Verification of Programs

Abstract Interpretation

A Bit of Architecture and System and Inline Assembly

Reduced Product with Ghost Variables

Discussion and Conclusion

Proving the Security of Software-Intensive Embedded Systems by Abstract Interpretation

Marc Chevalier

under the wise aegis of Jérôme Feret

ENS

November 27, 2020

Verification of Programs

Abstract Interpretation

A Bit of Architecture and System and Inline Assembly

Reduced Product with Ghost Variables

Discussion and Conclusion

Verification of Programs
Why?
How?
What?
Abstract Interpretation
A Bit of Architecture and System and Inline Assembly
Reduced Product with Ghost Variables
Discussion and Conclusion

Verification of Programs

Abstract Interpretation

A Bit of Architecture and System and Inline Assembly

Reduced Product with Ghost Variables

Discussion and Conclusion

Verification of Programs
Why?
How?
What?
Abstract Interpretation
A Bit of Architecture and System and Inline Assembly
Reduced Product with Ghost Variables
Discussion and Conclusion

Why? — What Happens When Software Fails

Figure 1: Ariane 5, 4th June 1996

Verification of Programs
Why?
How?
What?
Abstract Interpretation
A Bit of Architecture and System and Inline Assembly
Reduced Product with Ghost Variables
Discussion and Conclusion

Why? — What Happens When Software Fails

Figure 2: Ariane 5, 40s later

Verification of Programs
Why?
How?
What?
Abstract Interpretation
A Bit of Architecture and System and Inline Assembly
Reduced Product with Ghost Variables
Discussion and Conclusion

Why? — Cost of Software Failure

Bugs have various annoying consequences:

- Deaths (Patriot MIM-104, Toyota, Therac-25 radiotherapy machines)
- A lot of money:
  - Ariane 5 (payload: $\$370 \times 10^6$)
  - between $\$30 \times 10^9$ and $\$60 \times 10^9$/year in the US (NIST in 2002). How much in 2020…
- Privacy (Heartbleed)
- …

Verification of Programs
Why?
How?
What?
Abstract Interpretation
A Bit of Architecture and System and Inline Assembly
Reduced Product with Ghost Variables
Discussion and Conclusion

How? — What Developers Usually Do

How developers think they can avoid bugs:

- Tests
- High-level/safe language
- Strict code style

Still, Ariane 5 crashed.... “And here, poor fool[s], with all [their] lore, [they] stand no wiser than before”.

How? — Why it is not Enough

But:

- Tests only check one execution trace (even worse if the program is non-deterministic).
- An exception/error result value will still not be handled correctly if it is not supposed to happen.
- Strict code styles allow very explicit errors.

How? — What We Should Do

We need a way to find all bugs!

Finding exactly all (and only) bugs is not computable (Rice's theorem).

We have to accept to have false positive (detecting errors that do not exist): correct (sound) but not complete.

Verification of Programs
Why?
How?
What?
Abstract Interpretation
A Bit of Architecture and System and Inline Assembly
Reduced Product with Ghost Variables
Discussion and Conclusion

What? — Properties

What is an error? What do we want to prove? Well, it depends on the context:

- no undefined behavior,
- specification on the result,
- bounded execution time (e.g. for real-time systems),
- constant execution time/path (e.g. for cryptographic software),
- no unwanted data flow (private to public),
- termination (e.g. for usual applications),
- non-termination (e.g. for servers, OSes…),
- …

There is no small bugs: most crashes can be exploited to execute arbitrary code*.

*Kyriakos Ispoglou et al. "Block Oriented Programming: Automating Data-Only Attacks". In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (Jan. 15, 2018)

Verification of Programs
Why?
How?
What?
Abstract Interpretation
A Bit of Architecture and System and Inline Assembly
Reduced Product with Ghost Variables
Discussion and Conclusion

What? — A Big Stack

To certify software systems, we have to consider:

- End-user software (more classical)
- Libraries (similar)
- Compiler
- Run-time environment
- Operating system (here I work)
- Hypervisor
- Hardware
- Physics

The operating system layer is the place of terrible low-level operations.

Verification of Programs
Why?
How?
What?
Abstract Interpretation
A Bit of Architecture and System and Inline Assembly
Reduced Product with Ghost Variables
Discussion and Conclusion

What? — My Study Case

An industrial partner wants to verify its operating system for a x86 platform at the boundary between trusted and untrusted worlds:

- no run-time errors: we need it to always be alive;
- no memory access between processes (read or write);
- no privilege escalation;
- schedules the process with the highest priority;
- … other OS-specific properties.

Verification of Programs

Abstract Interpretation

A Bit of Architecture and System and Inline Assembly

Reduced Product with Ghost Variables

Discussion and Conclusion

Verification of Programs

Abstract Interpretation

- Some Characteristics
- Overapproximation on an Example
- The General Framework
- Reduced Product
- Abstract Domains

A Bit of Architecture and System and Inline Assembly

Reduced Product with Ghost Variables

Discussion and Conclusion

Some Characteristics

Abstract interpretation

- is a static analysis method: works on existing source code,
- can (should) be sound (detect all errors),
- is based on approximation of the semantics.

