Automated Verification of Systems Code using Type-Based Memory Abstractions

Olivier Nicole

19 April, 2022

École normale supérieure – PSL, CEA List

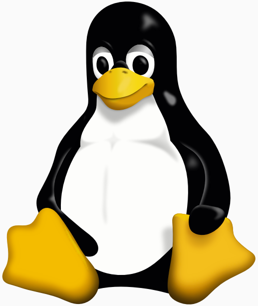
Alan Schmitt	Inria	Reviewer
Mihaela Sighireanu	ENS Paris-Saclay	Reviewer
Timothy Bourke	Inria	Examiner
Jean-Christophe Filliâtre	CNRS	Examiner
Julia Lawall	Inria	Examiner
Xavier Rival	ENS, Inria	Supervisor
Matthieu Lemerre	CEA List	Co-advisor

Software correctness in critical systems

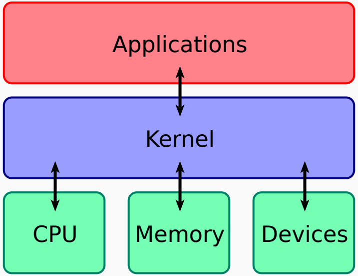
Software contains bugs and vulnerabilities

New bugs are discovered regularly

CVE-2021-22555 in **Linux**

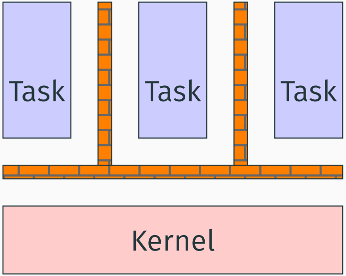


Kernels are a critical component of software systems



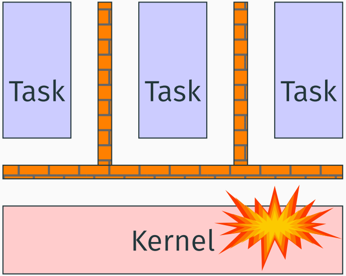
- The operating system kernel:
  - organizes the sharing of hardware resources between applications
  - prevents applications from disturbing the functioning of other applications

Kernels are a critical component of software systems



Two bugs particularly threaten safety and security:

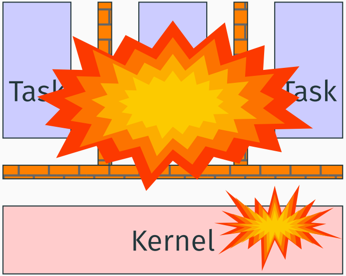
Kernels are a critical component of software systems



Two bugs particularly threaten safety and security:

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

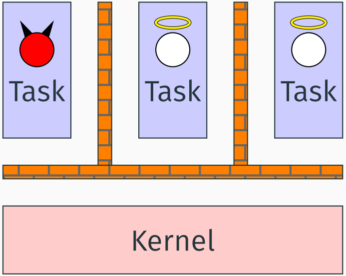
Kernels are a critical component of software systems



Two bugs particularly threaten safety and security:

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

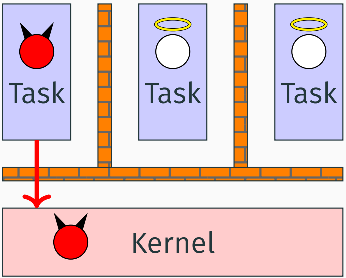
Kernels are a critical component of software systems



Two bugs particularly threaten safety and security:

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

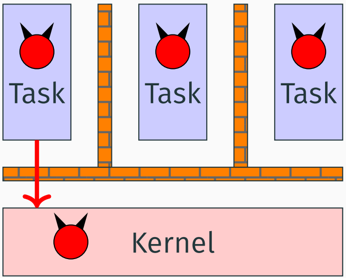
Kernels are a critical component of software systems



Two bugs particularly threaten safety and security:

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

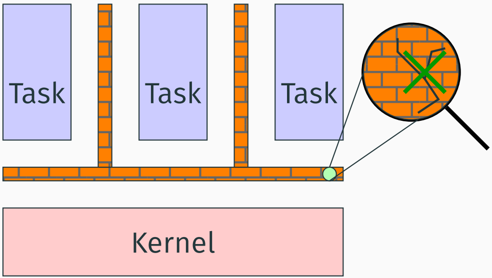
Kernels are a critical component of software systems



Two bugs particularly threaten safety and security:

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

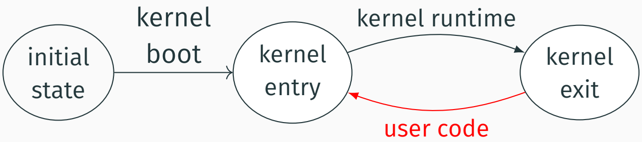
Kernels are a critical component of software systems



Two bugs particularly threaten safety and security:

- Runtime errors   Division by zero, illegal memory access...
- Privilege escalation
- **Goal:** Verify their absence using formal methods

Kernel execution: the system loop



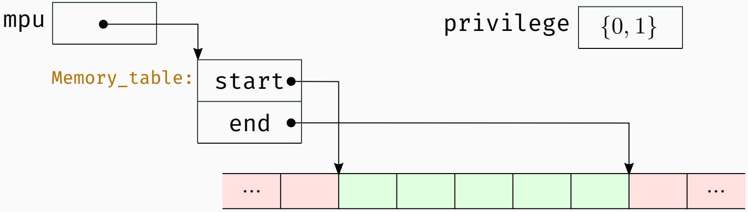
A system with a kernel executes alternatively **user code** and **kernel runtime**.

It may pause a task and run another one

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

Memory protection

- The Memory Protection Unit (MPU) controls access to memory
- Restricted to an interval of memory addresses
- Interval described in a memory protection table



- mpu can be changed only if privilege $= 1$

Kernel verification requires memory invariants

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

```c
Task *cur;

void handle_timer() {
  /* Save task context */
  cur→ctx = interrupted_ctx;
  /* Schedule next task */
  cur = cur→next;
  /* Load new memory protection */
  mpu = cur→mem_table;
  /* Give control to new task */
  switch_context(&cur→ctx);
}
```

Kernel verification requires memory invariants

```
Task *cur;

void handle_timer() {
  /* Save task context */
  cur→ctx = interrupted_ctx;
  /* Schedule next task */
  cur = cur→next;
  /* Load new memory protection */
  mpu = cur→mem_table;
  /* Give control to new task */
  switch_context(&cur→ctx);
}
```

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

Invariants needed

- cur does not point to a memory table

Kernel verification requires memory invariants

```
Task *cur;

void handle_timer() {
  /* Save task context */
  cur->ctx = interrupted_ctx;
  /* Schedule next task */
  cur = cur->next;
  /* Load new memory protection */
  mpu = cur->mem_table;
  /* Give control to new task */
  switch_context(&cur->ctx);
}
```

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

Invariants needed

- `cur` does not point to a memory table
- at the end, `mpu->start` > `kernel_last_addr`

Kernel verification requires memory invariants

```
Task *cur;

void handle_timer() {
  /* Save task context */
  cur→ctx = interrupted_ctx;
  /* Schedule next task */
  cur = cur→next;
  /* Load new memory protection */
  mpu = cur→mem_table;
  /* Give control to new task */
  switch_context(&cur→ctx);
}
```

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

Invariants needed

- cur does not point to a memory table
- at the end,
  $\texttt{mpu}{\rightarrow}\texttt{start} > \texttt{kernel\_last\_addr}$
  $\texttt{cur}{\rightarrow}\texttt{ctx.privilege} = 0$

Invariant proof methods for kernel verification

Interactive proof

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

Deductive verification

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

Prove strong properties, but require a lot of work from experts

Invariant proof methods for kernel verification

Interactive proof

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

Deductive verification

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

Prove strong properties, but require a lot of work from experts

"Push-button" verification

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

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

Invariant proof methods for kernel verification

Interactive proof

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

Deductive verification

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

Prove strong properties, but require a lot of work from experts

---

"Push-button" verification

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

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

Sound static analysis by abstract interpretation

- ASTERIOS

- Applied to the kernel executable
- Infers invariants
- Handles unbounded loops
- Handles parameterized verification
- Low annotation burden (e.g. 58 lines)

Sound static analysis by abstract interpretation

Illustration: Verifying numerical assertions

```
int x,y,z,r;
x = random(1, 4);
y = random(1, 4);

z = x - y;
r = z * z;

assert (r <= 9);
assert (r >= 0);
```

Sound static analysis by abstract interpretation

Illustration: Verifying numerical assertions

```
int x,y,z,r;
x = random(1, 4);
y = random(1, 4);

z = x - y;
r = z * z;

assert (r <= 9);
assert (r >= 0);
```

$x \in [1, 4]$

Illustration: Verifying numerical assertions

```
int x,y,z,r;
x = random(1, 4);
y = random(1, 4);

z = x - y;
r = z * z;

assert (r <= 9);
assert (r >= 0);
```

$x \in [1, 4]$

$y \in [1, 4]$

Sound static analysis by abstract interpretation

Illustration: Verifying numerical assertions

```
int x,y,z,r;
x = random(1, 4);
y = random(1, 4);

z = x - y;
r = z * z;

assert (r <= 9);
assert (r >= 0);
```

$x \in [1, 4]$

$y \in [1, 4]$

$z \in [-3, 3]$

Sound static analysis by abstract interpretation

Illustration: Verifying numerical assertions

```
int x,y,z,r;
x = random(1, 4);
y = random(1, 4);

z = x - y;
r = z * z;

assert (r <= 9);
assert (r >= 0);
```

$x \in [1, 4]$

$y \in [1, 4]$

$z \in [-3, 3]$

$r \in [-9, 9]$

Sound static analysis by abstract interpretation

Illustration: Verifying numerical assertions

```
int x,y,z,r;
x = random(1, 4);
y = random(1, 4);

z = x - y;
r = z * z;

assert (r <= 9);
assert (r >= 0);
```

