  ctx = &cur→ctx;
  load_protection();
  load_context();
}
```

Example in-context analysis

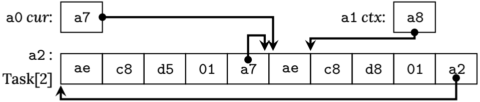
Initial state:



```
Task *cur; Context *ctx;

runtime() {                     cur ∈ {0xa7}, ctx ∈ {0xa8}
  save_context();               cur ∈ {0xa7}, ctx ∈ {0xa8}
  /* Schedule next task */
  cur = cur→next;               cur ∈ {0xa2}, ctx ∈ {0xa8}
  ctx = &cur→ctx;               cur ∈ {0xa2}, ctx ∈ {0xa3}
  load_protection();
  load_context();
}
```

Example in-context analysis

Initial state:

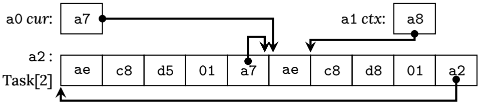


```
Task *cur; Context *ctx;
```

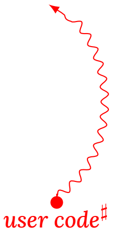
`runtime() {`	cur $\in \{0xa7\}$, ctx $\in \{0xa8\}$
`  save_context();`	cur $\in \{0xa7\}$, ctx $\in \{0xa8\}$
`  /* Schedule next task */`	
`  cur = cur→next;`	cur $\in \{0xa2\}$, ctx $\in \{0xa8\}$
`  ctx = &cur→ctx;`	cur $\in \{0xa2\}$, ctx $\in \{0xa3\}$
`  load_protection();`	cur $\in \{0xa2\}$,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

Example in-context analysis

Initial state:

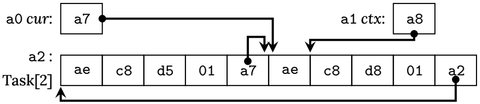


```
Task *cur; Context *ctx;

runtime() {                  cur ∈ {0xa7}, ctx ∈ {0xa8}
  save_context();            cur ∈ {0xa7}, ctx ∈ {0xa8}
  /* Schedule next task */
  cur = cur→next;            cur ∈ {0xa2}, ctx ∈ {0xa8}
  ctx = &cur→ctx;            cur ∈ {0xa2}, ctx ∈ {0xa3}
  load_protection();         cur ∈ {0xa2}, ctx ∈ {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();
}
```

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

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

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

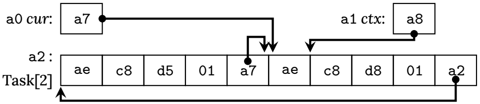
$\texttt{cur} \in \{0xa2\},\ \texttt{ctx} \in \{0xa3\}$

$\texttt{cur} \in \{0xa2\},\ \texttt{ctx} \in \{0xa3\}$
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;` — $\texttt{cur} \in \{0xa2\},\ \texttt{ctx} \in \{0xa8\}$

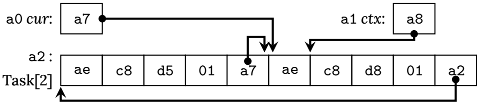
`ctx = &cur→ctx;` — $\texttt{cur} \in \{0xa2\},\ \texttt{ctx} \in \{0xa3\}$

`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;

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

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

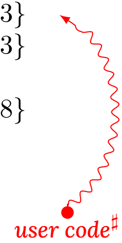
save_context(); — cur $\in \{0xa7, 0xa2\}$, ctx $\in \{0xa8, 0xa3\}$

cur = cur→next; — cur $\in \{0xa2, 0xa7\}$, ctx $\in \{0xa3, 0xa8\}$

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

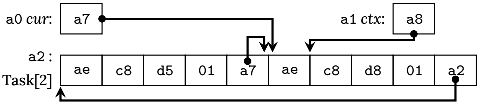
load_protection(); — cur $\in \{0xa2\}$, ctx $\in \{0xa3\}$

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();
}
```

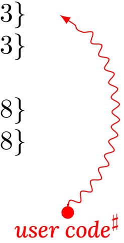
`runtime() {` : `cur` $\in \{0xa7, 0xa2\}$, `ctx` $\in \{0xa8, 0xa3\}$

