No Crash, No Exploit: Automated Verification of Embedded Kernels

Olivier Nicole \textsuperscript{1,2} Matthieu Lemerre \textsuperscript{1} Sébastien Bardin \textsuperscript{1} Xavier Rival \textsuperscript{2}

RTAS 2021

\textsuperscript{1} Université Paris-Saclay, CEA List
\textsuperscript{2} ENS, PSL University / Inria
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 ⇒ the whole system is compromised

Only way to guarantee their absence: formal methods.
How to protect an OS kernel against its worst defects?

Worst possible bugs for an OS kernel:

• **Runtime errors**  Division by zero, illegal memory access...
How to protect an OS kernel against its worst defects?

Worst possible bugs for an OS kernel:

- **Runtime errors**  Division by zero, illegal memory access...
  
  The kernel **crashes**
How to protect an OS kernel against its worst defects?

Worst possible bugs for an OS kernel:

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

Privilege escalation  Kernel protections are bypassed  $\implies$  the whole system is compromised

Only way to guarantee their absence: **formal methods.**
How to protect an OS kernel against its worst defects?

Worst possible bugs for an OS kernel:

- **Runtime errors**  Division by zero, illegal memory access...
  The kernel **crashes** \(\implies\) the whole system crashes
How to protect an OS kernel against its worst defects?

Worst possible bugs for an OS kernel:

- **Runtime errors**  Division by zero, illegal memory access...
  The kernel **crashes** $\implies$ the whole system crashes
- **Privilege escalation**
How to protect an OS kernel against its worst defects?

Worst possible bugs for an OS kernel:

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

```c
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

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

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

    volatile asm("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();
}
```
\[ \forall \text{ tasks}, \ (\text{kernel} \oplus \text{tasks}) \models \text{APE, ARTE} \]

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

<table>
<thead>
<tr>
<th>Interactive proof</th>
<th>Deductive verification</th>
</tr>
</thead>
<tbody>
<tr>
<td>• seL4 [SOSP’09]</td>
<td>• CertiKOS [OSDI’16]</td>
</tr>
<tr>
<td>• Verve [PLDI’10]</td>
<td>• Komodo [SOSP’17]</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Interactive proof</th>
<th>Deductive verification</th>
</tr>
</thead>
<tbody>
<tr>
<td>• seL4 [SOSP’09]</td>
<td>• Verve [PLDI’10]</td>
</tr>
<tr>
<td>• CertiKOS [OSDI’16]</td>
<td>• Komodo [SOSP’17]</td>
</tr>
</tbody>
</table>

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

**“Push-button” verification**

- 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**

- Infers all invariants
- Handles unbounded loops
- Handles parameterized verification
- Low annotation burden (e.g. 58 lines)
Verification principle
Abstract interpretation basics

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

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

• 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.
Abstract interpretation basics

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

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

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

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

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

```java
int i = 100;  \hspace{1cm} i \in \{100\}
int x = 0;  \hspace{1cm} i \in \{100\}, \ x \in \{0\}
while(i > 1) {
    i--;
}
int x = 42 / i;
```
Abstract each **numeric variable** by an **interval**.

```c
int i = 100;  i ∈ \{100\}
int x = 0;  i ∈ \{100\}, x ∈ \{0\}
while(i > 1) {
    i--;  i ∈ \{100\}, x ∈ \{0\}
    i--;  i ∈ \{99\}, x ∈ \{0\}
}
int x = 42 / i;
```
Abstract interpretation basics

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

```c
int i = 100; \quad \bullet \quad i \in \{100\}
int x = 0; \quad \bullet \quad i \in \{100\}, x \in \{0\}
while(i > 1) {
    \quad \bullet \quad i \in [99,100], x \in \{0\}
    i--; \quad \bullet \quad i \in \{99\}, x \in \{0\}
}
int x = 42 / i;
```
Abstract interpretation basics

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

```cpp
int i = 100;  // i ∈ \{100\}
int x = 0;    // i ∈ \{100\}, x ∈ \{0\}
while(i > 1) {
    i--;      // i ∈ [99, 100], x ∈ \{0\}
    i--;      // i ∈ [98, 99], x ∈ \{0\}
}
int x = 42 / i;
```

• 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.

\(i \in \{100\}\)
\(i \in [99, 100]\), \(x \in \{0\}\)
\(i \in [98, 99]\), \(x \in \{0\}\)
Abstract interpretation basics

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

```plaintext
int i = 100;  \quad i \in \{100\}
int x = 0;  \quad i \in \{100\}, \ x \in \{0\}
while(i > 1) {
    i--;  \quad i \in [98,100], \ x \in \{0\}
    i--;  \quad i \in [98,99], \ x \in \{0\}
}
int x = 42 / i;
```
Abstract interpretation basics

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

```c
int i = 100;  ➔ i ∈ {100}
int x = 0;    ➔ i ∈ {100}, x ∈ {0}
while(i > 1) {
    i--; ➔ i ∈ [98, 100], x ∈ {0}
    i--; ➔ i ∈ [97, 99], x ∈ {0}
}
int x = 42 / i;
```
Abstract interpretation basics

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

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

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

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

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

```c
int i = 100; i ∈ {100}
int x = 0; i ∈ {100}, x ∈ {0}
while(i > 1) {
    i--; i ∈ [2, 100], x ∈ {0}
}
int x = 42 / i; i ∈ {1}, x ∈ {42}
```
Abstract each **numeric variable** by an **interval**.

\[ i \in \{100\} \]

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

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

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

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

\[ i \in \{1\}, \ x \in \{42\} \]
Abstract each **numeric variable** by an **interval**.