$x \in [1, 4]$

$y \in [1, 4]$

$z \in [-3, 3]$

$r \in [-9, 9]$

True

Sound static analysis by abstract interpretation

Illustration: Verifying numerical assertions

```
int x,y,z,r;
x = random(1, 4);
y = random(1, 4);

z = x - y;
r = z * z;

assert (r <= 9);
assert (r >= 0);
```

$x \in [1,4]$

$y \in [1,4]$

$z \in [-3,3]$

$r \in [-9,9]$

True

Maybe

Invariants inferred by pointer analyses

- Infer points-to or aliasing relations between program variables

$$x \mapsto \{\&y, \&z\} \qquad \text{“x can pointer to either y or z”}$$

- Very efficient
- Difficult to apply to machine code
- Do not consider the structure of data nor values in memory

Invariants inferred by pointer analyses

```
Task *cur;

void handle_timer() {
  /* Save task context */
  cur->ctx = interrupted_ctx;
  /* Schedule next task */
  cur = cur->next;
  /* Load new memory protection */
  mpu = cur->mem_table;
  /* Give control to new task */
  switch_context(&cur->ctx);
}
```

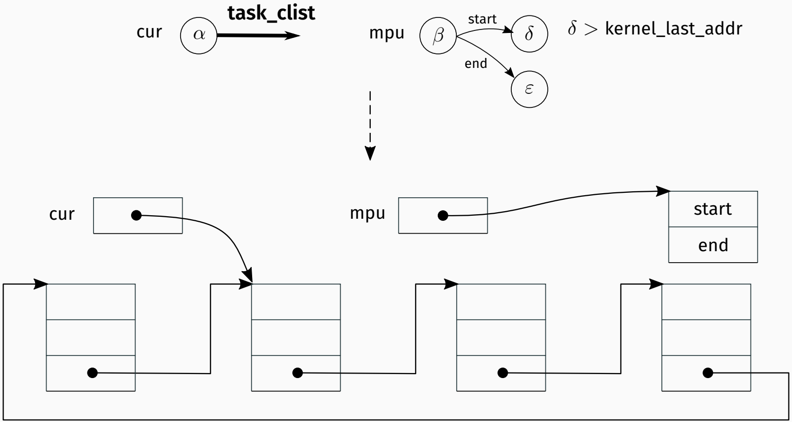
$\mathtt{cur} \mapsto ?$

$\mathtt{mpu} \mapsto ?$

Shape analyses

Shape analyses are good candidates...

- Verify strong properties on memory



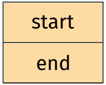
... but difficult to apply

- Difficult to apply to low-level code
- Requires case disjunctions which can be costly
- Fragile in presence of precision loss

Type-based memory invariants

Physical types describe the layout of values in memory

$$\texttt{type}\ \texttt{Memory\_table} = \{x : \texttt{int} \mid x > \texttt{kernel\_last\_addr}\} \times \texttt{int}$$



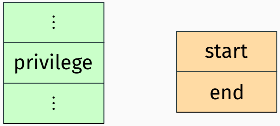
- Product: contiguous values in memory
- Refinement types: numerical constraints on values

Type-based memory invariants

Physical types describe the layout of values in memory

$$\texttt{type Memory\_table} = \{x : \texttt{int} \mid x > \texttt{kernel\_last\_addr}\} \times \texttt{int}$$

$$\texttt{type Context} = \cdots \times \{\mathit{privilege} : \texttt{int} \mid \mathit{privilege} = 0\} \times \cdots$$



- Product: contiguous values in memory
- Refinement types: numerical constraints on values

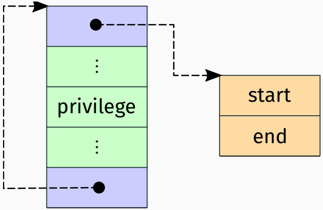
Type-based memory invariants

Physical types describe the layout of values in memory

$\texttt{type Memory\_table} = \{x : \texttt{int} \mid x > \texttt{kernel\_last\_addr}\} \times \texttt{int}$

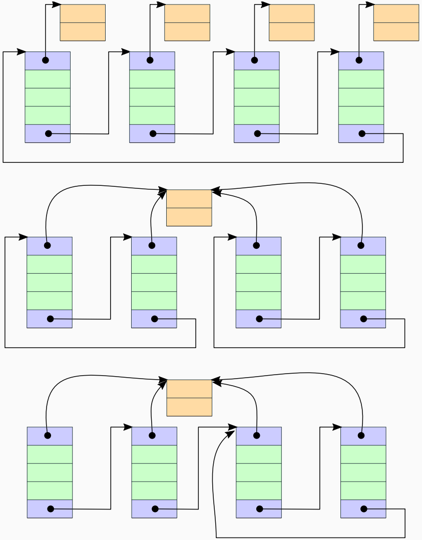
$\texttt{type Context} = \cdots \times \{\mathit{privilege} : \texttt{int} \mid \mathit{privilege} = 0\} \times \cdots$

$\texttt{type Task} = \texttt{Memory\_table}.(0)* \times \texttt{Context} \times \{x : \texttt{Task}.(0)* \mid x \neq 0\}$



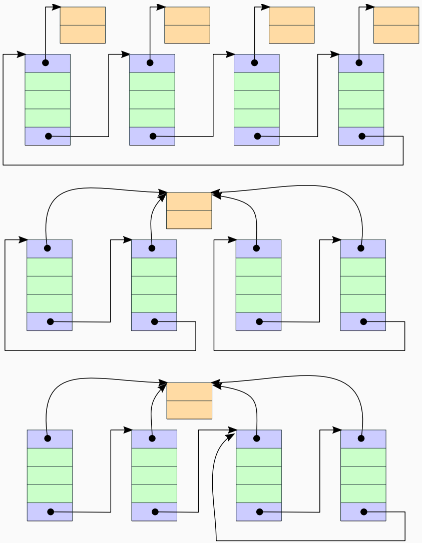
- Product: contiguous values in memory
- Refinement types: numerical constraints on values
- Pointer types with offset

Separation



- Tasks not necessarily separated

Separation



- Tasks not necessarily separated
- Task and Context may alias because Task contains a Context
- But Task and Memory_table never alias:

If α : Task.(0)$*$ and β : Memory_table.(0)$*$, then $\alpha \neq \beta$.

Type-based memory invariants

$\texttt{type Memory\_table} = \{x : \texttt{int} \mid x > \texttt{kernel\_last\_addr}\} \times \texttt{int}$
$\texttt{type Context} = \cdots \times \{\mathit{privilege} : \texttt{int} \mid \mathit{privilege} = 0\} \times \cdots$
$\texttt{type Task} = \texttt{Memory\_table}.(0)* \times \texttt{Context} \times \{x : \texttt{Task}.(0)* \mid x \neq 0\}$

```
Task *cur;

void handle_timer() {
  /* Save task context */
  cur→ctx = interrupted_ctx;
  /* Schedule next task */
  cur = cur→next;
  /* Load new memory protection */
  mpu = cur→mem_table;
  /* Give control to new task */
  switch_context(&cur→ctx);
}
```

Type-based memory invariants

$$\texttt{type Memory\_table} = \{x : \texttt{int} \mid x > \texttt{kernel\_last\_addr}\} \times \texttt{int}$$
$$\texttt{type Context} = \cdots \times \{\mathit{privilege} : \texttt{int} \mid \mathit{privilege} = 0\} \times \cdots$$
$$\texttt{type Task} = \texttt{Memory\_table}.(0)* \times \texttt{Context} \times \{x : \texttt{Task}.(0)* \mid x \neq 0\}$$

```
Task *cur;

void handle_timer() {
  /* Save task context */
  cur→ctx = interrupted_ctx;
  /* Schedule next task */
  cur = cur→next;
  /* Load new memory protection */
  mpu = cur→mem_table;
  /* Give control to new task */
  switch_context(&cur→ctx);
}
```

$\alpha \neq 0$

cur $\boxed{\alpha : \texttt{Task}.(0)*}$ mpu $\boxed{\beta : \texttt{Memory\_table}.(0)*}$

Type-based memory invariants

$\texttt{type Memory\_table} = \{x : \texttt{int} \mid x > \texttt{kernel\_last\_addr}\} \times \texttt{int}$
$\texttt{type Context} = \cdots \times \{\textit{privilege} : \texttt{int} \mid \textit{privilege} = 0\} \times \cdots$
$\texttt{type Task} = \texttt{Memory\_table}.(0)* \times \texttt{Context} \times \{x : \texttt{Task}.(0)* \mid x \neq 0\}$

```
Task *cur;

void handle_timer() {
  /* Save task context */
  cur→ctx = interrupted_ctx;
  /* Schedule next task */
  cur = cur→next;
  /* Load new memory protection */
  mpu = cur→mem_table;
  /* Give control to new task */
  switch_context(&cur→ctx);
}
```

$\alpha \neq 0$

cur $\boxed{\alpha : \texttt{Task}.(0)*}$ mpu $\boxed{\beta : \texttt{Memory\_table}.(0)*}$

cur $\boxed{\alpha : \texttt{Task}.(0)*}$ mpu $\boxed{\beta : \texttt{Memory\_table}.(0)*}$

Type-based memory invariants

$\texttt{type Memory\_table} = \{x : \texttt{int} \mid x > \texttt{kernel\_last\_addr}\} \times \texttt{int}$
$\texttt{type Context} = \cdots \times \{\mathit{privilege} : \texttt{int} \mid \mathit{privilege} = 0\} \times \cdots$
$\texttt{type Task} = \texttt{Memory\_table}.(0)* \times \texttt{Context} \times \{x : \texttt{Task}.(0)* \mid x \neq 0\}$

```
Task *cur;

void handle_timer() {
  /* Save task context */
  cur→ctx = interrupted_ctx;
  /* Schedule next task */
  cur = cur→next;
  /* Load new memory protection */
  mpu = cur→mem_table;
  /* Give control to new task */
  switch_context(&cur→ctx);
}
```