Verification of Programs
Abstract Interpretation
Some Characteristics
Overapproximation on an Example
The General Framework
Reduced Product
Abstract Domains
A Bit of Architecture and System and Inline Assembly
Reduced Product with Ghost Variables
Discussion and Conclusion

Overapproximation on an Example

```
1  /*@ \requires x % 2 == 0; */
2  int f(int x)
3  {                        // x ∈ [−2^31, 2^31 − 2]
4     int y = abs(x);       // y ∈ [0, 2^31 − 1] ∨ x = −2^31
5     int z = y + 1;        // z ∈ [1, 2^31 − 1] ∨ y = 2^31 − 1
6     return 1/z;           // 0 ∉ [1, 2^31 − 1] ⇒ OK!
7  }
```

- Line 4: real error;
- Line 5: false alarm, since y is always even (since x is even).

Verification of Programs

Abstract Interpretation
- Some Characteristics
- Overapproximation on an Example
- The General Framework
- Reduced Product
- Abstract Domains

A Bit of Architecture and System and Inline Assembly

Reduced Product with Ghost Variables

Discussion and Conclusion

The General Framework — Concrete World

$(D, \subseteq)$ a concrete poset too big/complicated to be represented directly: set of programs traces/states…

$$[\![P]\!] = f_1 \fatsemi \cdots \fatsemi f_n \qquad \text{with } f_i : D \to D.$$

We want that $\forall i, c_i := (f_1 \fatsemi \cdots \fatsemi f_i)(c_0) \subseteq$ specification

Verification of Programs

Abstract Interpretation
- Some Characteristics
- Overapproximation on an Example
- The General Framework
- Reduced Product
- Abstract Domains

A Bit of Architecture and System and Inline Assembly

Reduced Product with Ghost Variables

Discussion and Conclusion

The General Framework — Abstract World

- $(D^\sharp, \sqsubseteq)$, an abstract poset usable by a computer,
- $\gamma : D^\sharp \to D$, the concretization: the meaning of an abstract element in the concrete world.

$f_i^\sharp : D^\sharp \to D^\sharp$ the abstract counterpart of f_i, such that

$$f_i \circ \gamma \subseteq \gamma \circ f_i^\sharp \quad \text{(soundness)}$$

If $c_0 \subseteq \gamma(a_0)$ then

$$\forall i, (f_1 \mathbin{⨟} \cdots \mathbin{⨟} f_i)(c_0) \subseteq \gamma \circ \underbrace{(f_1^\sharp \mathbin{⨟} \cdots \mathbin{⨟} f_i^\sharp)(a_0)}_{a_i}$$

Goal:

$$\forall i, c_i \underset{\substack{\uparrow \\ \text{soundness}}}{\subseteq} \gamma(a_i) \underset{\substack{\uparrow \\ \text{what we want}}}{\subseteq} \text{specification}$$

Verification of Programs

Abstract Interpretation
- Some Characteristics
- Overapproximation on an Example
- The General Framework
- Reduced Product
- Abstract Domains

A Bit of Architecture and System and Inline Assembly

Reduced Product with Ghost Variables

Discussion and Conclusion

The General Framework — An Abstract Operator

$$[\![\times 2]\!] \circ \gamma \quad \subseteq \quad \gamma \circ [\![\times 2]\!]^\sharp$$

$$[-1;1] \xrightarrow{\gamma} \{-1,0,1\} \xrightarrow{\times 2} \{-2,0,2\}$$

$$[-1;1] \xrightarrow{(\times 2)^\sharp} [-2;2] \xrightarrow{\gamma} \{-2,-1,0,1,2\}$$

Reduced Product — Example

```
1  /*@ \requires x % 2 == 0; */
2  int f(int x)
3  {                      // x ∈ [−2^31, 2^31 − 2] ∧ x ≡ 0 [2]
4     int y = abs(x);     // (y ∈ [0, 2^31 − 2] ∧ y ≡ 0 [2]) ∨ x = −2^31
5     int z = y + 1;      // z ∈ [1, 2^31 − 1] ∧ z ≡ 1 [2]
6     return 1/z;         // 0 ∉ [1, 2^31 − 1] ⇒ OK!
7  }
```

- Line 4: real error;
- Line 5: no more alarm

Verification of Programs

Abstract Interpretation
- Some Characteristics
- Overapproximation on an Example
- The General Framework
- Reduced Product
- Abstract Domains

A Bit of Architecture and System and Inline Assembly

Reduced Product with Ghost Variables

Discussion and Conclusion

Reduced Product

Given two abstract domains $\left(D_1^\sharp, \gamma_1\right)$, $\left(D_2^\sharp, \gamma_2\right)$, we define $(D^\sharp, \gamma)$ where $D^\sharp = D_1^\sharp \times D_2^\sharp$ and

$$\begin{aligned} \gamma : D^\sharp &\to D \\ (a_1, a_2) &\mapsto \gamma_1(a_1) \cap \gamma_2(a_2) \end{aligned}$$

Reduction operator $\rho : D^\sharp \to D^\sharp$

$$\begin{array}{lllll} & \forall a \in D^\sharp, \gamma(a) & \subseteq & \gamma \circ \rho(a) & \text{(sound)} \\ \text{Morally} & \forall a \in D^\sharp, \rho(a) & \sqsubseteq & a & \text{(improvement)} \end{array}$$

Proving the Security of Software-Intensive Embedded Systems by Abstract Interpretation

Marc Chevalier

Verification of Programs

Abstract Interpretation

Some Characteristics

Overapproximation on an Example

The General Framework

Reduced Product

Abstract Domains

A Bit of Architecture and System and Inline Assembly

Reduced Product with Ghost Variables

Discussion and Conclusion

Reduced Product — Communication Channel

Communication channel: $\left(IO^\sharp, \gamma_{IO}\right)$.