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

`cur = cur→next;` : `cur` $\in \{0xa2, 0xa7\}$, `ctx` $\in \{0xa3, 0xa8\}$

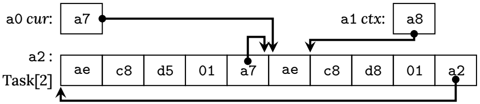
`ctx = &cur→ctx;` : `cur` $\in \{0xa2, 0xa7\}$, `ctx` $\in \{0xa3, 0xa8\}$

`load_protection();` : `cur` $\in \{0xa2\}$, `ctx` $\in \{0xa3\}$
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();
}
```

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

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

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

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

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

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

Shortcomings of in-context analyses

The method is:

- **Not generic:** Cannot analyze kernel independently from the applications
- **Not scalable:** 1000 tasks $\Longrightarrow$ 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.

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

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

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** $\llparenthesis t \rrparenthesis$ as a set of values.

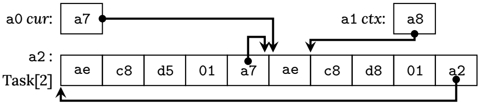
E.g.

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

$$\llparenthesis \texttt{Flags} \rrparenthesis = \{x \mid x \,\&\, \texttt{PRIVILEGED} = 0\}$$

Example parameterized analysis

Initial state:



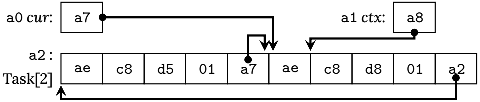
$$(\!|\texttt{Task}*|\!) = \{0xa2, 0xa7\}$$

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

```
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:



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

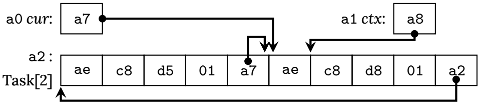
$$(\!|\texttt{Context}*|\!) = \{0xa3, 0xa8\}$$

```
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:



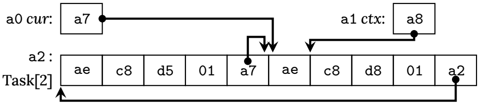
$$\llparenthesis \texttt{Task*} \rrparenthesis = \{0xa2, 0xa7\}$$

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

```
Task *cur; Context *ctx;

runtime() {   ●————————— cur ∈ ⦇Task*⦈, ctx ∈ ⦇Context*⦈
  save_context();   ●———— cur ∈ ⦇Task*⦈, ctx ∈ ⦇Context*⦈
  /* Schedule next task */
  cur = cur→next;
  ctx = &cur→ctx;
  load_protection();
  load_context();
}
```

Example parameterized analysis

Initial state:



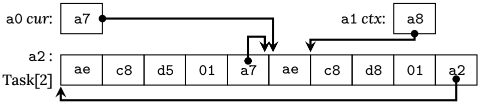
$$(\!|\texttt{Task}*|\!) = \{0xa2, 0xa7\}$$

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

```
Task *cur; Context *ctx;

runtime() {  ●——————————————  cur ∈ ⦇Task∗⦈, ctx ∈ ⦇Context∗⦈
  save_context();  ●————————  cur ∈ ⦇Task∗⦈, ctx ∈ ⦇Context∗⦈
  /* Schedule next task */
  cur = cur→next;  ●————————  cur ∈ ⦇Task∗⦈, ctx ∈ ⦇Context∗⦈
  ctx = &cur→ctx;
  load_protection();
  load_context();
}
```

Example parameterized analysis

Initial state:



a0 *cur*: a7

a1 *ctx*: a8

a2 : Task[2]

ae	c8	d5	01	a7	ae	c8	d8	01	a2

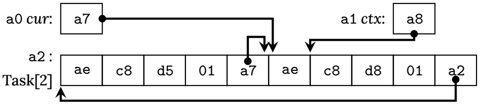
$$(\!|\texttt{Task}*|\!) = \{0xa2, 0xa7\}$$

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

```
Task *cur; Context *ctx;

runtime() {   ●──────────────────────── cur ∈ ⦇Task∗⦈, ctx ∈ ⦇Context∗⦈
  save_context();  ●─────────────────── cur ∈ ⦇Task∗⦈, ctx ∈ ⦇Context∗⦈
  /* Schedule next task */
  cur = cur→next;   ●────────────────── cur ∈ ⦇Task∗⦈, ctx ∈ ⦇Context∗⦈
  ctx = &cur→ctx;   ●────────────────── cur ∈ ⦇Task∗⦈, ctx ∈ ⦇Context∗⦈
  load_protection();
  load_context();
}
```

Example parameterized analysis

Initial state:



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

```
Task *cur; Context *ctx;