$\alpha \neq 0 \qquad \delta \neq 0$

cur $\boxed{\alpha : \texttt{Task}.(0)*}$ mpu $\boxed{\beta : \texttt{Memory\_table}.(0)*}$

cur $\boxed{\alpha : \texttt{Task}.(0)*}$ mpu $\boxed{\beta : \texttt{Memory\_table}.(0)*}$

cur $\boxed{\delta : \texttt{Task}.(0)*}$ mpu $\boxed{\beta : \texttt{Memory\_table}.(0)*}$

Type-based memory invariants

$\texttt{type Memory\_table} = \{x : \texttt{int} \mid x > \texttt{kernel\_last\_addr}\} \times \texttt{int}$
$\texttt{type Context} = \cdots \times \{\mathit{privilege} : \texttt{int} \mid \mathit{privilege} = 0\} \times \cdots$
$\texttt{type Task} = \texttt{Memory\_table}.(0)* \times \texttt{Context} \times \{x : \texttt{Task}.(0)* \mid x \neq 0\}$

$\alpha \neq 0 \qquad \delta \neq 0$

```
Task *cur;

void handle_timer() {
  /* Save task context */
  cur→ctx = interrupted_ctx;
  /* Schedule next task */
  cur = cur→next;
  /* Load new memory protection */
  mpu = cur→mem_table;
  /* Give control to new task */
  switch_context(&cur→ctx);
}
```

cur	$\boxed{\alpha : \texttt{Task}.(0)*}$	mpu	$\boxed{\beta : \texttt{Memory\_table}.(0)*}$
cur	$\boxed{\alpha : \texttt{Task}.(0)*}$	mpu	$\boxed{\beta : \texttt{Memory\_table}.(0)*}$
cur	$\boxed{\delta : \texttt{Task}.(0)*}$	mpu	$\boxed{\beta : \texttt{Memory\_table}.(0)*}$
cur	$\boxed{\delta : \texttt{Task}.(0)*}$	mpu	$\boxed{\varepsilon : \texttt{Memory\_table}.(0)*}$

Type-based memory invariants

$$\texttt{type Memory\_table} = \{x : \texttt{int} \mid x > \texttt{kernel\_last\_addr}\} \times \texttt{int}$$
$$\texttt{type Context} = \cdots \times \{privilege : \texttt{int} \mid privilege = 0\} \times \cdots$$
$$\texttt{type Task} = \texttt{Memory\_table}.(0)* \times \texttt{Context} \times \{x : \texttt{Task}.(0)* \mid x \neq 0\}$$

```
Task *cur;

void handle_timer() {
  /* Save task context */
  cur→ctx = interrupted_ctx;
  /* Schedule next task */
  cur = cur→next;
  /* Load new memory protection */
  mpu = cur→mem_table;
  /* Give control to new task */
  switch_context(&cur→ctx);
}
```

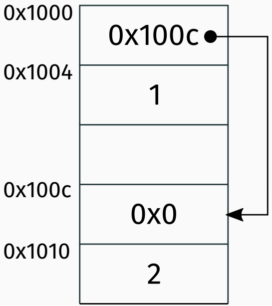
$\alpha \neq 0 \qquad \delta \neq 0$

`void handle_timer() {`	cur $\boxed{\alpha : \texttt{Task}.(0)*}$	mpu $\boxed{\beta : \texttt{Memory\_table}.(0)*}$
`cur→ctx = interrupted_ctx;`	cur $\boxed{\alpha : \texttt{Task}.(0)*}$	mpu $\boxed{\beta : \texttt{Memory\_table}.(0)*}$
`cur = cur→next;`	cur $\boxed{\delta : \texttt{Task}.(0)*}$	mpu $\boxed{\beta : \texttt{Memory\_table}.(0)*}$
`mpu = cur→mem_table;`	cur $\boxed{\delta : \texttt{Task}.(0)*}$	mpu $\boxed{\varepsilon : \texttt{Memory\_table}.(0)*}$
`switch_context(&cur→ctx);`	cur $\boxed{\delta : \texttt{Task}.(0)*}$	mpu $\boxed{\varepsilon : \texttt{Memory\_table}.(0)*}$

Contributions and outline

- A type-based memory abstraction;
    - Structural invariants on memory encoded as type safety
    - Precision improvements using points-to predicates
- Two low-level type-based program analyses
    - For C and machine code
- A method to verify absence of run-time errors (ARTE) and absence of privilege escalation (APE) on embedded kernels
    - Applied to an unmodified industrial kernel

Tagging memory with types

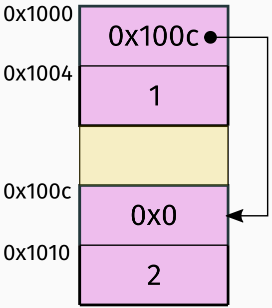


Tagging memory with types

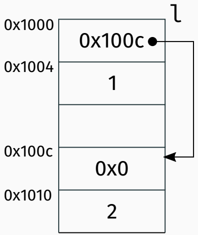
```
typedef int data;
struct list {
  list *next;
  data d;
};
```

$$\text{type } \texttt{list} = \texttt{list}.(0)* \times \texttt{data}$$

$$\text{type } \texttt{data} = \texttt{word}_4$$



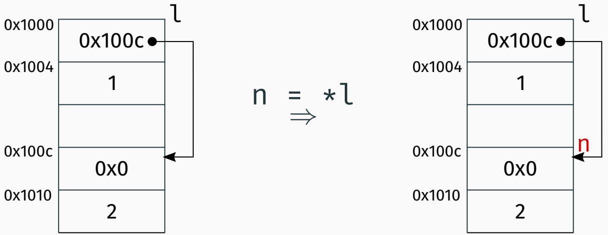
Type-based analysis: Overview



$$\left[\begin{array}{l} \mathtt{l} \mapsto \mathtt{0x100c} \\ \mathtt{n} \mapsto \mathtt{0xcafe} \end{array}\right]$$

Start from the untyped semantics...

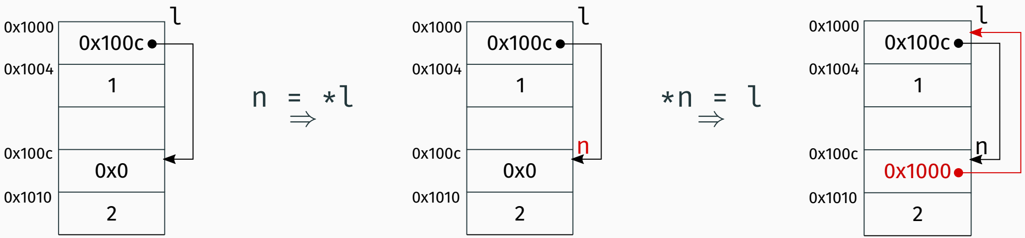
Type-based analysis: Overview



$$\begin{bmatrix} \texttt{l} \mapsto \texttt{0x100c} \\ \texttt{n} \mapsto \texttt{0xcafe} \end{bmatrix} \qquad \begin{bmatrix} \texttt{l} \mapsto \texttt{0x100c} \\ \texttt{n} \mapsto \color{red}{\texttt{0x100c}} \end{bmatrix}$$

Start from the untyped semantics…

Type-based analysis: Overview

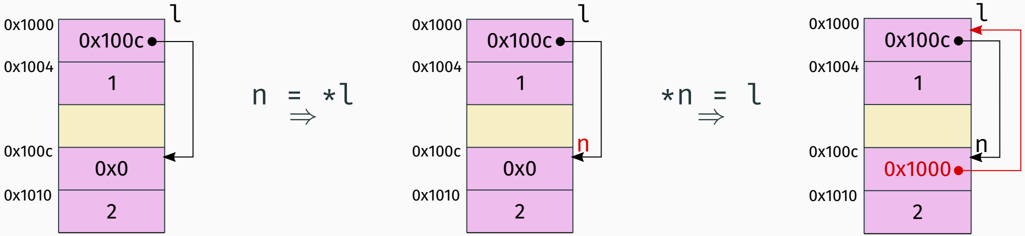


$$\begin{bmatrix} l \mapsto \texttt{0x100c} \\ n \mapsto \texttt{0xcafe} \end{bmatrix} \qquad \begin{bmatrix} l \mapsto \texttt{0x100c} \\ n \mapsto \texttt{0x100c} \end{bmatrix} \qquad \begin{bmatrix} l \mapsto \texttt{0x100c} \\ n \mapsto \texttt{0x100c} \end{bmatrix}$$

Start from the untyped semantics...

Type-based analysis: Overview

$$\begin{bmatrix} \mathtt{l} \mapsto \mathtt{list}.(0)* \\ \mathtt{n} \mapsto \mathtt{word}_4 \end{bmatrix} \qquad \begin{bmatrix} \mathtt{l} \mapsto \mathtt{list}.(0)* \\ \mathtt{n} \mapsto \mathtt{list}.(0)* \end{bmatrix} \qquad \begin{bmatrix} \mathtt{l} \mapsto \mathtt{list}.(0)* \\ \mathtt{n} \mapsto \mathtt{list}.(0)* \end{bmatrix}$$



$$\begin{bmatrix} \mathtt{l} \mapsto \mathtt{0x100c} \\ \mathtt{n} \mapsto \mathtt{0xcafe} \end{bmatrix} \qquad \begin{bmatrix} \mathtt{l} \mapsto \mathtt{0x100c} \\ \mathtt{n} \mapsto \mathtt{0x100c} \end{bmatrix} \qquad \begin{bmatrix} \mathtt{l} \mapsto \mathtt{0x100c} \\ \mathtt{n} \mapsto \mathtt{0x100c} \end{bmatrix}$$

Add types to the heap and variables...