For each abstract domain:

- $\textsc{Enrich} : D^\sharp \times IO^\sharp \to IO^\sharp$

$$\forall (a, io) \in D^\sharp \times IO^\sharp, \gamma(a) \cap \gamma_{IO}(io) \subseteq \gamma_{IO}(\textsc{Enrich}(a, io))$$

- $\textsc{Refine} : D^\sharp \times IO^\sharp \to D^\sharp$

$$\forall (a, io) \in D^\sharp \times IO^\sharp, \gamma(a) \cap \gamma_{IO}(io) \subseteq \gamma(\textsc{Refine}(a, io))$$

Domains side-to-side communicate constraints through $IO^\sharp$.

Abstract Domains

- Numerical:

  Non-relational:
  - Modulo: $x_i \equiv c_i[n_i]$
  - Bitwise: $x_i = 0?1??100010111????$
  - Sign: $x_i < 0$, $x_i > 0$, $x_i \leqslant 0 \ldots$

  Relational:
  - Equality: $x_i = x_j$
  - Polytope: $\sum a_i x_i \leqslant c_i$
  - Octagon: $\pm x_i \pm x_j \leqslant c_i$

  And combination of domains.
- Memory: some value points to another, memory structures, separation logic. . . .
- Partitioning: $(x > 0 \Rightarrow \ldots) \wedge (x \leqslant 0 \Rightarrow \ldots)$
- All ad hoc domain you need.

Verification of Programs

Abstract Interpretation

A Bit of Architecture and System and Inline Assembly

Reduced Product with Ghost Variables

Discussion and Conclusion

Verification of Programs
Abstract Interpretation
A Bit of Architecture and System and Inline Assembly
Memory Protection
Goals
What I Did
How It Works
Reduced Product with Ghost Variables
Discussion and Conclusion

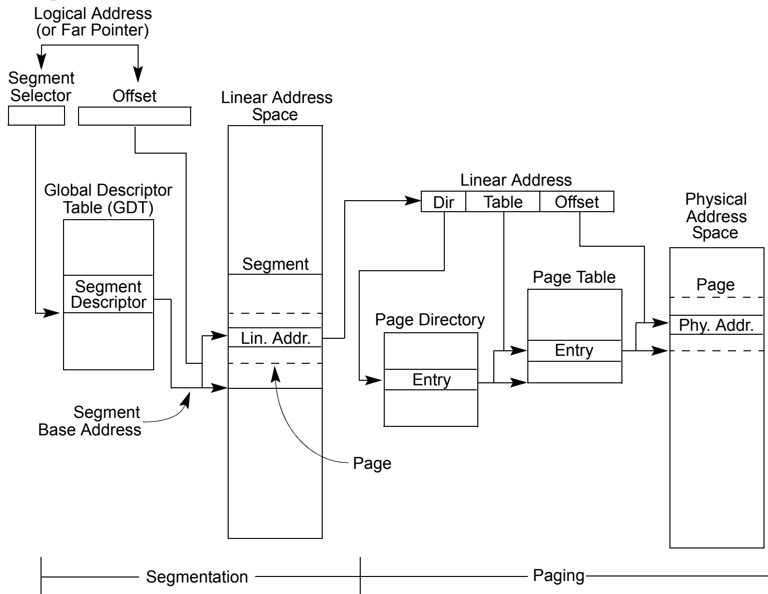
Memory Protection

There are many things to prove. Let's start with memory isolation: a process cannot access the memory of another one.

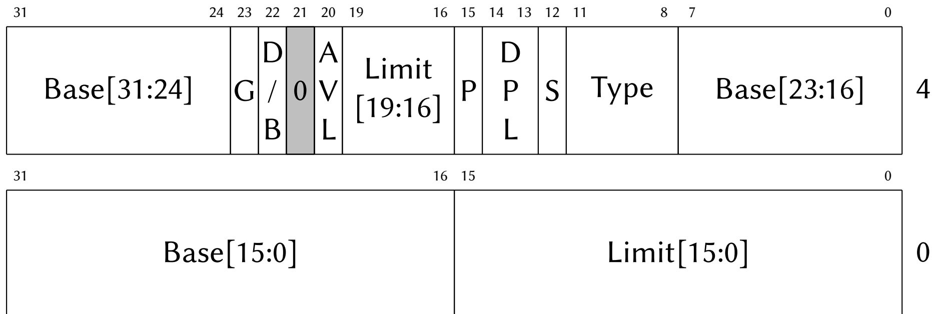
This is ensured using memory management feature of x86 CPUs.

