Lightweight Shape Analysis based on Physical Types

Olivier Nicole[1 2], M. Lemerre[1], X. Rival[2]

[1]CEA LIST/Université Paris-Saclay

[2]CNRS/ENS/INRIA/PSL University

The need for automated memory analysis

Memory safety errors are the most pervasive and most severe cybersecurity issue today

"*~70% of the vulnerabilities addressed through a security update each year continue to be memory safety issues*"

– M. Miller (Microsoft Security Response Center), Blue Hat 2019

"63% *of 2019's exploited 0-day vulnerabilities fall under memory corruption*"

– Google's project zero, "A Year in Review of 0-days Used In-The-Wild in 2019"

The need for automated memory analysis

Memory safety errors are the most pervasive and most severe cybersecurity issue today

"~70% *of the vulnerabilities addressed through a security update each year continue to be memory safety issues*"

– M. Miller (Microsoft Security Response Center), Blue Hat 2019

"63% of 2019's *exploited 0-day vulnerabilities fall under memory corruption*"

– Google's project zero, "A Year in Review of 0-days Used In-The-Wild in 2019"

Structural invariants on memory are the backbone of the proof in systems programs

"*Much of the kernel-call code is directed at maintaining [data-structure] invariants*"

– Walker et al., *Specification and Verification of the UCLA Unix Security Kernel*, 1980

"*There are four main categories of invariants in our proof: 1. low-level memory invariants, 2. typing invariants, 3. data structure invariants, and 4. algorithmic invariants. [...] 80% of the effort [...] went into establishing invariants.*"

– Klein et al., *Comprehensive Formal Verification of an OS Microkernel*, 2015

➜ Need for a practical automated memory analysis for low-level languages.

Families of automated memory analyses

	Analysis efficiency	Automation & Ease of setup	Proves memory safety	Proves structural invariants	Low-level code (binary)
Pointer analyses	++	++	−−	−−	∼
Shape analyses	−	∼	++	++	∼
Our approach: Type-based analysis	+	+	+	+	++

Our goal

Analysis efficiency + Easy setup + Spatial Memory Safety + Structural Constraints
On C and machine code

Our approach: type-based abstract interpretation

Our approach

An *abstract interpretation* that

- simultaneously *relies on* and *establishes type safety*
- using a *dedicated type system* designed for abstract interpretation.

Our approach: type-based abstract interpretation

Our approach

An *abstract interpretation* that

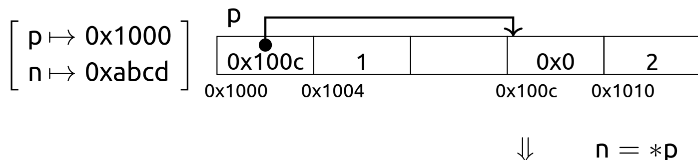
- simultaneously *relies on* and *establishes type safety*
- using a *dedicated type system* designed for abstract interpretation.

Existing works on automated type-based static analysis either are:

- Static analyses that *assume* type safety;
- Or *syntactic and decidable type inference* algorithms that are insufficiently precise for systems code.

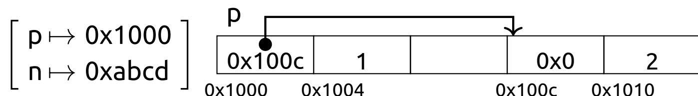
Overview

We start from standard & untyped concrete semantics for machine code...

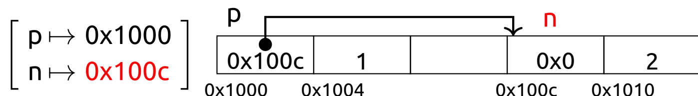


Overview

We start from standard & untyped concrete semantics for machine code...



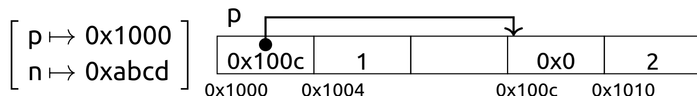
$\Downarrow \qquad n = *p$



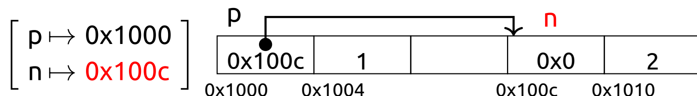
$\Downarrow \qquad *n = p$

Overview

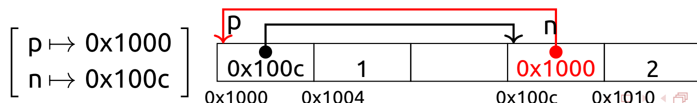
We start from standard & untyped concrete semantics for machine code...



$\Downarrow$ $n = *p$



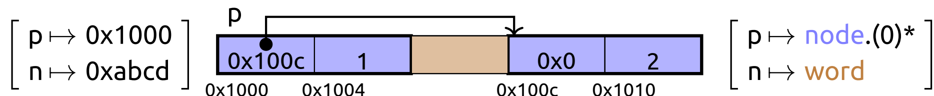
$\Downarrow$ $*n = p$



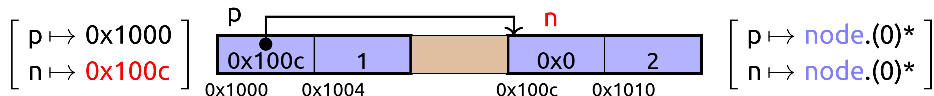
Overview

We add types to tag the memory and the variables...