Type-based analysis: Overview

$$\begin{bmatrix} \texttt{l} \mapsto \texttt{list}.(0)* \\ \texttt{n} \mapsto \texttt{word}_4 \end{bmatrix} \xRightarrow{\texttt{n = *l}} \begin{bmatrix} \texttt{l} \mapsto \texttt{list}.(0)* \\ \texttt{n} \mapsto \texttt{list}.(0)* \end{bmatrix} \xRightarrow{\texttt{*n = l}} \begin{bmatrix} \texttt{l} \mapsto \texttt{list}.(0)* \\ \texttt{n} \mapsto \texttt{list}.(0)* \end{bmatrix}$$

$$\gamma(\mathsf{Env}) = \{s \mid s \text{ is well typed with environment Env}\}$$

Keep only variable types; memory is abstracted by type invariants.

Combination with numerical predicates

- Necessary for:
  - Non-nullness of pointers
  - Array indices and pointer arithmetic
  - Use and verify refinement predicates

Reduced product with a numerical abstract domain

`l` $\boxed{\lambda : \texttt{list}.(0)*}$ $\quad \lambda \neq 0$

`n` $\boxed{\eta : \texttt{word}_4}$ $\quad \eta \leq 10$

Example analysis

$$\mathtt{type\ data} = \mathtt{word}_4$$
$$\mathtt{type\ list} = \mathtt{list}.(0)* \times \mathtt{data}$$

```
if (l != 0) {
  n = *l;
}
```

l $\boxed{\lambda : \mathtt{list}.(0)*}$

n $\boxed{\eta : \mathtt{word}_4}$

Example analysis

$\texttt{type data} = \texttt{word}_4$
$\texttt{type list} =$
$\quad \texttt{list}.(0)* \times \texttt{data}$

```
if (l != 0) {
  n = *l;
}
```

$\texttt{l}\ \boxed{\lambda : \texttt{list}.(0)*}$

$\texttt{n}\ \boxed{\eta : \texttt{word}_4}$

$\Downarrow \quad \texttt{assume}(\texttt{l} \mathrel{!=} 0)$

$\texttt{l}\ \boxed{\lambda : \texttt{list}.(0)*}$

$\texttt{n}\ \boxed{\eta : \texttt{word}_4}$

$\lambda \neq 0$

Example analysis

$\text{type } \mathtt{data} = \mathtt{word}_4$
$\text{type } \mathtt{list} =$
$\quad \mathtt{list}.(0)* \times \mathtt{data}$

```
if (l != 0) {
  n = *l;
}
```

$$\begin{array}{ll} \mathtt{l} & \boxed{\lambda : \mathtt{list}.(0)*} \\ \mathtt{n} & \boxed{\eta : \mathtt{word}_4} \end{array}$$

$$\Downarrow \qquad \text{assume}(\mathtt{l} \mathrel{!=} 0)$$

$$\begin{array}{ll} \mathtt{l} & \boxed{\lambda : \mathtt{list}.(0)*} \\ \mathtt{n} & \boxed{\eta : \mathtt{word}_4} \end{array} \quad \lambda \neq 0$$

$$\Downarrow \qquad \mathtt{n} = *\mathtt{l} \ \checkmark$$

$$\begin{array}{ll} \mathtt{l} & \boxed{\lambda : \mathtt{list}.(0)*} \\ \mathtt{n} & \boxed{\eta : \mathtt{list}.(0)*} \end{array} \quad \lambda \neq 0$$

Soundness

$$\begin{tikzcd}
\mathbb{S}^\sharp \arrow[r, "{[\![p]\!]^\sharp_S}"] \arrow[d, "\gamma"'] & \mathbb{S}^\sharp \arrow[d, "\gamma"] \\
\mathcal{P}(\mathbb{S}_t) \arrow[d, "\text{erase_types}"'] & \mathcal{P}(\mathbb{S}_t) \arrow[d, "\text{erase_types}"] \\
\mathcal{P}(\mathbb{S}) \arrow[r, "{[\![p]\!]}"'] & \mathcal{P}(\mathbb{S})
\end{tikzcd}$$

Theorem (Soundness of the abstract semantics)

For all abstract states $s^\sharp \in \mathbb{S}^\sharp$ and programs p:

$$([\![p]\!] \circ \text{erase\_types} \circ \gamma)(s^\sharp) \subseteq (\text{erase\_types} \circ \gamma \circ [\![p]\!]^\sharp_S)(s^\sharp)$$

A storeless abstraction is often not enough

$\mathtt{l}$ $\boxed{\lambda : \mathtt{list}.(0)*}$

$\mathtt{n}$ $\boxed{\eta : \mathtt{word}_4}$

$\lambda \neq 0$

type $\mathtt{data} = \mathtt{word}_4$
type $\mathtt{list} =$
  $\mathtt{list}.(0)* \times \mathtt{data}$

```
if (*l != 0) {
  n = **l;
}
```

A storeless abstraction is often not enough

$$\begin{array}{ll} \mathtt{l} & \boxed{\lambda : \mathtt{list}.(0)*} \\ \mathtt{n} & \boxed{\eta : \mathtt{word}_4} \end{array} \quad \lambda \neq 0$$

$$\Downarrow \qquad \text{assume}(*\mathtt{l} \;!= 0) \;\checkmark$$

$$\begin{array}{ll} \mathtt{l} & \boxed{\lambda : \mathtt{list}.(0)*} \\ \mathtt{n} & \boxed{\eta : \mathtt{word}_4} \end{array} \quad \lambda \neq 0$$

type $\mathtt{data} = \mathtt{word}_4$
type $\mathtt{list} =$
$\quad \mathtt{list}.(0)* \times \mathtt{data}$

```
if (*l != 0) {
  n = **l;
}
```

A storeless abstraction is often not enough

type $\mathtt{data} = \mathtt{word}_4$
type $\mathtt{list} =$
  $\mathtt{list}.(0)* \times \mathtt{data}$

$\mathbf{if}\ (*\mathtt{l} \mathrel{!=} 0)\ \{$
  $\mathtt{n} = **\mathtt{l};$
$\}$

$$\begin{array}{ll} \mathtt{l} & \boxed{\lambda : \mathtt{list}.(0)*} \\ \mathtt{n} & \boxed{\eta : \mathtt{word}_4} \end{array} \quad \lambda \neq 0$$

$\Downarrow$ assume$(*\mathtt{l} \mathrel{!=} 0)$ ✓

$$\begin{array}{ll} \mathtt{l} & \boxed{\lambda : \mathtt{list}.(0)*} \\ \mathtt{n} & \boxed{\eta : \mathtt{word}_4} \end{array} \quad \lambda \neq 0$$

$\Downarrow$ $\mathtt{n} = **\mathtt{l}$ ✗

A storeless abstraction is often not enough

Add points-to predicates to retain information about the heap

l $\boxed{\lambda : \texttt{list}.(0)*}$

n $\boxed{\eta : \texttt{word}_4}$

$\lambda \neq 0$

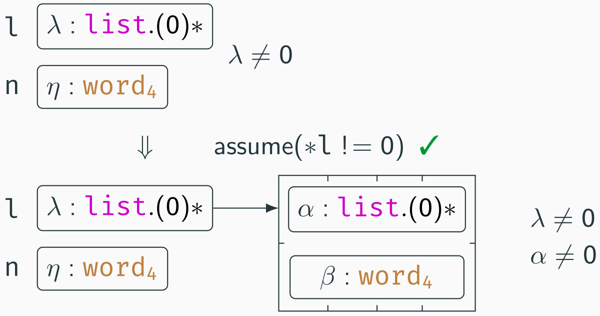
type $\texttt{data} = \texttt{word}_4$
type $\texttt{list} =$
  $\texttt{list}.(0)* \times \texttt{data}$

```
if (*l != 0) {
  n = **l;
}
```

A storeless abstraction is often not enough

Add points-to predicates to retain information about the heap

type $\texttt{data} = \texttt{word}_4$
type $\texttt{list} =$
  $\texttt{list}.(0)* \times \texttt{data}$



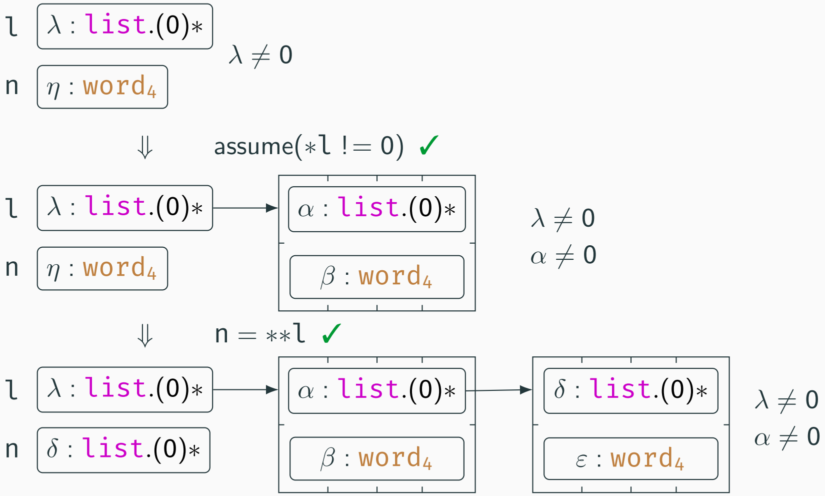

```
if (*l != 0) {
  n = **l;
}
```

A storeless abstraction is often not enough

Add points-to predicates to retain information about the heap

type $\texttt{data} = \texttt{word}_4$
type $\texttt{list} =$
  $\texttt{list}.(0)* \times \texttt{data}$

```
if (*l != 0) {
  n = **l;
}
```



Temporary violation of structural invariants

$\mathtt{l}$ $\boxed{\lambda_0 : \mathtt{word}_4}$

$\mathtt{n}$ $\boxed{\eta : \mathtt{list}.(0)*}$

type $\mathtt{data} = \mathtt{word}_4$
type $\mathtt{list} =$
$\quad \mathtt{list}.(0)* \times \mathtt{data}$