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

$\texttt{cur} \in \llparenthesis \texttt{Task}* \rrparenthesis$, $\texttt{ctx} \in \llparenthesis \texttt{Context}* \rrparenthesis$

$\texttt{cur} \in \llparenthesis \texttt{Task}* \rrparenthesis$, $\texttt{ctx} \in \llparenthesis \texttt{Context}* \rrparenthesis$

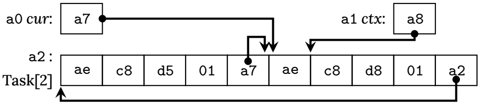
$\texttt{cur} \in \llparenthesis \texttt{Task}* \rrparenthesis$, $\texttt{ctx} \in \llparenthesis \texttt{Context}* \rrparenthesis$

$\texttt{cur} \in \llparenthesis \texttt{Task}* \rrparenthesis$, $\texttt{ctx} \in \llparenthesis \texttt{Context}* \rrparenthesis$

$\texttt{cur} \in \llparenthesis \texttt{Task}* \rrparenthesis$, $\texttt{ctx} \in \llparenthesis \texttt{Context}* \rrparenthesis$
and kernel is protected

Example parameterized analysis

Initial state:



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

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

```
Task *cur; Context *ctx;

runtime() {                  cur ∈ ⦇Task∗⦈, ctx ∈ ⦇Context∗⦈
  save_context();            cur ∈ ⦇Task∗⦈, ctx ∈ ⦇Context∗⦈
  /* Schedule next task */
  cur = cur→next;            cur ∈ ⦇Task∗⦈, ctx ∈ ⦇Context∗⦈
  ctx = &cur→ctx;            cur ∈ ⦇Task∗⦈, ctx ∈ ⦇Context∗⦈
  load_protection();         cur ∈ ⦇Task∗⦈, ctx ∈ ⦇Context∗⦈
  load_context();            and kernel is protected
}
```

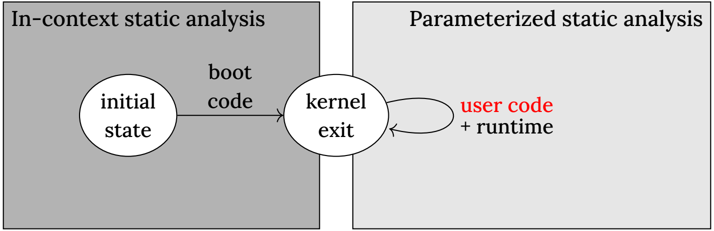
user code$^\sharp$

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: 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:

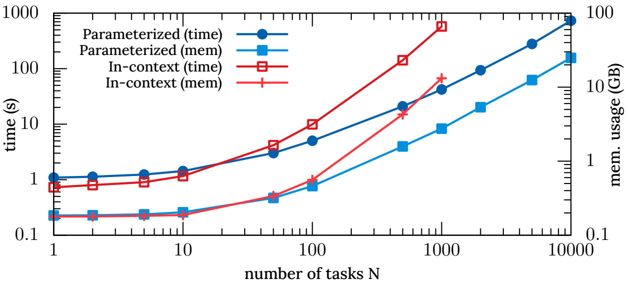
- compiler (GCC/Clang), optimization flags
- scheduling algorithm (EDF/FP) dynamic thread creation (on/off) ...

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**

First 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)

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

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

Followup: Type-based abstract interpretation

[RTAS 2021 best paper, VMCAI 2022, Thèses O. Nicole & J. Simonnet, EMASS ANR project]

Verifying type safety using abstract interpretation

- + Encompasses memory safety
  - Still 70% of the vulnerabilities in the wild
  - An alternative to rewriting in Rust or Microsoft Checked C
- + Cheap analyses operation for quite strong memory invariants
  - Allows precise handling of code performing dynamic memory allocation
- + Familar abstraction easily tunable and understandable by the user
  - Recovering types is a key step in reverse engineering.
- + Type is the abstraction for modular development → modular analysis
- + Gracefully handles analyses imprecision
  - Provides the necessary safeguards to complete abstract interpretation of machine code .
- + A useful base to combine with other memory abstractions
  - Array abstractions for array types, shape abstractions for recursive types, variant abstraction for unions, etc.
- Key successes:
  - 0 alarms when verifying AsteriOS (low-level OS code, variable memory)
  - Good results on challenging benchmarks (data structure libraries, Emacs...)
  - Makes static analysis of machine code doable on a variety of benchmarks