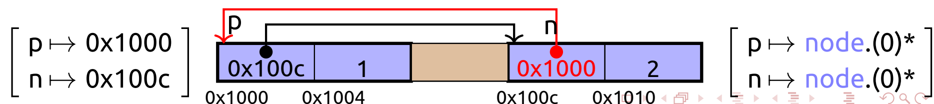
type data = word with self <= 10;   type node = struct { node.(0)*; data }



$\Downarrow$ $n = *p$



$\Downarrow$ $*n = p$



Overview

Type domain is just type of the variables + invariant that states are well-typed

```
type data = word with self <= 10;
type node = struct { node.(0)*; data }
```

$$\begin{bmatrix} p \mapsto \text{node.(0)*} \\ n \mapsto \text{word} \end{bmatrix}$$

$$\Downarrow \quad n = *p$$

$$\begin{bmatrix} p \mapsto \text{node.(0)*} \\ n \mapsto \text{node.(0)*} \end{bmatrix}$$

$$\Downarrow \quad *n = p$$

$$\begin{bmatrix} p \mapsto \text{node.(0)*} \\ n \mapsto \text{node.(0)*} \end{bmatrix}$$

Overview

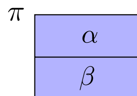
The full analysis adds numerical constraints and points-to predicates

type data = word with self <= 10; type node = struct { node.(0)*; data }

$\pi \neq 0$ $\begin{bmatrix} \mathsf{p} \mapsto \pi \\ \mathsf{n} \mapsto \eta \end{bmatrix}$ $\begin{bmatrix} \mathsf{p} \mapsto \text{node.(0)*} \\ \mathsf{n} \mapsto \text{word} \end{bmatrix}$

$\Downarrow$ $\mathsf{n} = *\mathsf{p}$

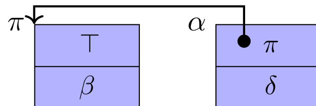
$\pi \neq 0 \quad \beta \leq 10$ $\begin{bmatrix} \mathsf{p} \mapsto \pi \\ \mathsf{n} \mapsto \alpha \end{bmatrix}$



$\begin{bmatrix} \mathsf{p} \mapsto \text{node.(0)*} \\ \mathsf{n} \mapsto \text{node.(0)*} \end{bmatrix}$

$\Downarrow$ $*\mathsf{n} = \mathsf{p}$

$\pi \neq 0 \quad \beta \leq 10$
$\alpha \neq 0 \quad \delta \leq 10$ $\begin{bmatrix} \mathsf{p} \mapsto \pi \\ \mathsf{n} \mapsto \alpha \end{bmatrix}$



$\begin{bmatrix} \mathsf{p} \mapsto \text{node.(0)*} \\ \mathsf{n} \mapsto \text{node.(0)*} \end{bmatrix}$

Outline and contributions

1. Typed concrete semantics
    - Structural invariants expressed by types
2. Abstract domains
    - Type-based domain
    - Points-to predicates domain
3. Experiments on shape benchmark and binary-level microkernel verification
    - Existing shape benchmarks (C + binary)
        - Verification of memory safety + preservation of light structural invariants
    - Verification of absence of privilege escalation
        - Of a full commercial microkernel
        - From its binary executable
4. Conclusion

Outline

1. Typed concrete semantics
   - Structural invariants expressed by types
2. Abstract domains
   - Type-based domain
   - Points-to predicates domain
3. Experiments on shape benchmark and binary-level microkernel verification
   - Existing shape benchmarks (C + binary)
     - Verification of memory safety + preservation of light structural invariants
   - Verification of absence of privilege escalation
     - Of a full commercial microkernel
     - From its binary executable
4. Conclusion

Types

$$
\begin{array}{rcll}
\mathbb{T}ypes \ni t & ::= & \mathsf{word}_k & \text{(base type of size } k \text{ bytes)} \\
& | & \mathsf{n} & \text{(named type with type name } \mathsf{n} \in \mathcal{N}ames\text{)} \\
& | & t_o* & \text{(possibly null pointer)} \\
& | & \mathsf{struct}\{t; t\} & \text{(record type)} \\
& | & t[s] & \text{(array type)} \\
& | & t \ \mathsf{with}\ pred & \text{(type with a refinement predicate)} \\
\mathbb{T}ypes_O \ni t_o & ::= & t.(o) & \text{(type with offset } o \in \mathbb{N}\text{)}
\end{array}
$$

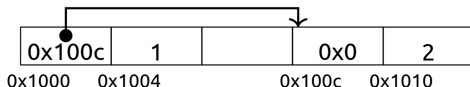
$$\mathcal{D}efinitions \in \mathcal{N}ames \to \mathbb{T}ypes$$

type data = word_4 with self $\leq$ 10;

type node = struct { node.(0)*; data }

Heaps and labellings

$h \ni \mathbb{H}eap = \mathbb{A}ddress \to \mathbb{V}alue$

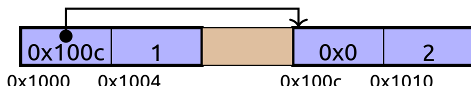


- Byte-level reasoning:

$$h[0x1000 - 0x1003] = 0x100c \iff h[0x1000] = 0x0c \land h[0x1001] = 0x10 \land \ldots$$

Heaps and labellings

$$h \ni \mathbb{H}eap = \mathbb{A}ddress \to \mathbb{V}alue$$
$$\mathcal{L} \ni \mathbb{L}abellings = \mathbb{A}ddress \to \mathbb{T}ypes_{\mathbb{O}}$$



- Byte-level reasoning:

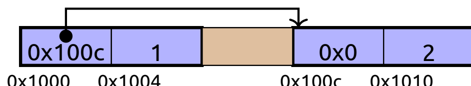
$$h[\mathtt{0x1000} - \mathtt{0x1003}] = \mathtt{0x100c} \iff$$
$$h[\mathtt{0x1000}] = \mathtt{0x0c} \wedge h[\mathtt{0x1001}] = \mathtt{0x10} \wedge \ldots$$

- Labellings are whole and contiguous:

$$\mathcal{L}[\mathtt{0x1000} - \mathtt{0x1007}] = \mathtt{node} \iff$$
$$\mathcal{L}[\mathtt{0x1000}] = \mathtt{node}.(0) \wedge \ldots \wedge \mathcal{L}[\mathtt{0x1007}] = \mathtt{node}.(7)$$

Heaps and labellings

$$h \ni \mathbb{H}eap = \mathbb{A}ddress \to \mathbb{V}alue$$
$$\mathcal{L} \ni \mathbb{L}abellings = \mathbb{A}ddress \to \mathbb{T}ypes_{\mathrm{O}}$$



- Byte-level reasoning:

$$h[\mathtt{0x1000} - \mathtt{0x1003}] = \mathtt{0x100c} \iff$$
$$h[\mathtt{0x1000}] = \mathtt{0x0c} \wedge h[\mathtt{0x1001}] = \mathtt{0x10} \wedge \ldots$$

- Labellings are whole and contiguous:

$$\mathcal{L}[\mathtt{0x1000} - \mathtt{0x1007}] = \mathsf{node} \iff$$
$$\mathcal{L}[\mathtt{0x1000}] = \mathsf{node}.(0) \wedge \ldots \wedge \mathcal{L}[\mathtt{0x1007}] = \mathsf{node}.(7)$$

- This allows safe pointer arithmetics:

Safe Pointer Arithmetics

$$\frac{e_1 : \mathtt{t}.(o)* \qquad e_1 \Downarrow v_1 \neq 0 \qquad 0 \leq o + i < \mathsf{size}(\mathtt{t})}{e + i : \mathtt{t}.(o + i)*}$$

The lattice of types with offset, subtyping, aliasing

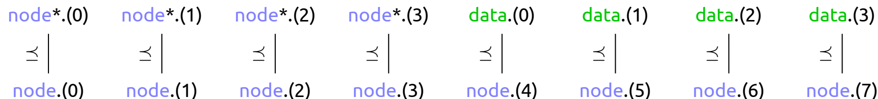
type node = struct { node.(0)*; data }

node*.(0)	node*.(1)	node*.(2)	node*.(3)	data.(0)	data.(1)	data.(2)	data.(3)

node.(0)	node.(1)	node.(2)	node.(3)	node.(4)	node.(5)	node.(6)	node.(7)

The lattice of types with offset, subtyping, aliasing

type node = struct { node.(0)*; data }



The lattice of types with offset, subtyping, aliasing

type node = struct { node.(0)*; data }

node*.(0)	node*.(1)	node*.(2)	node*.(3)	data.(0)	data.(1)	data.(2)	data.(3)
$\preceq$	$\preceq$	$\preceq$	$\preceq$	$\preceq$	$\preceq$	$\preceq$	$\preceq$
node.(0)	node.(1)	node.(2)	node.(3)	node.(4)	node.(5)	node.(6)	node.(7)

Theorem: Containment relation $\preceq \in \mathbb{T}ypes_{\mathbb{O}} \times \mathbb{T}ypes_{\mathbb{O}}$ is an order relation.

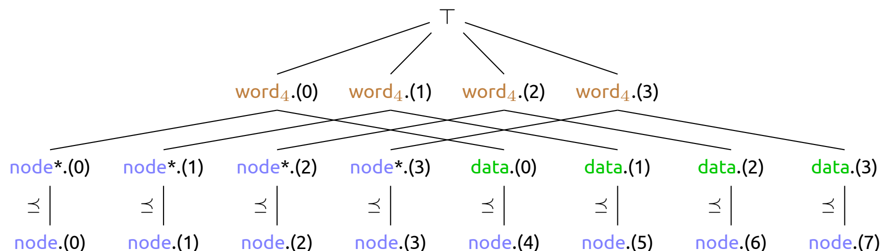
Corollary:

Subtyping

$$\frac{e : \mathsf{t}.(o)* \qquad \mathsf{t}.(o) \preceq \mathsf{u}.(i)}{e : \mathsf{u}.(i)*}$$

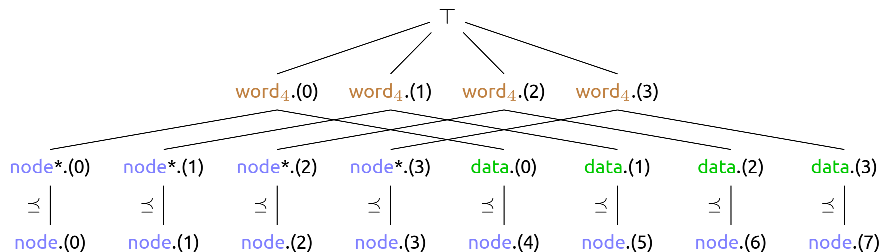
The lattice of types with offset, subtyping, aliasing

type data = word_4 with self $\leq$ 10;
type node = struct { node.(0)*; data }



The lattice of types with offset, subtyping, aliasing

type data = word_4 with self $\leq$ 10;
type node = struct { node.(0)*; data }



Theorem: The semi-lattice $(\mathbb{T}ypes_{\mathbb{O}}, \preceq)$ is a tree

Corollary:

$$\text{No-Alias}\ \frac{e_1 : \mathsf{t}.(o)* \quad e_2 : \mathsf{u}.(i)* \quad e_1 \Downarrow v_1 \quad e_2 \Downarrow v_2 \quad \mathsf{t}.(o) \not\preceq \mathsf{u}.(i) \quad \mathsf{u}.(i) \not\preceq \mathsf{t}.(o)}{v_1 \neq v_2}$$