$\mathtt{l} = \textbf{malloc}_{\mathtt{list}}(8);$
$*\mathtt{l} = \mathtt{n};$
$*(\mathtt{l} + 4) = 0;$

Temporary violation of structural invariants

$\mathtt{l}$ $\boxed{\lambda_0 : \mathtt{word}_4}$

$\mathtt{n}$ $\boxed{\eta : \mathtt{list}.(0)*}$

$$\Downarrow \qquad \mathtt{l} = \textbf{malloc}_{\mathtt{list}}(8);$$

type $\mathtt{data} = \mathtt{word}_4$
type $\mathtt{list} =$
$\quad \mathtt{list}.(0)* \times \mathtt{data}$

$\mathtt{l} = \textbf{malloc}_{\mathtt{list}}(8);$
$*\mathtt{l} = \mathsf{n};$
$*(\mathtt{l} + 4) = 0;$

Temporary violation of structural invariants

$\mathtt{l}$ $\boxed{\lambda_0 : \mathtt{word}_4}$

$\mathtt{n}$ $\boxed{\eta : \mathtt{list}.(0)*}$

$\Downarrow$ $\mathtt{l} = \mathbf{malloc}_{\mathtt{list}}(8);$

$\mathtt{l}$ $\boxed{\lambda : \mathtt{list}.(0)*}$

$\mathtt{n}$ $\boxed{\eta : \mathtt{list}.(0)*}$ ✗

Incorrect: $\mathtt{l}$ should point to uninitialized memory

type $\mathtt{data} = \mathtt{word}_4$
type $\mathtt{list} =$
$\quad \mathtt{list}.(0)* \times \mathtt{data}$

$\mathtt{l} = \mathbf{malloc}_{\mathtt{list}}(8);$
$*\mathtt{l} = \mathtt{n};$
$*(\mathtt{l} + 4) = 0;$

Temporary violation of structural invariants

$\mathtt{l}$ $\boxed{\lambda_0 : \mathtt{word}_4}$

$\mathtt{n}$ $\boxed{\eta : \mathtt{list}.(0)*}$

$\Downarrow$ $\mathtt{l} = \textbf{malloc}_{\mathtt{list}}(8);$

$\mathtt{l}$ $\boxed{\lambda : \mathtt{word}_8.(0)*}$

$\mathtt{n}$ $\boxed{\eta : \mathtt{list}.(0)*}$

Correct but very imprecise

type $\mathtt{data} = \mathtt{word}_4$
type $\mathtt{list} =$
$\mathtt{list}.(0)* \times \mathtt{data}$

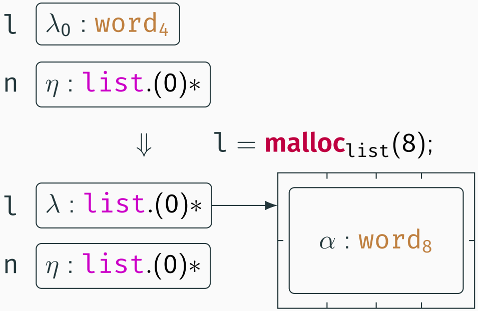
$\mathtt{l} = \textbf{malloc}_{\mathtt{list}}(8);$
$*\mathtt{l} = \mathtt{n};$
$*(\mathtt{l} + 4) = 0;$

Temporary violation of structural invariants

Represent memory writes with "staged" points-to predicates, delaying type verification

type $\texttt{data} = \texttt{word}_4$
type $\texttt{list} =$
$\quad \texttt{list}.(0)* \times \texttt{data}$

$\texttt{l} = \textbf{malloc}_{\texttt{list}}(8);$
$*\texttt{l} = \texttt{n};$
$*(\texttt{l} + 4) = 0;$

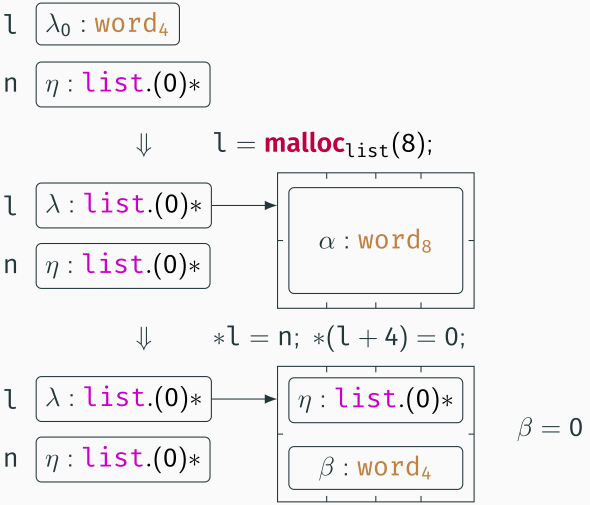


Temporary violation of structural invariants

Represent memory writes with "staged" points-to predicates, delaying type verification

type $\texttt{data} = \texttt{word}_4$
type $\texttt{list} =$
$\quad \texttt{list}.(0)* \times \texttt{data}$

$\texttt{l} = \textbf{malloc}_{\texttt{list}}(8);$
$*\texttt{l} = \texttt{n};$
$*(\texttt{l} + 4) = 0;$

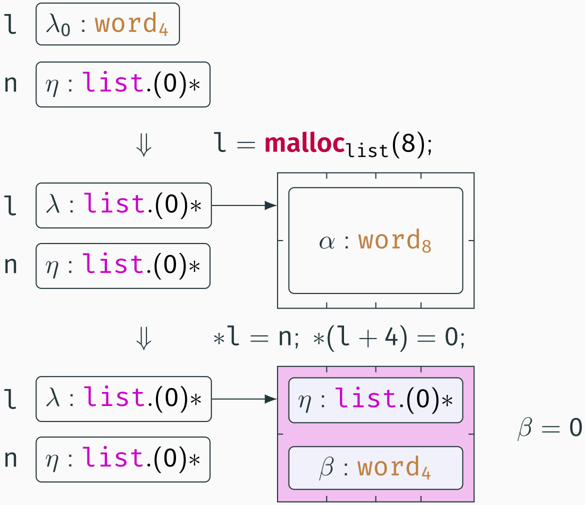


Temporary violation of structural invariants

Represent memory writes with “staged” points-to predicates, delaying type verification

type $\mathtt{data} = \mathtt{word}_4$
type $\mathtt{list} =$
$\quad \mathtt{list}.(0)* \times \mathtt{data}$

$\mathtt{l} = \mathbf{malloc}_{\mathtt{list}}(8);$
$*\mathtt{l} = \mathtt{n};$
$*(\mathtt{l}+4) = 0;$



Temporary violation of structural invariants

Represent memory writes with "staged" points-to predicates, delaying type verification

type $\texttt{data} = \texttt{word}_4$
type $\texttt{list} =$
$\quad \texttt{list}.(0)* \times \texttt{data}$

$\texttt{l} = \textbf{malloc}_{\texttt{list}}(8);$
$*\texttt{l} = \texttt{n};$
$*(\texttt{l} + 4) = 0;$

$\texttt{l}\ \boxed{\lambda_0 : \texttt{word}_4}$

$\texttt{n}\ \boxed{\eta : \texttt{list}.(0)*}$

$\Downarrow \qquad \texttt{l} = \textbf{malloc}_{\texttt{list}}(8);$

$\texttt{l}\ \boxed{\lambda : \texttt{list}.(0)*} \rightarrow \boxed{\alpha : \texttt{word}_8}$

$\texttt{n}\ \boxed{\eta : \texttt{list}.(0)*}$

$\Downarrow \qquad *\texttt{l} = \texttt{n};\ *(\texttt{l} + 4) = 0;$

$\texttt{l}\ \boxed{\lambda : \texttt{list}.(0)*}$

$\texttt{n}\ \boxed{\eta : \texttt{list}.(0)*}$

✓

$\beta = 0$

Building a C analyzer

Full analysis: 3,300 lines of OCaml in the Codex abstract interpretation library

FRAMA-C/CODEX, a module of Frama-C

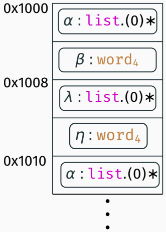
Numerical abstract domain

- Intervals with congruence information
- Bitwise abstraction
- Symbolic equalities and inequalities

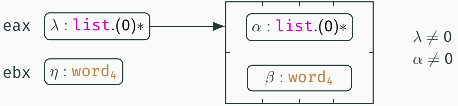
Building a machine code analyzer

BINSEC/CODEX, a module of BINSEC

Memory abstraction



$\times$



"Array of bytes" abstract domain

For the stack and global data

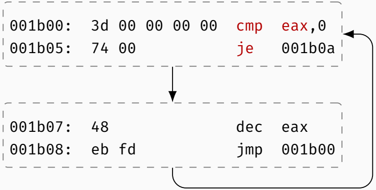
Type-based abstract domain

For the rest of memory

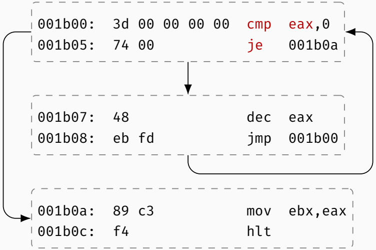
Machine code analysis: Incremental inference of control flow

```
001b00:  3d 00 00 00
001b04:  00 74 00 48
001b08:  eb f8 89 c3
001b0c:  f4 00 00 00
```

(a) Binary program



(b) Partial CFG



(c) Invariant CFG

First experimental evaluation: Shape benchmarks

We analyzed all shape benchmarks from [1] and [2], featuring:

- Complex data structure manipulation
- Unstructured sharing (e.g. union-find, graphs)

34 programs, between 9 and 329 lines of code (mean 61)