- Non-interference is a property on sets of traces: hard
- Correctly using these features is a property on traces: less hard

Memory Protection



Memory Protection — Segment Descriptor



Verification of Programs
Abstract Interpretation
A Bit of Architecture and System and Inline Assembly
Memory Protection
Goals
What I Did
How It Works
Reduced Product with Ghost Variables
Discussion and Conclusion

Memory Protection — Building a SD

```
1  uint64_t /* A type of 64 bits */
2  build_sd(uint32_t* base, uint32_t limit,
3      uint32_t type, uint32_t flags1, uint32_t flags2) {
4      uint64_t seg_desc = 0;
5      seg_desc = limit & 0xffff;
6      seg_desc |= (base & 0xffff) << 16;
7      seg_desc |= ((base >> 16) & 0xff) << 32;
8      seg_desc |= (type & 0xf) << (32 + 8);
9      seg_desc |= (flags1 & 0xf) << (32 + 12);
10     seg_desc |= ((limit >> 16) & 0xf) << (32 + 16);
11     seg_desc |= (flags2 & 0xf) << (32 + 20);
12     seg_desc |= (base >> 24) << (32 + 24);
13     return seg_desc;
14 }
```

Verification of Programs

Abstract Interpretation

A Bit of Architecture and System and Inline Assembly

Memory Protection

Goals

What I Did

How It Works

Reduced Product with Ghost Variables

Discussion and Conclusion

Memory Protection — Setting a GDT

```
1  struct {short limit; int offset;} pointer_to_gdt;
2  uint64_t gdt[N];
3  void set_gdt() {
4      pointer_to_gdt.limit = N * 8;
5      pointer_to_gdt.offset = (int)(&gdt);
6      gdt[0] = build_sd(base, limit, type, flags1, flags2);
7      // ...
8      asm {
9          lgdt pointer_to_gdt;
10     }
11 }
```

Verification of Programs
Abstract Interpretation
A Bit of Architecture and System and Inline Assembly
Memory Protection
Goals
What I Did
How It Works
Reduced Product with Ghost Variables
Discussion and Conclusion

Goals

2 directions:

- Mix of C and assembly:
  - CPU-specific features,
  - complicated control flow impossible in C,
  - usual operations with more precise specification wrt. C.
- Abstract precisely complicated low-level operations:
  - bit slices,
  - linear combinations,
  - slices of linear combinations of pointers,
  - ...

Verification of Programs

Abstract Interpretation

A Bit of Architecture and System and Inline Assembly

Memory Protection

Goals

What I Did

How It Works

Reduced Product with Ghost Variables

Discussion and Conclusion

What I Did

- Abstract precisely complicated low-level operations (next section).
- Mix of C and assembly (sketch now):
  - Registers
  - Arithmetic and logic statements, status flags (EFLAGS)
  - Stack
  - Dynamic, local and inter-function jumps
  - Calls from C to assembly and from assembly to C
  - System statements
  - Dynamic code

Verification of Programs
Abstract Interpretation
A Bit of Architecture and System and Inline Assembly
Memory Protection
Goals
What I Did
How It Works
Reduced Product with Ghost Variables
Discussion and Conclusion

How It Works — Assembly Blocks

```
1  int a;
2  void f()
3  {
4      // C code
5      asm {
6          ; assembly code
7          mov a, 4
8      }
9  }
```

Verification of Programs
Abstract Interpretation
A Bit of Architecture and System and Inline Assembly
Memory Protection
Goals
What I Did
How It Works
Reduced Product with Ghost Variables
Discussion and Conclusion

How It Works — Register, Arithmetic & Logic

One variable for each register and signedness:

```
1  int a = 4;
2  int b;
3  asm {
4      mov EAX, a
5      mov b, EAX
6  }
```

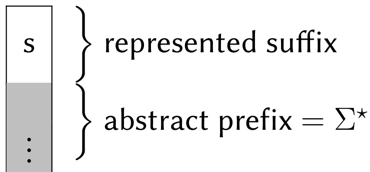
↓

```
1  int a = 4;
2  int b;
3  EAX_s = a;
4  EAX_u = *((unsigned int*)(&a));
5  b = EAX_s
```

Verification of Programs
Abstract Interpretation
A Bit of Architecture and System and Inline Assembly
Memory Protection
Goals
What I Did
How It Works
Reduced Product with Ghost Variables
Discussion and Conclusion

How It Works — Stack

We only represent what come above the stack: what come above the unknown zone.



Where

- s is empty at the beginning of an assembly block,
- s must be empty at the end of the assembly block.
- the abstracted prefix is a forbidden zone.

Verification of Programs

Abstract Interpretation

A Bit of Architecture and System and Inline Assembly

Memory Protection

Goals

What I Did

How It Works

Reduced Product with Ghost Variables

Discussion and Conclusion

How It Works — Jumps & co

- (Dynamic) jumps:
  - Resolve code pointers to program points.
  - lfp to collect possible environments at each target address.
- Mixed calls (C $\longleftrightarrow$ asm):
  - Implement both view of the calling convention.
  - Stack as an alternation of C and assembly call frames.
- System statements:
  - Compositions of simple statements on system registers.
- Dynamic code:
  - Controlled concretization of bytes, and disassembling.