Interpretation of a type

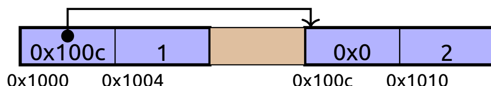
Using labellings $\mathcal{L}$, types can be interpreted as a set of values.

$$
\begin{aligned}
(\!| \cdot |\!)_{\mathcal{L}} &: \mathbb{T}ypes \to \mathcal{P}(\mathbb{V}) \\
(\!| \mathsf{word}_n |\!)_{\mathcal{L}} &= \mathbb{V}_n \\
(\!| t \mathsf{\ with\ } p |\!)_{\mathcal{L}} &= \{ v \in (\!| t |\!)_{\mathcal{L}} \mid \mathrm{eval}(p, v) = \mathbf{true} \} \\
&\vdots \qquad\qquad \vdots \\
(\!| t_o * |\!)_{\mathcal{L}} &= \{ a \in \mathbb{A} \mid \mathcal{L}(a) \preceq t_o \} \cup \{0\}
\end{aligned}
$$

Interpretation of a type

Using labellings $\mathcal{L}$, types can be interpreted as a set of values.

$$\begin{aligned}
(\!|\cdot|\!)_{\mathcal{L}} &: \mathbb{T}ypes \to \mathcal{P}(\mathbb{V}) \\
(\!|\,\mathsf{word}_n\,|\!)_{\mathcal{L}} &= \mathbb{V}_n \\
(\!|\,t \;\mathsf{with}\; p\,|\!)_{\mathcal{L}} &= \{v \in (\!|\,t\,|\!)_{\mathcal{L}} \mid \mathsf{eval}(p, v) = \mathbf{true}\} \\
&\vdots \\
(\!|\,t_o *\,|\!)_{\mathcal{L}} &= \{a \in \mathbb{A} \mid \mathcal{L}(a) \preceq t_o\} \cup \{0\}
\end{aligned}$$



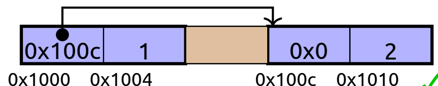
$$\begin{aligned}
(\!|\,\mathsf{node}.(0)* \;\mathsf{with\ self} \neq 0\,|\!)_{\mathcal{L}} &= \{v \in \{0, \mathsf{0x1000}, \mathsf{0x100c}\} \mid v \neq 0\} \\
&= \{\mathsf{0x1000}, \mathsf{0x100c}\}
\end{aligned}$$

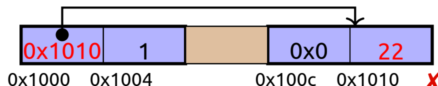
Putting everything together: well-typed states

Definition (Well-typed states)

Addresses of type t should contain a value in $(\!| t |\!)_{\mathcal{L}}$:

$$\mathcal{L}[a..a + \text{size}(t)] = t \Rightarrow h[a..a + \text{size}(t)] \in (\!| t |\!)_{\mathcal{L}}$$



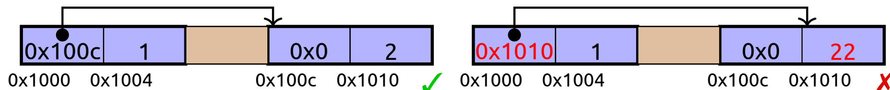


Putting everything together: well-typed states

Definition (Well-typed states)

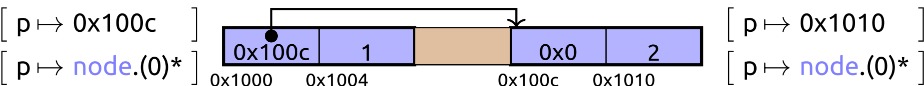
Addresses of type t should contain a value in $(\!| t |\!)_{\mathcal{L}}$:

$$\mathcal{L}[a..a + \text{size}(t)] = t \Rightarrow h[a..a + \text{size}(t)] \in (\!| t |\!)_{\mathcal{L}}$$



Variables of type t should contain a value in $(\!| t |\!)_{\mathcal{L}}$:

$$\Gamma[x] = t \Rightarrow \sigma[x] \in (\!| t |\!)_{\mathcal{L}}$$



Outline

1. Typed concrete semantics
   - Structural invariants expressed by types
2. Abstract domains
   - Type-based domain
   - Points-to predicates domain
3. Experiments on shape benchmark and binary-level microkernel verification
   - Existing shape benchmarks (C + binary)
     - Verification of memory safety + preservation of light structural invariants
   - Verification of absence of privilege escalation
     - Of a full commercial microkernel
     - From its binary executable
4. Conclusion

The type abstract domain

$$\Gamma^\sharp \triangleq \mathbb{X} \to \mathbb{T}ypes$$

$$\gamma_{\Gamma^\sharp}(\Gamma) = \{\text{erase_type}(s) \mid s \text{ is a well-typed state with variables typed using } \Gamma\}$$

The type abstract domain

$$\Gamma^\sharp \triangleq \mathbb{X} \to \mathbb{T}ypes$$

$$\gamma_{\Gamma^\sharp}(\Gamma) = \{\text{erase_type}(s) \mid s \text{ is a well-typed state with variables typed using } \Gamma\}$$