Target property

Spatial memory safety + Preservation of structural invariants

We evaluate:

- Analysis time
- **Precision** (number of alarms)
- Ease of setup and **automation** (number of annotations)

[1] Li et al. "Shape Analysis for Unstructured Sharing". In: *SAS*. 2015
[2] Li et al. "Semantic-Directed Clumping of Disjunctive Abstract States". In: *POPL*. 2017

Analysis process

Generate physical types automatically from C types

Refine types if necessary

Run the analysis

✗ ✓

“annotations”

- type predicates
- array lengths

First experimental evaluation: Shape benchmarks

Benchmark	Annotations gen	ed	pre	LOC	C Time	$\mapsto$	$\not\mapsto$	O0 Time	$\mapsto$	$\not\mapsto$	O1 Time	$\mapsto$	$\not\mapsto$	O2 Time	$\mapsto$	$\not\mapsto$	O3 Time	$\mapsto$	$\not\mapsto$
sll-delmin	11	0	1	25	0.27	0	0	0.13	0	0	0.15	0	0	0.15	0	0	0.13	0	0
sll-delminmax			1	49	0.30	0	0	0.19	0	0	0.17	0	0	0.17	0	0	0.16	0	0
psll-bsort	10	0	0	25	0.30	0	22	0.41	0	3	0.25	0	3	0.26	0	3	0.29	0	3
psll-reverse			0	11	0.28	0	2	0.10	0	1	0.13	0	1	0.10	0	1	0.10	0	1
psll-isort			0	20	0.29	0	2	0.34	0	1	0.34	0	1	0.32	0	1	0.33	0	1
bstree-find	12	0	1	26	0.27	0	0	0.14	0	0	0.13	0	0	0.15	0	0	0.16	0	0
gdll-findmin	25	5	1	49	0.50	0	0	0.41	0	0	0.39	0	0	0.41	0	0	0.42	0	0
gdll-findmax			1	58	0.55	0	0	0.33	0	0	0.22	0	0	0.21	0	0	0.20	0	0
gdll-find			1	78	0.56	0	0	0.15	0	0	0.15	0	0	0.14	0	0	0.14	0	0
gdll-index			1	55	0.53	0	0	0.32	0	0	0.33	0	0	0.30	0	0	0.29	0	0
gdll-delete			1	107	0.57	0	2	0.16	0	0	0.14	0	0	0.13	0	0	0.13	0	0
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
graph-nodelisttrav	23	0	1	12	0.20	0	0	0.10	0	0	0.10	0	0	0.10	0	0	0.11	0	0
graph-path			1	19	0.21	0	14	0.15	0	5	0.16	0	0	0.14	0	0	0.16	0	0
graph-pathrand			1	25	0.22	0	10	0.13	0	0	0.21	0	0	0.12	0	0	0.11	0	0
graph-edgeadd			1	15	0.27	0	2	0.12	0	1	0.11	0	1	0.10	0	1	0.10	0	1
graph-nodeadd			1	15	0.26	0	2	0.10	0	1	0.08	0	1	0.09	0	1	0.10	0	1
graph-edgedelete			1	11	0.20	0	2	0.10	0	1	0.10	0	0	0.10	0	0	0.11	0	0
graph-edgeiter			1	22	0.23	0	0	0.13	0	0	0.11	0	0	0.12	0	0	0.12	0	0
uf-find	33	3	1	11	0.31	0	24	0.07	0	6	0.09	0	0	0.08	0	0	0.07	0	0
uf-merge			1	17	0.34	0	50	0.13	0	7	0.18	0	0	0.18	0	0	0.15	0	0
uf-make			0	9	0.31	0	4	0.05	0	3	0.06	0	3	0.07	0	3	0.06	0	3
Total verified						30	13		30	16		30	21		30	21		30	21

The annotation effort is low

Annotation effort varies between **0** and **12 lines** (on average **3.2 lines**).

	min.	mean	median	max.
Annotation/code ratio	0 %	3.2 %	2.7 %	7.8 %

Precision

	C	binary -O0	binary -O1	binary -O2	binary -O3
Property verified ✓	**30** / 34	**30** / 34	**30** / 34	**30** / 34	**30** / 34

Points-to predicates are necessary

Without points-to predicates:

	C	binary -O0	binary -O1	binary -O2	binary -O3
Property verified ✓	13 / 34	16 / 34	21 / 34	21 / 34	21 / 34

The analysis time is predictable

All analyses complete in **less than 1 s**.

Comparison with shape analyses from [2]

	Base shape analysis	Guided clumping [2]	**This work**
javl-insert (baseline)	✗	1.84	0.43
javl-insert-32×	✗	129.9	40

Run times (s)

[2] Li et al. “Semantic-Directed Clumping of Disjunctive Abstract States”. In: *POPL*. 2017

Overview: Kernel verification using BINSEC/CODEX

We propose a method based on BINSEC/CODEX to verify **APE** and **ARTE** on **embedded kernels**.

- **Automated**
  - Abstract interpretation *infers* kernel invariants
  - APE is an implicit property (no specification to write)
- **Comprehensive**
  - *Machine code* verification on the kernel executable
- **Generic**
  - *Parameterized* verification by *abstracting* task-specific data with *types*
- **Practical**
  - Comprehensive evaluation on challenging case studies
    unmodified version of ASTERIOS RTK, 96 variants of EducRTOS

Implicit properties

Definition (Implicit property)

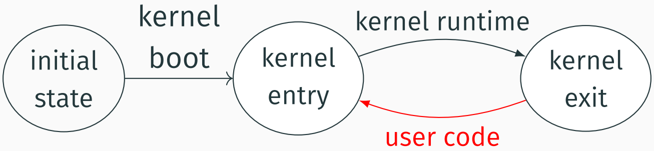
An implicit property is a property that does not depend on a particular program.

Example

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

Verifying an implicit property does not require to **write a specification**.

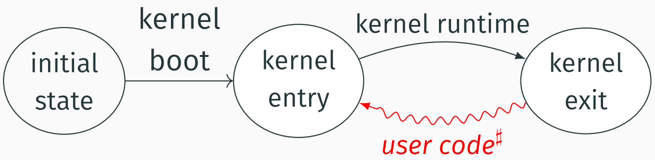
Abstraction of user code



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

The user code is unknown

Abstraction of user code



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).

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.

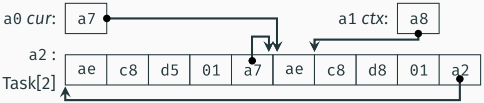
Proof.

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

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

Example in-context analysis

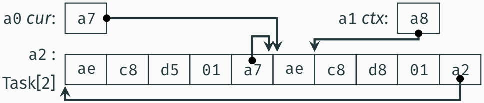
Initial state:




```
Task *cur;

void handle_timer() {
  /* Save task context */
  cur→ctx = interrupted_ctx;
  /* Schedule next task */
  cur = cur→next;
  /* Load new memory protection */
  mpu = cur→mem_table;
  /* Give control to new task */
  switch_context(&cur→ctx);
}
```

Example in-context analysis

Initial state:

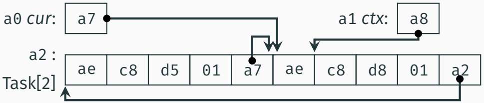



```
Task *cur;

void handle_timer() {
  /* Save task context */
  cur→ctx = interrupted_ctx;
  /* Schedule next task */
  cur = cur→next;
  /* Load new memory protection */
  mpu = cur→mem_table;
  /* Give control to new task */
  switch_context(&cur→ctx);
}
```

$\mathrm{cur} \in \{0xa7\},\ \mathrm{mpu} \in \{0xae\}$

Example in-context analysis

Initial state:



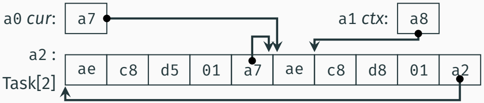

```
Task *cur;

void handle_timer() {
  /* Save task context */
  cur→ctx = interrupted_ctx;
  /* Schedule next task */
  cur = cur→next;
  /* Load new memory protection */
  mpu = cur→mem_table;
  /* Give control to new task */
  switch_context(&cur→ctx);
}
```

$\mathrm{cur} \in \{0xa7\}, \mathrm{mpu} \in \{0xae\}$

$\mathrm{cur} \in \{0xa7\}, \mathrm{mpu} \in \{0xae\}$

Example in-context analysis

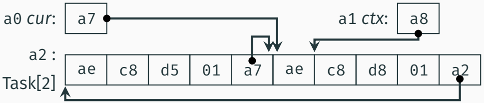
Initial state:




```
Task *cur;

void handle_timer() {                    cur ∈ {0xa7}, mpu ∈ {0xae}
  /* Save task context */
  cur→ctx = interrupted_ctx;             cur ∈ {0xa7}, mpu ∈ {0xae}
  /* Schedule next task */
  cur = cur→next;                        cur ∈ {0xa2}, mpu ∈ {0xae}
  /* Load new memory protection */
  mpu = cur→mem_table;
  /* Give control to new task */
  switch_context(&cur→ctx);
}
```

Example in-context analysis

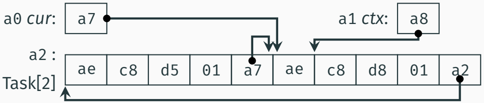
Initial state:




```
Task *cur;

void handle_timer() {            cur ∈ {0xa7}, mpu ∈ {0xae}
  /* Save task context */
  cur→ctx = interrupted_ctx;     cur ∈ {0xa7}, mpu ∈ {0xae}
  /* Schedule next task */
  cur = cur→next;                cur ∈ {0xa2}, mpu ∈ {0xae}
  /* Load new memory protection */
  mpu = cur→mem_table;           cur ∈ {0xa2}, mpu ∈ {0xae}
  /* Give control to new task */
  switch_context(&cur→ctx);
}
```