Verification of Programs

Abstract Interpretation

A Bit of Architecture and System and Inline Assembly

Reduced Product with Ghost Variables

Discussion and Conclusion

Verification of Programs
Abstract Interpretation
A Bit of Architecture and System and Inline Assembly
Reduced Product with Ghost Variables
Ghost Variables
Building a Domain with Ghost Variables
A Bigger Problem
Sharing Ghost Variables
Dialectic
Discussion and Conclusion

Presentation of Ghost Variables

A ghost variable is a quantity that does not appear in the (standard) semantics of the program, but helps.

```
1 int* f(int* input) {
2       int low = (input + 1) & 0xffff;
3       int high = (input + 1) >> 16;
4       input = 0;
5       int* output = low | (high << 16);
6       return output;
7 }
```

Worse if we replace `(input + 1)` by `g(input)`.
Easy, if we remember the original value of `input + 1` (resp. `g(input)`).

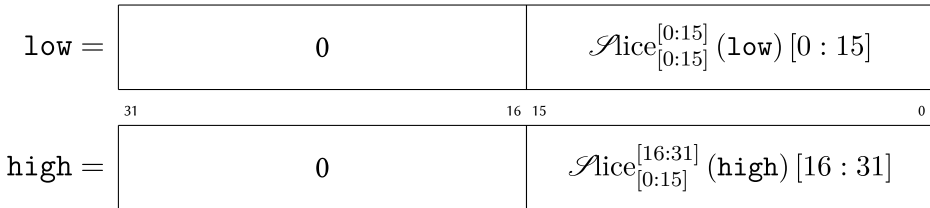
Proving the Security of Software-Intensive Embedded Systems by Abstract Interpretation

Marc Chevalier

Verification of Programs
Abstract Interpretation
A Bit of Architecture and System and Inline Assembly
Reduced Product with Ghost Variables
Ghost Variables
Building a Domain with Ghost Variables
A Bigger Problem
Sharing Ghost Variables
Dialectic
Discussion and Conclusion

Ghost Variables to the Rescue

```
1  int low = (input + 1) & 0xffff;
2  int high = (input + 1) >> 16;
3  ...
```



where

- $\mathscr{A}\text{lice}^{[0:15]}_{[0:15]}$ (low) and $\mathscr{A}\text{lice}^{[16:31]}_{[0:15]}$ (high) are ghost variables.
- $\mathscr{A}\text{lice}^{[0:15]}_{[0:15]}$ (low) = $\mathscr{A}\text{lice}^{[16:31]}_{[0:15]}$ (high) = `input + 1;`

Verification of Programs
Abstract Interpretation
A Bit of Architecture and System and Inline Assembly
Reduced Product with Ghost Variables
Ghost Variables
Building a Domain with Ghost Variables
A Bigger Problem
Sharing Ghost Variables
Dialectic
Discussion and Conclusion

Another Problem

```c
int something_interesting = ..., noise = ...;
int noisy = something_interesting + noise;
...
int clean = noisy - noise;
```

with enough assumptions:

$$\texttt{something_interesting} = \texttt{clean}$$

Verification of Programs
Abstract Interpretation
A Bit of Architecture and System and Inline Assembly
Reduced Product with Ghost Variables
Ghost Variables
Building a Domain with Ghost Variables
A Bigger Problem
Sharing Ghost Variables
Dialectic
Discussion and Conclusion

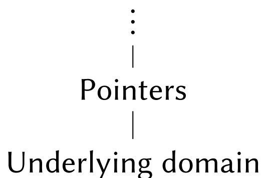
Ghost Variables for Pointers

```
1 int a[4];
2 int* p = &a[3]
```

$$\mathtt{p} = \mathtt{a} + \mathscr{O}\text{ffset}\,(p)$$
$$\mathscr{O}\text{ffset}\,(p) = 3 \times \mathtt{sizeof}\,(\mathtt{int})$$

Verification of Programs
Abstract Interpretation
A Bit of Architecture and System and Inline Assembly
Reduced Product with Ghost Variables
Ghost Variables
Building a Domain with Ghost Variables
A Bigger Problem
Sharing Ghost Variables
Dialectic
Discussion and Conclusion

Building a Domain with Ghost Variables



Theoretically:

Parametric domain.

Slices_D

Implementation:

Dependency injection.

(Objects, templates, functors…)

Verification of Programs

Abstract Interpretation

A Bit of Architecture and System and Inline Assembly

Reduced Product with Ghost Variables

Ghost Variables

Building a Domain with Ghost Variables

A Bigger Problem

Sharing Ghost Variables

Dialectic

Discussion and Conclusion

A Bigger Problem

Slices of linear combinations:

```
1  int* p = ...;
2  int* q = ...;
3  int* r = p + q;
4  int l = r & 0xffff;
5  int h = r << 16;
6  ... // kill r and p
7  int* r2 = l | h >> 16;
8  int* p2 = r2 - q;
```

(It happens… really)

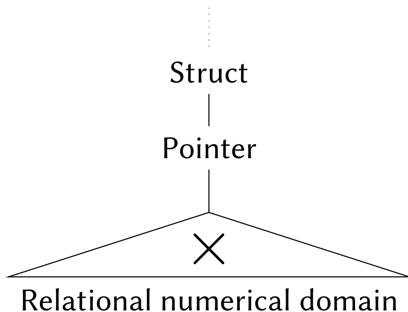
Linear combinations of slices:

```
1  int* p = ...;
2  int* n = ...;
3  int* n2 = ....;
4  int l = p & 0xffff;
5  int h = p << 16;
6  int a = l + n;
7  int b = h + n2;
8  ... // kill p, l and h
9  int l2 = a - n;
10 int h2 = b - n2;
11 int* p2 = l2 | h2 >> 16;
```

Thus any domain should be aware of everybody helping variables.