Example

type data = word with self <= 10;   type foo = data;   type bar = data;

$$\gamma_{\Gamma^\sharp}\left(\begin{bmatrix} \mathsf{p} \mapsto \mathsf{foo}.(0)* \\ \mathsf{q} \mapsto \mathsf{foo}.(0)* \\ \mathsf{r} \mapsto \mathsf{bar}.(0)* \end{bmatrix}\right) =$$

{ [p,q → 0x1000: 3 | r → 0x1004: 7 | 0x1008: 17], [r → 0x1000: 4 | p,q → 0x1004: 2 | 0x1008: 31], [r → 0x1000: 5 | p → 0x1004: 3 | q → 0x1008: 2] ... }

The base analysis: combination with a numerical domain

Reduced product with numerical constraints attached to program variables.

```
type data = word with self <= 10;
```

$$\begin{bmatrix} \mathsf{p} \mapsto \pi \\ \mathsf{n} \mapsto \eta \end{bmatrix}$$

```
type node = struct { node.(0)*; data }
```

$$\begin{bmatrix} \mathsf{p} \mapsto \mathsf{node.(0)*} \\ \mathsf{n} \mapsto \mathsf{word} \end{bmatrix}$$

The base analysis: combination with a numerical domain

Reduced product with numerical constraints attached to program variables.

```
type data = word with self <= 10;        type node = struct { node.(0)*; data }
```

$$\begin{bmatrix} \mathsf{p} \mapsto \pi \\ \mathsf{n} \mapsto \eta \end{bmatrix} \qquad \begin{bmatrix} \mathsf{p} \mapsto \mathsf{node.(0)*} \\ \mathsf{n} \mapsto \mathsf{word} \end{bmatrix}$$

$$\Downarrow \quad \mathsf{assume}(\mathsf{p} \neq 0) \ ✓$$

$$\pi \neq 0 \quad \begin{bmatrix} \mathsf{p} \mapsto \pi \\ \mathsf{n} \mapsto \eta \end{bmatrix} \qquad \begin{bmatrix} \mathsf{p} \mapsto \mathsf{node.(0)*} \\ \mathsf{n} \mapsto \mathsf{word} \end{bmatrix}$$

The base analysis: combination with a numerical domain

Reduced product with numerical constraints attached to program variables.

type data = word with self <= 10;

type node = struct { node.(0)*; data }

$$\begin{bmatrix} \mathsf{p} \mapsto \pi \\ \mathsf{n} \mapsto \eta \end{bmatrix} \qquad \begin{bmatrix} \mathsf{p} \mapsto \text{node.(0)*} \\ \mathsf{n} \mapsto \text{word} \end{bmatrix}$$

$\Downarrow$ assume$(\mathsf{p} \neq 0)$ ✓

$$\pi \neq 0 \quad \begin{bmatrix} \mathsf{p} \mapsto \pi \\ \mathsf{n} \mapsto \eta \end{bmatrix} \qquad \begin{bmatrix} \mathsf{p} \mapsto \text{node.(0)*} \\ \mathsf{n} \mapsto \text{word} \end{bmatrix}$$

$\Downarrow$ $\mathsf{n} = *\mathsf{p}$ ✓

$$\pi \neq 0 \quad \begin{bmatrix} \mathsf{p} \mapsto \pi \\ \mathsf{n} \mapsto \alpha \end{bmatrix} \qquad \begin{bmatrix} \mathsf{p} \mapsto \text{node.(0)*} \\ \mathsf{n} \mapsto \text{node.(0)*} \end{bmatrix}$$

The base analysis: combination with a numerical domain

Reduced product with numerical constraints attached to program variables.

```
type data = word with self <= 10;
```

```
type node = struct { node.(0)*; data }
```

$$\begin{bmatrix} \mathsf{p} \mapsto \pi \\ \mathsf{n} \mapsto \eta \end{bmatrix} \qquad \begin{bmatrix} \mathsf{p} \mapsto \mathsf{node.(0)*} \\ \mathsf{n} \mapsto \mathsf{word} \end{bmatrix}$$

$\Downarrow$ $\mathsf{assume}(\mathsf{p} \neq 0)$ ✓

$$\pi \neq 0 \quad \begin{bmatrix} \mathsf{p} \mapsto \pi \\ \mathsf{n} \mapsto \eta \end{bmatrix} \qquad \begin{bmatrix} \mathsf{p} \mapsto \mathsf{node.(0)*} \\ \mathsf{n} \mapsto \mathsf{word} \end{bmatrix}$$

$\Downarrow$ $\mathsf{n} = *\mathsf{p}$ ✓

$$\pi \neq 0 \quad \begin{bmatrix} \mathsf{p} \mapsto \pi \\ \mathsf{n} \mapsto \alpha \end{bmatrix} \qquad \begin{bmatrix} \mathsf{p} \mapsto \mathsf{node.(0)*} \\ \mathsf{n} \mapsto \mathsf{node.(0)*} \end{bmatrix}$$

A storeless abstraction is often not enough

Add points-to predicates to retain information about the heap

type data = word with self <= 10;

type node = struct { node.(0)*; data }

$\pi \neq 0$ $\begin{bmatrix} \mathsf{p} \mapsto \pi \\ \mathsf{n} \mapsto \eta \end{bmatrix}$

$\begin{bmatrix} \mathsf{p} \mapsto \mathsf{node.(0)*} \\ \mathsf{n} \mapsto \mathsf{word} \end{bmatrix}$