Example in-context analysis

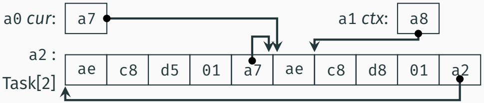
Initial state:




```
Task *cur;

void handle_timer() {            // cur ∈ {0xa7}, mpu ∈ {0xae}
  /* Save task context */
  cur→ctx = interrupted_ctx;     // cur ∈ {0xa7}, mpu ∈ {0xae}
  /* Schedule next task */
  cur = cur→next;                // cur ∈ {0xa2}, mpu ∈ {0xae}
  /* Load new memory protection */
  mpu = cur→mem_table;           // cur ∈ {0xa2}, mpu ∈ {0xae}
  /* Give control to new task */
  switch_context(&cur→ctx);      // cur ∈ {0xa2}, mpu ∈ {0xae}
}
```

Example in-context analysis

Initial state:



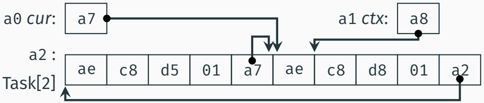

```c
Task *cur;

void handle_timer() {
  /* Save task context */
  cur->ctx = interrupted_ctx;
  /* Schedule next task */
  cur = cur->next;
  /* Load new memory protection */
  mpu = cur->mem_table;
  /* Give control to new task */
  switch_context(&cur->ctx);
}
```

`void handle_timer() {`	$cur \in \{0xa7\}$, $mpu \in \{0xae\}$
`cur->ctx = interrupted_ctx;`	$cur \in \{0xa7\}$, $mpu \in \{0xae\}$
`cur = cur->next;`	$cur \in \{0xa2\}$, $mpu \in \{0xae\}$
`mpu = cur->mem_table;`	$cur \in \{0xa2\}$, $mpu \in \{0xae\}$
`switch_context(&cur->ctx);`	$cur \in \{0xa2\}$, $mpu \in \{0xae\}$

user code$^\sharp$

Example in-context analysis

Initial state:




```
Task *cur;

void handle_timer() {
  /* Save task context */
  cur→ctx = interrupted_ctx;
  /* Schedule next task */
  cur = cur→next;
  /* Load new memory protection */
  mpu = cur→mem_table;
  /* Give control to new task */
  switch_context(&cur→ctx);
}
```

$\mathtt{cur} \in \{0xa7, 0xa2\}$, $\mathtt{mpu} \in \{0xae\}$

$\mathtt{cur} \in \{0xa7\}$, $\mathtt{mpu} \in \{0xae\}$

$\mathtt{cur} \in \{0xa2\}$, $\mathtt{mpu} \in \{0xae\}$

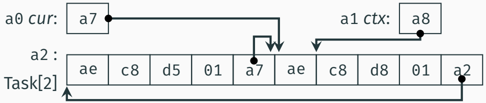
$\mathtt{cur} \in \{0xa2\}$, $\mathtt{mpu} \in \{0xae\}$

$\mathtt{cur} \in \{0xa2\}$, $\mathtt{mpu} \in \{0xae\}$

user code$^\sharp$

Example in-context analysis

Initial state:




```
Task *cur;

void handle_timer() {
  /* Save task context */
  cur→ctx = interrupted_ctx;
  /* Schedule next task */
  cur = cur→next;
  /* Load new memory protection */
  mpu = cur→mem_table;
  /* Give control to new task */
  switch_context(&cur→ctx);
}
```

$\mathsf{cur} \in \{0xa7, 0xa2\}$, $\mathsf{mpu} \in \{0xae\}$

$\mathsf{cur} \in \{0xa7, 0xa2\}$, $\mathsf{mpu} \in \{0xae\}$

$\mathsf{cur} \in \{0xa2\}$, $\mathsf{mpu} \in \{0xae\}$

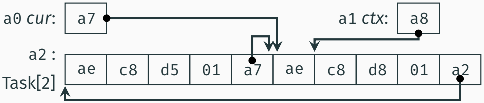
$\mathsf{cur} \in \{0xa2\}$, $\mathsf{mpu} \in \{0xae\}$

$\mathsf{cur} \in \{0xa2\}$, $\mathsf{mpu} \in \{0xae\}$

user code$^\sharp$

Example in-context analysis

Initial state:



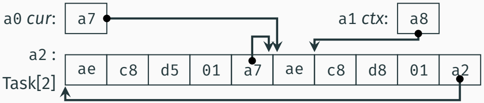

```
Task *cur;

void handle_timer() {
  /* Save task context */
  cur→ctx = interrupted_ctx;
  /* Schedule next task */
  cur = cur→next;
  /* Load new memory protection */
  mpu = cur→mem_table;
  /* Give control to new task */
  switch_context(&cur→ctx);
}
```

- `void handle_timer() {` — $\mathtt{cur} \in \{0xa7, 0xa2\}$, $\mathtt{mpu} \in \{0xae\}$
- `cur→ctx = interrupted_ctx;` — $\mathtt{cur} \in \{0xa7, 0xa2\}$, $\mathtt{mpu} \in \{0xae\}$
- `cur = cur→next;` — $\mathtt{cur} \in \{0xa2, 0xa7\}$, $\mathtt{mpu} \in \{0xae\}$
- `mpu = cur→mem_table;` — $\mathtt{cur} \in \{0xa2\}$, $\mathtt{mpu} \in \{0xae\}$
- `switch_context(&cur→ctx);` — $\mathtt{cur} \in \{0xa2\}$, $\mathtt{mpu} \in \{0xae\}$

user code$^\sharp$

Example in-context analysis

Initial state:




```
Task *cur;

void handle_timer() {
  /* Save task context */
  cur→ctx = interrupted_ctx;
  /* Schedule next task */
  cur = cur→next;
  /* Load new memory protection */
  mpu = cur→mem_table;
  /* Give control to new task */
  switch_context(&cur→ctx);
}
```

$\mathtt{cur} \in \{0xa7, 0xa2\},\ \mathtt{mpu} \in \{0xae\}$

$\mathtt{cur} \in \{0xa7, 0xa2\},\ \mathtt{mpu} \in \{0xae\}$

$\mathtt{cur} \in \{0xa2, 0xa7\},\ \mathtt{mpu} \in \{0xae\}$

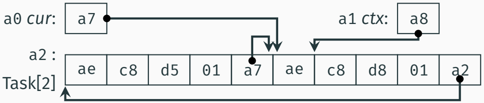
$\mathtt{cur} \in \{0xa2, 0xa7\},\ \mathtt{mpu} \in \{0xae\}$

$\mathtt{cur} \in \{0xa2\},\ \mathtt{mpu} \in \{0xae\}$

user code$^\sharp$

Example in-context analysis

Initial state:




```
Task *cur;

void handle_timer() {
  /* Save task context */
  cur→ctx = interrupted_ctx;
  /* Schedule next task */
  cur = cur→next;
  /* Load new memory protection */
  mpu = cur→mem_table;
  /* Give control to new task */
  switch_context(&cur→ctx);
}
```

$\mathtt{cur} \in \{0xa7, 0xa2\}$, $\mathtt{mpu} \in \{0xae\}$

$\mathtt{cur} \in \{0xa7, 0xa2\}$, $\mathtt{mpu} \in \{0xae\}$

$\mathtt{cur} \in \{0xa2, 0xa7\}$, $\mathtt{mpu} \in \{0xae\}$

$\mathtt{cur} \in \{0xa2, 0xa7\}$, $\mathtt{mpu} \in \{0xae\}$

$\mathtt{cur} \in \{0xa2, 0xa7\}$, $\mathtt{mpu} \in \{0xae\}$

user code$^\sharp$

Example in-context analysis

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

Shortcomings of in-context analyses

In-context analysis is:

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

Key idea

We use the type-based abstract domain to abstract task data.

Example parameterized analysis

$$\texttt{type}\ \texttt{Memory\_table} = \{x : \texttt{int} \mid x > \texttt{kernel\_last\_addr}\} \times \texttt{int}$$
$$\texttt{type}\ \texttt{Context} = \cdots \times \{\mathit{privilege} : \texttt{int} \mid \mathit{privilege} = 0\} \times \cdots$$
$$\texttt{type}\ \texttt{Task} = \{x : \texttt{Memory\_table}.(0)* \mid x \neq 0\} \times \texttt{Context} \times \{x : \texttt{Task}.(0)* \mid x \neq 0\}$$

```
Task *cur;

void handle_timer() {
  /* Save task context */
  cur→ctx = interrupted_ctx;
  /* Schedule next task */
  cur = cur→next;
  /* Load new memory protection */
  mpu = cur→mem_table;
  /* Give control to new task */
  switch_context(&cur→ctx);
}
```

Example parameterized analysis

$$\texttt{type Memory\_table} = \{x : \texttt{int} \mid x > \texttt{kernel\_last\_addr}\} \times \texttt{int}$$

$$\texttt{type Context} = \cdots \times \{\mathit{privilege} : \texttt{int} \mid \mathit{privilege} = 0\} \times \cdots$$

$$\texttt{type Task} = \{x : \texttt{Memory\_table}.(0)* \mid x \neq 0\} \times \texttt{Context} \times \{x : \texttt{Task}.(0)* \mid x \neq 0\}$$

```
Task *cur;

void handle_timer() {
  /* Save task context */
  cur→ctx = interrupted_ctx;
  /* Schedule next task */
  cur = cur→next;
  /* Load new memory protection */
  mpu = cur→mem_table;
  /* Give control to new task */
  switch_context(&cur→ctx);
}
```

$\alpha \neq 0 \qquad \beta \neq 0$

cur $\boxed{\alpha : \texttt{Task}.(0)*}$ mpu $\boxed{\beta : \texttt{Memory\_table}.(0)*}$