Verification of Programs
Abstract Interpretation
A Bit of Architecture and System and Inline Assembly
Reduced Product with Ghost Variables
Ghost Variables
Building a Domain with Ghost Variables
A Bigger Problem
Sharing Ghost Variables
Dialectic
Discussion and Conclusion

Sharing Ghost Variables — ASTRÉE on the Inside



Downsides

- Variables ids are handled by Struct domain: missing ids for ghost variables.
- There is no way to add another Pointer-like domain.

Verification of Programs

Abstract Interpretation

A Bit of Architecture and System and Inline Assembly

Reduced Product with Ghost Variables

Ghost Variables

Building a Domain with Ghost Variables

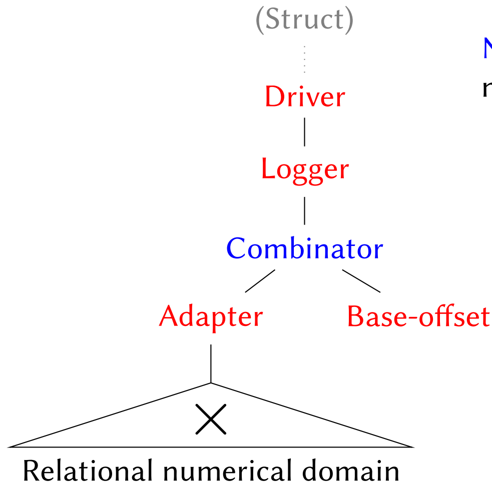
A Bigger Problem

Sharing Ghost Variables

Dialectic

Discussion and Conclusion

Sharing Ghost Variables — Astrées on the Inside



New combinator for pointers domains:

- Id translation by Driver
- Can add domains for pointer slices, linear combinations. . . .
- Cleaner interfaces.
- Each domain can ask everybody to store a ghost variable and do computations on it.
- Reuse old domains with Adapter.

Verification of Programs

Abstract Interpretation

A Bit of Architecture and System and Inline Assembly

Reduced Product with Ghost Variables

Ghost Variables

Building a Domain with Ghost Variables

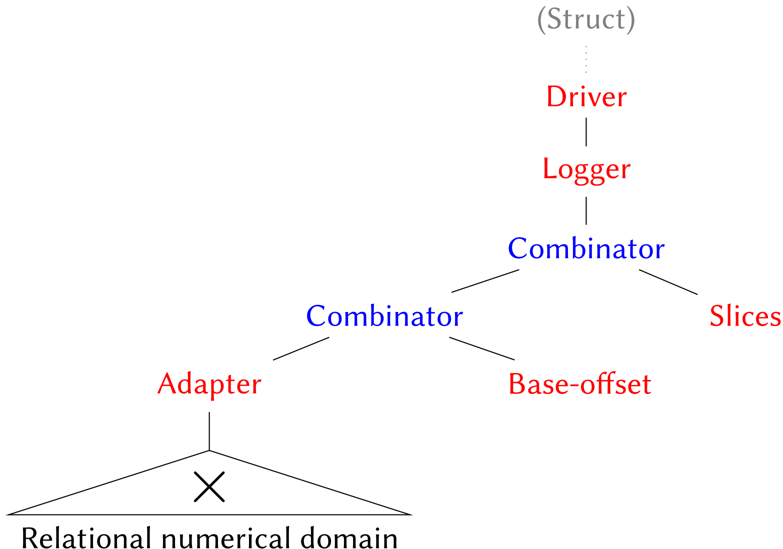
A Bigger Problem

Sharing Ghost Variables

Dialectic

Discussion and Conclusion

Sharing Ghost Variables — More



Verification of Programs
Abstract Interpretation
A Bit of Architecture and System and Inline Assembly
Reduced Product with Ghost Variables
Ghost Variables
Building a Domain with Ghost Variables
A Bigger Problem
Sharing Ghost Variables
Dialectic
Discussion and Conclusion

Sharing Ghost Variables — On an Example

```c
1 int a[4];
2 int* p = &a[3];
3 int low = p & 0xffff;
```

Before line 3:

$$\mathtt{p} = \mathtt{a} + \mathscr{O}\text{ffset}\,(\mathtt{p})$$
$$\mathscr{O}\text{ffset}\,(\mathtt{p}) = 12$$

Verification of Programs

Abstract Interpretation

A Bit of Architecture and System and Inline Assembly

Reduced Product with Ghost Variables

- Ghost Variables
- Building a Domain with Ghost Variables
- A Bigger Problem
- Sharing Ghost Variables
- Dialectic

Discussion and Conclusion

Sharing Ghost Variables — On an Example

```c
int a[4];
int* p = &a[3];
int low = p & 0xffff;
```

To run line 3, slice domain:

- requires the creation of $\mathscr{S}\mathrm{lice}_{[0:15]}^{[0:15]}(\texttt{low})$ (initially unknown),
- sets $\mathscr{S}\mathrm{lice}_{[0:15]}^{[0:15]}(\texttt{low})$ to p.