$\Downarrow$ $\mathsf{assume}(*\mathsf{p}) \neq 0$ ✓

A storeless abstraction is often not enough

Add points-to predicates to retain information about the heap

```
type data = word with self <= 10;
```

```
type node = struct { node.(0)*; data }
```

$\pi \neq 0 \quad \begin{bmatrix} \mathsf{p} \mapsto \pi \\ \mathsf{n} \mapsto \eta \end{bmatrix}$

$\begin{bmatrix} \mathsf{p} \mapsto \text{node.(0)*} \\ \mathsf{n} \mapsto \text{word} \end{bmatrix}$

$\Downarrow \quad \mathsf{assume}(*\mathsf{p}) \neq 0$ ✓

$\pi \neq 0 \quad \begin{bmatrix} \mathsf{p} \mapsto \pi \\ \mathsf{n} \mapsto \eta \end{bmatrix}$

$\begin{bmatrix} \mathsf{p} \mapsto \text{node.(0)*} \\ \mathsf{n} \mapsto \text{word} \end{bmatrix}$

$\Downarrow \quad \mathsf{n} = **\mathsf{p}$ ✗

A storeless abstraction is often not enough

Add points-to predicates to retain information about the heap

type data = word with self <= 10;

type node = struct { node.(0)*; data }

$\pi \neq 0 \quad \begin{bmatrix} \mathsf{p} \mapsto \pi \\ \mathsf{n} \mapsto \eta \end{bmatrix}$

$\begin{bmatrix} \mathsf{p} \mapsto \texttt{node.(0)*} \\ \mathsf{n} \mapsto \texttt{word} \end{bmatrix}$

$\Downarrow \quad \mathsf{assume}(*\mathsf{p}) \neq 0$ ✓

$\pi \neq 0 \quad \begin{bmatrix} \mathsf{p} \mapsto \pi \\ \mathsf{n} \mapsto \eta \end{bmatrix}$

$\begin{bmatrix} \mathsf{p} \mapsto \texttt{node.(0)*} \\ \mathsf{n} \mapsto \texttt{word} \end{bmatrix}$

$\Downarrow \quad \mathsf{n} = **\mathsf{p}$ ✗

$\pi \neq 0 \quad \begin{bmatrix} \mathsf{p} \mapsto \pi \\ \mathsf{n} \mapsto \zeta \end{bmatrix}$

$\begin{bmatrix} \mathsf{p} \mapsto \texttt{node.(0)*} \\ \mathsf{n} \mapsto \texttt{node.(0)*} \end{bmatrix}$

A storeless abstraction is often not enough

Add points-to predicates to retain information about the heap

type data = word with self <= 10;

type node = struct { node.(0)*; data }

$\pi \neq 0$ $\begin{bmatrix} \mathsf{p} \mapsto \pi \\ \mathsf{n} \mapsto \eta \end{bmatrix}$

$\begin{bmatrix} \mathsf{p} \mapsto \mathsf{node.(0)*} \\ \mathsf{n} \mapsto \mathsf{word} \end{bmatrix}$

$\Downarrow$ $\mathsf{assume}(*\mathsf{p}) \neq 0$ ✓

A storeless abstraction is often not enough

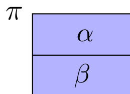
Add points-to predicates to retain information about the heap

type data = word with self <= 10;　　type node = struct { node.(0)*; data }

$\pi \neq 0$　$\begin{bmatrix} \mathsf{p} \mapsto \pi \\ \mathsf{n} \mapsto \eta \end{bmatrix}$　　$\begin{bmatrix} \mathsf{p} \mapsto \text{node.(0)*} \\ \mathsf{n} \mapsto \text{word} \end{bmatrix}$

$\Downarrow$　assume$(*\mathsf{p}) \neq 0$ ✓

$\alpha \neq 0 \quad \pi \neq 0$
$\beta \leq 10$　$\begin{bmatrix} \mathsf{p} \mapsto \pi \\ \mathsf{n} \mapsto \eta \end{bmatrix}$



$\begin{bmatrix} \mathsf{p} \mapsto \text{node.(0)*} \\ \mathsf{n} \mapsto \text{word} \end{bmatrix}$

$\Downarrow$　$\mathsf{n} = **\mathsf{p}$ ✓

A storeless abstraction is often not enough

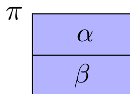
Add points-to predicates to retain information about the heap

type data = word with self <= 10;   type node = struct { node.(0)*; data }

$\pi \neq 0$ $\begin{bmatrix} \mathsf{p} \mapsto \pi \\ \mathsf{n} \mapsto \eta \end{bmatrix}$ $\begin{bmatrix} \mathsf{p} \mapsto \text{node.(0)*} \\ \mathsf{n} \mapsto \text{word} \end{bmatrix}$

$\Downarrow$ $\mathsf{assume}(*\mathsf{p}) \neq 0$ ✓

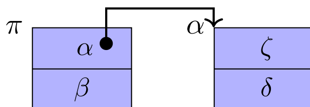
$\alpha \neq 0 \quad \pi \neq 0$
$\beta \leq 10$ $\begin{bmatrix} \mathsf{p} \mapsto \pi \\ \mathsf{n} \mapsto \eta \end{bmatrix}$



$\begin{bmatrix} \mathsf{p} \mapsto \text{node.(0)*} \\ \mathsf{n} \mapsto \text{word} \end{bmatrix}$

$\Downarrow$ $\mathsf{n} = **\mathsf{p}$ ✓

$\alpha \neq 0 \quad \pi \neq 0$
$\beta \leq 10 \quad \delta \leq 10$ $\begin{bmatrix} \mathsf{p} \mapsto \pi \\ \mathsf{n} \mapsto \zeta \end{bmatrix}$