Example parameterized analysis

$$\texttt{type Memory\_table} = \{x : \texttt{int} \mid x > \texttt{kernel\_last\_addr}\} \times \texttt{int}$$
$$\texttt{type Context} = \cdots \times \{\mathit{privilege} : \texttt{int} \mid \mathit{privilege} = 0\} \times \cdots$$
$$\texttt{type Task} = \{x : \texttt{Memory\_table}.(0)* \mid x \neq 0\} \times \texttt{Context} \times \{x : \texttt{Task}.(0)* \mid x \neq 0\}$$

```
Task *cur;

void handle_timer() {
  /* Save task context */
  cur→ctx = interrupted_ctx;
  /* Schedule next task */
  cur = cur→next;
  /* Load new memory protection */
  mpu = cur→mem_table;
  /* Give control to new task */
  switch_context(&cur→ctx);
}
```

$\alpha \neq 0 \qquad \beta \neq 0$

At `void handle_timer() {`: cur $\boxed{\alpha : \texttt{Task}.(0)*}$ mpu $\boxed{\beta : \texttt{Memory\_table}.(0)*}$

At `cur→ctx = interrupted_ctx;`: cur $\boxed{\alpha : \texttt{Task}.(0)*}$ mpu $\boxed{\beta : \texttt{Memory\_table}.(0)*}$

Example parameterized analysis

$$\texttt{type } \texttt{Memory\_table} = \{x : \texttt{int} \mid x > \texttt{kernel\_last\_addr}\} \times \texttt{int}$$
$$\texttt{type } \texttt{Context} = \cdots \times \{privilege : \texttt{int} \mid privilege = 0\} \times \cdots$$
$$\texttt{type } \texttt{Task} = \{x : \texttt{Memory\_table}.(0)* \mid x \neq 0\} \times \texttt{Context} \times \{x : \texttt{Task}.(0)* \mid x \neq 0\}$$

$\alpha \neq 0 \quad \beta \neq 0 \quad \delta \neq 0$

```
Task *cur;

void handle_timer() {
  /* Save task context */
  cur→ctx = interrupted_ctx;
  /* Schedule next task */
  cur = cur→next;
  /* Load new memory protection */
  mpu = cur→mem_table;
  /* Give control to new task */
  switch_context(&cur→ctx);
}
```

At `void handle_timer() {`: cur $\boxed{\alpha : \texttt{Task}.(0)*}$ mpu $\boxed{\beta : \texttt{Memory\_table}.(0)*}$

At `cur→ctx = interrupted_ctx;`: cur $\boxed{\alpha : \texttt{Task}.(0)*}$ mpu $\boxed{\beta : \texttt{Memory\_table}.(0)*}$

At `cur = cur→next;`: cur $\boxed{\delta : \texttt{Task}.(0)*}$ mpu $\boxed{\beta : \texttt{Memory\_table}.(0)*}$

Example parameterized analysis

$$\texttt{type Memory\_table} = \{x : \texttt{int} \mid x > \texttt{kernel\_last\_addr}\} \times \texttt{int}$$
$$\texttt{type Context} = \cdots \times \{privilege : \texttt{int} \mid privilege = 0\} \times \cdots$$
$$\texttt{type Task} = \{x : \texttt{Memory\_table}.(0)* \mid x \neq 0\} \times \texttt{Context} \times \{x : \texttt{Task}.(0)* \mid x \neq 0\}$$

$\alpha \neq 0 \quad \beta \neq 0 \quad \delta \neq 0 \quad \varepsilon \neq 0$

```
Task *cur;

void handle_timer() {
  /* Save task context */
  cur→ctx = interrupted_ctx;
  /* Schedule next task */
  cur = cur→next;
  /* Load new memory protection */
  mpu = cur→mem_table;
  /* Give control to new task */
  switch_context(&cur→ctx);
}
```

	cur	mpu
`void handle_timer() {`	$\boxed{\alpha : \texttt{Task}.(0)*}$	$\boxed{\beta : \texttt{Memory\_table}.(0)*}$
`cur→ctx = interrupted_ctx;`	$\boxed{\alpha : \texttt{Task}.(0)*}$	$\boxed{\beta : \texttt{Memory\_table}.(0)*}$
`cur = cur→next;`	$\boxed{\delta : \texttt{Task}.(0)*}$	$\boxed{\beta : \texttt{Memory\_table}.(0)*}$
`mpu = cur→mem_table;`	$\boxed{\delta : \texttt{Task}.(0)*}$	$\boxed{\varepsilon : \texttt{Memory\_table}.(0)*}$

Example parameterized analysis

$$\texttt{type Memory\_table} = \{x : \texttt{int} \mid x > \texttt{kernel\_last\_addr}\} \times \texttt{int}$$
$$\texttt{type Context} = \cdots \times \{\mathit{privilege} : \texttt{int} \mid \mathit{privilege} = 0\} \times \cdots$$
$$\texttt{type Task} = \{x : \texttt{Memory\_table}.(0)* \mid x \neq 0\} \times \texttt{Context} \times \{x : \texttt{Task}.(0)* \mid x \neq 0\}$$

```
Task *cur;
```

$\alpha \neq 0 \quad \beta \neq 0 \quad \delta \neq 0 \quad \varepsilon \neq 0$

	cur	mpu
`void handle_timer() {`	$\boxed{\alpha : \texttt{Task}.(0)*}$	$\boxed{\beta : \texttt{Memory\_table}.(0)*}$
`  /* Save task context */`		
`  cur→ctx = interrupted_ctx;`	$\boxed{\alpha : \texttt{Task}.(0)*}$	$\boxed{\beta : \texttt{Memory\_table}.(0)*}$
`  /* Schedule next task */`		
`  cur = cur→next;`	$\boxed{\delta : \texttt{Task}.(0)*}$	$\boxed{\beta : \texttt{Memory\_table}.(0)*}$
`  /* Load new memory protection */`		
`  mpu = cur→mem_table;`	$\boxed{\delta : \texttt{Task}.(0)*}$	$\boxed{\varepsilon : \texttt{Memory\_table}.(0)*}$
`  /* Give control to new task */`		
`  switch_context(&cur→ctx);`	$\boxed{\delta : \texttt{Task}.(0)*}$	$\boxed{\varepsilon : \texttt{Memory\_table}.(0)*}$
`}`		

Example parameterized analysis

$$\texttt{type}\ \texttt{Memory\_table} = \{x : \texttt{int} \mid x > \texttt{kernel\_last\_addr}\} \times \texttt{int}$$
$$\texttt{type}\ \texttt{Context} = \cdots \times \{\mathit{privilege} : \texttt{int} \mid \mathit{privilege} = 0\} \times \cdots$$
$$\texttt{type}\ \texttt{Task} = \{x : \texttt{Memory\_table}.(0)* \mid x \neq 0\} \times \texttt{Context} \times \{x : \texttt{Task}.(0)* \mid x \neq 0\}$$

```
Task *cur;
```

$\alpha \neq 0 \quad \beta \neq 0 \quad \delta \neq 0 \quad \varepsilon \neq 0$

```
void handle_timer() {
  /* Save task context */
  cur→ctx = interrupted_ctx;
  /* Schedule next task */
  cur = cur→next;
  /* Load new memory protection */
  mpu = cur→mem_table;
  /* Give control to new task */
  switch_context(&cur→ctx);
}
```

Program point	cur	mpu
`void handle_timer() {`	$\alpha : \texttt{Task}.(0)*$	$\beta : \texttt{Memory\_table}.(0)*$
`cur→ctx = interrupted_ctx;`	$\alpha : \texttt{Task}.(0)*$	$\beta : \texttt{Memory\_table}.(0)*$
`cur = cur→next;`	$\delta : \texttt{Task}.(0)*$	$\beta : \texttt{Memory\_table}.(0)*$
`mpu = cur→mem_table;`	$\delta : \texttt{Task}.(0)*$	$\varepsilon : \texttt{Memory\_table}.(0)*$
`switch_context(&cur→ctx);`	$\delta : \texttt{Task}.(0)*$	$\varepsilon : \texttt{Memory\_table}.(0)*$

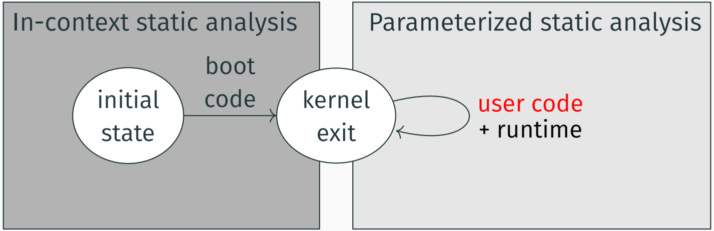
user code$^\sharp$

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*.

		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	✓

2 versions

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

Specific = restriction on stack sizes

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)

  ...

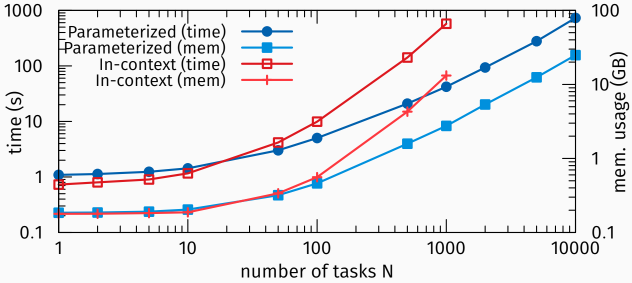
Verification time: from 1.6 s to 73 s.

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

- A type-based memory abstract domain
  - Points-to predicates improve precision without disjunctions
- Two low-level program analyses in Binsec and Frama-C
  - Verify structural invariants and spatial memory safety
  - Less precise but more easily applicable than shape analyses
- A method to verify absence of run-time errors and absence of privilege escalation on embedded kernels.

Perspectives

- Collaboration with other proof methods
  - Other methods verify more properties but establishing invariants is costly
  - Use facts inferred by our analysis as input to other verification tools
- Enrich the type system to verify stronger properties
  - Full dependent types (dynamic-size arrays, non-interference)
  - Combination with separation logic to specify local separation constraints