Pointer domain, to run $\mathscr{S}\mathrm{lice}_{[0:15]}^{[0:15]}(\texttt{low}) \leftarrow \texttt{p}$:

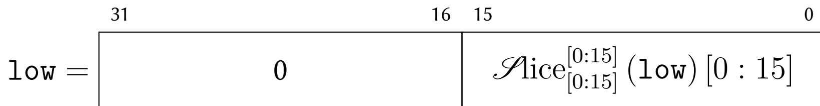
- requires the creation of $\mathscr{O}\mathrm{ffset}\left(\mathscr{S}\mathrm{lice}_{[0:15]}^{[0:15]}(\texttt{low})\right)$,
- sets $\mathscr{O}\mathrm{ffset}\left(\mathscr{S}\mathrm{lice}_{[0:15]}^{[0:15]}(\texttt{low})\right)$ to $\mathscr{O}\mathrm{ffset}(\texttt{p})$.

Verification of Programs
Abstract Interpretation
A Bit of Architecture and System and Inline Assembly
Reduced Product with Ghost Variables
Ghost Variables
Building a Domain with Ghost Variables
A Bigger Problem
Sharing Ghost Variables
Dialectic
Discussion and Conclusion

Sharing Ghost Variables — On an Example

```
1 int a[4];
2 int* p = &a[3];
3 int low = p & 0xffff;
```

After line 3:



$$\texttt{p} = \texttt{a} + \mathscr{O}\text{ffset}(\texttt{p})$$

$$\mathscr{S}\text{lice}^{[0:15]}_{[0:15]}(\texttt{low}) = \texttt{a} + \mathscr{O}\text{ffset}\left(\mathscr{S}\text{lice}^{[0:15]}_{[0:15]}(\texttt{low})\right)$$

$$\mathscr{O}\text{ffset}(\texttt{p}) = 12$$

$$\mathscr{O}\text{ffset}\left(\mathscr{S}\text{lice}^{[0:15]}_{[0:15]}(\texttt{low})\right) = 12$$

Verification of Programs

Abstract Interpretation

A Bit of Architecture and System and Inline Assembly

Reduced Product with Ghost Variables

Ghost Variables

Building a Domain with Ghost Variables

A Bigger Problem

Sharing Ghost Variables

Dialectic

Discussion and Conclusion

Sharing Ghost Variables — Adapting a Domain

We need to add ghost variable management features:

- role definition,
- ghost variable management (allocation, deletion, and update),
- support unification.

and change assignment: assigning real variable x resets to $\top$ all ghost variables under x.

Verification of Programs

Abstract Interpretation

A Bit of Architecture and System and Inline Assembly

Reduced Product with Ghost Variables

Ghost Variables

Building a Domain with Ghost Variables

A Bigger Problem

Sharing Ghost Variables

Dialectic

Discussion and Conclusion

Sharing Ghost Variables — Legitimate Concerns

Two problems:

- Each domain can change ghost variables: **soundness**.
- Each computation on ghost variables can generate new computations to perform: **termination**.

Verification of Programs

Abstract Interpretation

A Bit of Architecture and System and Inline Assembly

Reduced Product with Ghost Variables

Ghost Variables

Building a Domain with Ghost Variables

A Bigger Problem

Sharing Ghost Variables

Dialectic

Discussion and Conclusion

Sharing Ghost Variables — Soundness

2 ways to update ghost variables:

- regular constraints: $e \mathcal{R} e'$
- directed constraints: $v \leftarrow e$; equivalent to

  $$v = e$$

  when $v = \top$ and v does not appear in e. We can perform

  $$v := e$$

  Performance!

Verification of Programs

Abstract Interpretation

A Bit of Architecture and System and Inline Assembly

Reduced Product with Ghost Variables

Ghost Variables

Building a Domain with Ghost Variables

A Bigger Problem

Sharing Ghost Variables

Dialectic

Discussion and Conclusion

Sharing Ghost Variables — Soundness

Each ghost variable has a semantics defined by its role.

Constraint concretization must include the intersection of semantics of involved ghost variables.

E.g. semantics of $\mathscr{A}\mathrm{lice}^{[a:b]}_{[c:d]}(v)$ is all environments where bits a to b of $\mathscr{A}\mathrm{lice}^{[a:b]}_{[c:d]}(v)$ are bits c to d of v.

Verification of Programs

Abstract Interpretation

A Bit of Architecture and System and Inline Assembly

Reduced Product with Ghost Variables

Ghost Variables

Building a Domain with Ghost Variables

A Bigger Problem

Sharing Ghost Variables

Dialectic

Discussion and Conclusion

Sharing Ghost Variables — Soundness

But what about directed constraints?

$v \leftarrow e$ is $v = e$ where $v = \top$: it is only a performance concern.

How to be sure that $v = \top$?

By assigning ghostlier and ghostlier variables and using ownership, we never assign the same twice (and variables are independent since indirections are already solved).

Verification of Programs

Abstract Interpretation

A Bit of Architecture and System and Inline Assembly

Reduced Product with Ghost Variables

- Ghost Variables
- Building a Domain with Ghost Variables
- A Bigger Problem
- Sharing Ghost Variables
- Dialectic

Discussion and Conclusion

Sharing Ghost Variables — Soundness

Directed constraint $\rightarrow$ directed constraint: lhs is an immediately ghostlier variable belonging to the domain that emits this reduction, rhs is not less ghostly. $v \leftarrow e \Longrightarrow \mathscr{R}(v) \leftarrow f$ with $f \trianglelefteq e$.