$\begin{bmatrix} \mathsf{p} \mapsto \text{node.(0)*} \\ \mathsf{n} \mapsto \text{node.(0)*} \end{bmatrix}$

Pragmatic extensions: exemple for array handling

```
type task = struct { ... }

type kernel_interface = struct {
  (int with self = nb_tasks);
  task[nb_tasks].(0)*;
}
```

The type system is easily extendable.
Extensions for handling arrays:

- Array types;
- Global parameters (nb_tasks)
- Variables offsets in pointers

Example (Abstract state)

$$\pi \neq 0 \qquad \alpha \leq \mathsf{nb_tasks} * \mathsf{size}(\mathsf{task}) \qquad [\ \mathsf{p} \mapsto \pi\ ] \qquad [\ \mathsf{p} \mapsto \mathsf{task[nb_tasks]}.(\alpha)*\ ]$$

Outline

1. Typed concrete semantics
   - Structural invariants expressed by types
2. Abstract domains
   - Type-based domain
   - Points-to predicates domain
3. Experiments on shape benchmark and binary-level microkernel verification
   - Existing shape benchmarks (C + binary)
     - Verification of memory safety + preservation of light structural invariants
   - Verification of absence of privilege escalation
     - Of a full commercial microkernel
     - From its binary executable
4. Conclusion

Experimental setups

- We implemented:
  - Abstract domains in the Codex library for abstract interpretation
  - Two static analyses:
    - Frama-C/Codex (for C)
    - Binsec/Codex (for machine code)

Experimental setups

- We implemented:
  - Abstract domains in the Codex library for abstract interpretation
  - Two static analyses:
    - Frama-C/Codex (for C)
    - Binsec/Codex (for machine code)
- We experimented on:
  1. Challenging, existing shape benchmarks
     - Verification goal: memory safety and preservation of structural invariants
     - Complex data structure manipulations
  2. A full industrial embedded microkernel (no access to the source)
     - Verification goals: absence of privilege escalation and absence of RTE
     - System program relying on low-level structural invariants

Experimental setups

- We implemented:
  - Abstract domains in the Codex library for abstract interpretation
  - Two static analyses:
    - Frama-C/Codex (for C)
    - Binsec/Codex (for machine code)
- We experimented on:
  1. Challenging, existing shape benchmarks
     - Verification goal: memory safety and preservation of structural invariants
     - Complex data structure manipulations
  2. A full industrial embedded microkernel (no access to the source)
     - Verification goals: absence of privilege escalation and absence of RTE
     - System program relying on low-level structural invariants
- We evaluated the:
  - Performance of the analysis (analysis time)
  - Precision of the analysis (number of alarms)
  - Ease of setup and automation (number of annotations)

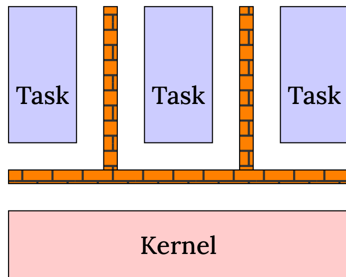
Evaluation on shape benchmarks

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	$\mapsto$	$\not\mapsto$
psll-bsort			0	25	0.30	0	22	0.41	0	3	0.25	0	3	0.26	0	3	0.29	0	3
psll-reverse	10	0	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
gdll-findmin			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	25	5	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
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
bsplay-find			1	81	0.53	0	18	0.25	0	7	0.23	0	7	0.23	0	7	0.23	0	7
bsplay-delete	22	1	1	95	0.72	0	38	0.45	0	11	0.44	0	10	0.44	0	10	0.44	0	10
bsplay-insert			1	101	0.57	0	18	0.25	0	7	0.25	0	7	0.25	0	7	0.25	0	7
graph-nodelisttrav			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	23	0	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			1	11	0.31	0	24	0.07	0	6	0.09	0	0	0.08	0	0	0.07	0	0
uf-merge	33	3	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 (max. 34)						30	13		30	16		30	21		30	21		30	21

Verification of the Asterios embedded real-time kernel

Asterios [1]

- Industrial real-time microkernel
- Version: ARM quad-core port
- 329 functions, ~10,000 instructions



2 versions

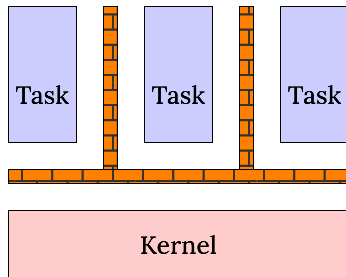
- beta: **1 discovered vulnerability**
- v1 : verified APE and ARTE
  - Analysis time: 430s
  - 58 lines of manual annotations.

[1] No Crash, No Exploit: Automated Verification of Embedded Kernels.
O. Nicole, M. Lemerre, S. Bardin, X. Rival. RTAS 2021

Verification of the Asterios embedded real-time kernel

Asterios [1]

- Industrial real-time microkernel
- Version: ARM quad-core port
- 329 functions, ~10,000 instructions



2 versions

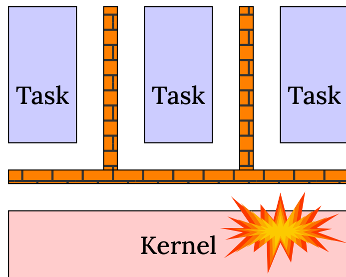
- beta: **1 discovered vulnerability**
- v1 : verified APE and ARTE
  - Analysis time: 430s
  - 58 lines of manual annotations.