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

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

The **user code** is unknown

⇒ 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

<table>
<thead>
<tr>
<th>Theorem</th>
</tr>
</thead>
<tbody>
<tr>
<td>If the system satisfies a non-trivial invariant, then no privilege escalation is possible on that system.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Proof.</th>
</tr>
</thead>
<tbody>
<tr>
<td>If the system fails to self-protect, the empowered attacker can reach any state.</td>
</tr>
</tbody>
</table>

→ APE can be verified without writing a specification.
Example kernel

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;
};
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();
}
Initial state:

Task *cur; Context *ctx;

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

```c
runtime() {
    cur ∈ \{0xa7\}, ctx ∈ \{0xa8\}
    save_context();  
    /* 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() {
    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();
}

and kernel is protected
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();
}

cur ∈ \{0xa7, 0xa2\}, ctx ∈ \{0xa8, 0xa3\}

user code
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 ∈ \{0xa7, 0xa2\}, ctx ∈ \{0xa8, 0xa3\}

and kernel is protected
Example in-context analysis

Initial state:

Task \*cur; Context \*ctx;

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

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

and kernel is protected
Example in-context analysis

Initial state:

Task *cur; Context *ctx;

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

Cur \( \in \{0xa7, 0xa2\} \), \( \text{ctx} \in \{0xa8, 0xa3\} \)
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 ∈ \{0xa7, 0xa2\}, ctx ∈ \{0xa8, 0xa3\}

and kernel is protected
Example in-context analysis

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

Abstractions we use:

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

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

**Key idea**

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

We summarize **task data** using **types**.
Types refined with **predicates**.

\[
\text{type Flags} = \text{Int8 with (self \& PRIVILEGED) == 0}
\]

\[
\text{type Context} = \text{struct }
\begin{align*}
&\text{Int8 pc; Int8 sp; } \\
&\text{Flags flags; }
\end{align*}
\]

\[
\text{type Task} = \text{struct }
\begin{align*}
&\text{Memory_table* mem_table; } \\
&\text{Context ctx; } \\
&\text{Task* next; }
\end{align*}
\]

Each type \( t \) has an **interpretation** \( \mathcal{L}(t) \) as a set of values.

E.g.

\[
\mathcal{L}(\text{Task}) = \{0xa2, 0xa7\}
\]

\[
\mathcal{L}(\text{Flags}) = \{x \mid x \& \text{PRIVILEGED} = 0\}
\]
Type system: a few examples

Types refined with predicates.

define Flags = Int8 with
    (self \& PRIVILEGED) = 0

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

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

Each type t has an interpretation \( |t| \) as a set of values.
E.g.

\[ |Task*| = \{ 0xa2, 0xa7 \} \]

\[ |Flags| = \{ x \mid x \& PRIVILEGED = 0 \} \]
Example parameterized analysis

Initial state:

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:

Task *cur; Context *ctx;

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

\[\{\text{Task}\}*\} = \{0xa2, 0xa7\}
\[\{\text{Context}\}*\} = \{0xa3, 0xa8\}
Example parameterized analysis

Initial state:

Task *cur; Context *ctx;

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

\(\{\text{Task}\} = \{0xa2, 0xa7\}\)
\(\{\text{Context}\} = \{0xa3, 0xa8\}\)
Example parameterized analysis

Initial state:

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:**

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

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

\( \{\text{Task}\} = \{0xa2, 0xa7\} \)

\( \{\text{Context}\} = \{0xa3, 0xa8\} \)
Example parameterized analysis

Initial state:

Task *cur; Context *ctx;

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

\|Task*\| = \{0xa2, 0xa7\}
\|Context*\| = \{0xa3, 0xa8\}
Example parameterized analysis

**Initial state:**

Task *cur; Context *ctx;

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

\(|\text{Task}^*\) = \{0xa2, 0xa7\}
\(|\text{Context}^*\) = \{0xa3, 0xa8\}
Differentiated handling of boot and runtime code

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

Based on this, we

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

Case study 1: ASTERIOS

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

2 versions

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

Specific = restriction on stack sizes

<table>
<thead>
<tr>
<th></th>
<th>Generic annotations</th>
<th>Specific annotations</th>
</tr>
</thead>
<tbody>
<tr>
<td># shape annotations</td>
<td>generated manual</td>
<td>1057</td>
</tr>
<tr>
<td></td>
<td>57 (5.11%)</td>
<td>58 (5.20%)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Kernel version</th>
<th>BETA</th>
<th>v1</th>
<th>BETA</th>
<th>v1</th>
</tr>
</thead>
<tbody>
<tr>
<td>invariant computation status</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>time (s)</td>
<td>647</td>
<td>417</td>
<td>599</td>
<td>406</td>
</tr>
<tr>
<td># alarms in runtime</td>
<td>1 true error</td>
<td>1 false alarm</td>
<td>1 true error</td>
<td>0 ✓</td>
</tr>
<tr>
<td>user tasks checking status</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>time (s)</td>
<td>32</td>
<td>29</td>
<td>31</td>
<td>30</td>
</tr>
</tbody>
</table>

Proves APE? N/A ▼ N/A ✓

Proved APE and ARTE in 430 s.
58 lines of annotations.
Experimental evaluation: Genericity

Case study 2: EducRTOS

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

Proved APE and ARTE on 96 variants. Varying parameters:

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

Verification time: from 1.6 s to 73 s.

14 lines of annotations.
Experimental evaluation: Automation and Scalability

We compare

- fully automated in-context analysis vs parameterized analysis (12 lines of annotations)

- for a simple variant of EducRTOS

- with varying numbers of tasks.

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

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

- from the executable
- with a low annotation burden.

We address existing limitations:

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

⇒ Key enabler for more automated verification of larger systems.

https://binsec.github.io/