[1] No Crash, No Exploit: Automated Verification of Embedded Kernels.
O. Nicole, M. Lemerre, S. Bardin, X. Rival. RTAS 2021

Verification of the Asterios embedded real-time kernel

Asterios [1]

- Industrial real-time microkernel
- Version: ARM quad-core port
- 329 functions, ~10,000 instructions



2 versions

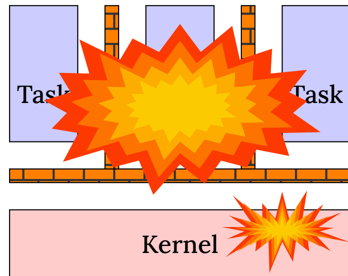
- beta: **1 discovered vulnerability**
- v1 : verified APE and ARTE
  - Analysis time: 430s
  - 58 lines of manual annotations.

[1] No Crash, No Exploit: Automated Verification of Embedded Kernels.
O. Nicole, M. Lemerre, S. Bardin, X. Rival. RTAS 2021

Verification of the Asterios embedded real-time kernel

Asterios [1]

- Industrial real-time microkernel
- Version: ARM quad-core port
- 329 functions, ~10,000 instructions



2 versions

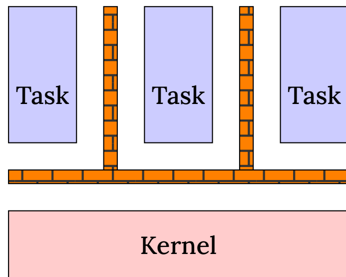
- beta: **1 discovered vulnerability**
- v1 : verified APE and ARTE
  - Analysis time: 430s
  - 58 lines of manual annotations.

[1] No Crash, No Exploit: Automated Verification of Embedded Kernels.
O. Nicole, M. Lemerre, S. Bardin, X. Rival. RTAS 2021

Verification of the Asterios embedded real-time kernel

Asterios [1]

- Industrial real-time microkernel
- Version: ARM quad-core port
- 329 functions, ~10,000 instructions



2 versions

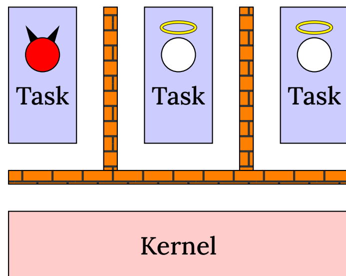
- beta: **1 discovered vulnerability**
- v1 : verified APE and ARTE
  - Analysis time: 430s
  - 58 lines of manual annotations.

[1] No Crash, No Exploit: Automated Verification of Embedded Kernels.
O. Nicole, M. Lemerre, S. Bardin, X. Rival. RTAS 2021

Verification of the Asterios embedded real-time kernel

Asterios [1]

- Industrial real-time microkernel
- Version: ARM quad-core port
- 329 functions, ~10,000 instructions



2 versions

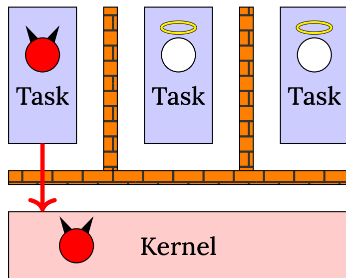
- beta: **1 discovered vulnerability**
- v1 : verified APE and ARTE
  - Analysis time: 430s
  - 58 lines of manual annotations.

[1] No Crash, No Exploit: Automated Verification of Embedded Kernels.
O. Nicole, M. Lemerre, S. Bardin, X. Rival. RTAS 2021

Verification of the Asterios embedded real-time kernel

Asterios [1]

- Industrial real-time microkernel
- Version: ARM quad-core port
- 329 functions, ~10,000 instructions



2 versions

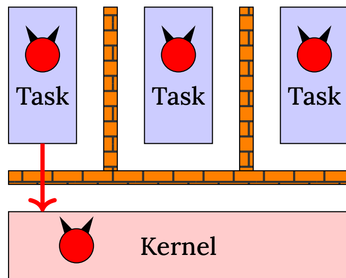
- beta: **1 discovered vulnerability**
- v1 : verified APE and ARTE
  - Analysis time: 430s
  - 58 lines of manual annotations.

[1] No Crash, No Exploit: Automated Verification of Embedded Kernels.
O. Nicole, M. Lemerre, S. Bardin, X. Rival. RTAS 2021

Verification of the Asterios embedded real-time kernel

Asterios [1]

- Industrial real-time microkernel
- Version: ARM quad-core port
- 329 functions, ~10,000 instructions



2 versions

- beta: **1 discovered vulnerability**
- v1 : verified APE and ARTE
  - Analysis time: 430s
  - 58 lines of manual annotations.

[1] No Crash, No Exploit: Automated Verification of Embedded Kernels.
O. Nicole, M. Lemerre, S. Bardin, X. Rival. RTAS 2021

Outline

1. Typed concrete semantics
   - Structural invariants expressed by types
2. Abstract domains
   - Type-based domain
   - Points-to predicates domain
3. Experiments on shape benchmark and binary-level microkernel verification
   - Existing shape benchmarks (C + binary)
     - Verification of memory safety + preservation of light structural invariants
   - Verification of absence of privilege escalation
     - Of a full commercial microkernel
     - From its binary executable
4. Conclusion

Conclusion

Type-based static analysis is:

- Efficient
- Easily automatable, requires only few annotations
- Relatively precise, can prove memory safety and light structural invariants
- Working on binary code nearly as well as on C
- A robust basis for a combination with advanced memory analyses