Directed constraint $\rightarrow$ regular constraint on not less ghostly expressions. $v \leftarrow e \Longrightarrow f \mathcal{R} f'$ with $f \trianglelefteq e$ and $f' \trianglelefteq e$

Regular constraint $\rightarrow$ regular constraint on not less ghostly expressions. $e \mathcal{R} e' \Longrightarrow f \mathcal{S} f'$ with $f \trianglelefteq e$ and $f' \trianglelefteq e'$

A directed constraint can only come from a real assignment, without prior regular constraint. Following constraints are about ghostlier variables.

Verification of Programs

Abstract Interpretation

A Bit of Architecture and System and Inline Assembly

Reduced Product with Ghost Variables

Ghost Variables

Building a Domain with Ghost Variables

A Bigger Problem

Sharing Ghost Variables

Dialectic

Discussion and Conclusion

Sharing Ghost Variables — Termination

From each constraint, domains can generate new constraints.

We consume complexity of expressions to create ghost variables: simpler and about ghostlier variables.

Finite number of new variables.

Finite maximum depth for the tree of ghost variables, i.e. bounded support.

Since we constrain ghostlier and ghostlier variables, we eventually reach the bottom of the tree of ghost variables.

Dialectic

+ More general, future-proof
+ Solve my problem
+ Solve older problems
+ Remove some hacks
+ Cleaner code (more parametric, more abstraction)

- Tricky to implement
- More difficult to add new domains (but adapter for domains without ghost variables)
- More internal instructions for each real one: slower

And opportunistically: clean and optimize some old code I adapted.

Verification of Programs

Abstract Interpretation

A Bit of Architecture and System and Inline Assembly

Reduced Product with Ghost Variables

Ghost Variables

Building a Domain with Ghost Variables

A Bigger Problem

Sharing Ghost Variables

Dialectic

Discussion and Conclusion

Dialectic — How Slow?!

After a terrifying month of catastrophic performances, slowdown reduced to 3.

But we know why:

- More precision through more variables $\Rightarrow$ almost linear cost: acceptable trade-off.
- Pointer-numerical adapter spends a lot of time preparing union-like binary operations. Can be improved, heavy work.
- Pointer-numerical adapter spends even more time preparing comparison. Can be improved, heavy work, conceptually simple.
- Time spend in the product within experimental fluctuations.

Verification of Programs

Abstract Interpretation

A Bit of Architecture and System and Inline Assembly

Reduced Product with Ghost Variables

Discussion and Conclusion

Current Status

Future

Conclusion

Verification of Programs

Abstract Interpretation

A Bit of Architecture and System and Inline Assembly

Reduced Product with Ghost Variables

Discussion and Conclusion

Verification of Programs
Abstract Interpretation
A Bit of Architecture and System and Inline Assembly
Reduced Product with Ghost Variables
Discussion and Conclusion
Current Status
Future
Conclusion

Current Status

Astrée: 200 kloc OCaml

All my modifications:

- Astrée: +100 kloc -35 kloc
- Haunted product: +35 kloc -18 kloc
- Byproducts: 10 kloc OCaml, 10 kloc Python

A lot of implementation. Everything seems to work, tested on known industrial source code (without assembly). Assembly tested on tailored examples.

Used successfully by our partner on its study case (with assembly).

Current Status — What We Prove

2 main parts:

- initialization,
- handling of interrupts (including syscall and scheduler triggering).

No RTE in some parts of initialization.

Correctness of segmentation (wrt. modified flat model).

Verification of Programs
Abstract Interpretation
A Bit of Architecture and System and Inline Assembly
Reduced Product with Ghost Variables
Discussion and Conclusion
Current Status
Future
Conclusion

Future — Haunted Product

Two kinds of improvements:

- Performances:
  - adapting old domains for comparisons,
  - adapting old domains for unions,
  - transferring costly tasks when haunted product can do them more efficiently (partitioning),
- Features:
  - variable renaming (see supra and infra)
  - new domains: linear combination,
  - lifting above struct domain,
  - lifting it to the top to create a most general and modular analyzer.

Verification of Programs

Abstract Interpretation

A Bit of Architecture and System and Inline Assembly

Reduced Product with Ghost Variables

Discussion and Conclusion

Current Status

Future

Conclusion

Future — Inline Assembly

Problems:

- access through call frames (more hypotheses on the compiler),
- more on dynamic code,
- paging.

First is a problem for syscalls.

Two lasts are problems for initialization and syscalls.

Verification of Programs
Abstract Interpretation
A Bit of Architecture and System and Inline Assembly
Reduced Product with Ghost Variables
Discussion and Conclusion
Current Status
Future
Conclusion

Conclusion

Analyzing OSes is complicated: inline assembly and specific operations

Inline assembly well advanced, not so much to do, but few blocking problems to go further. Learned important lessons for mixed language support in general.

Some kinds of specific operations are well abstracted thanks to the haunted product. Still need clever memory model and data structure abstraction to analyze paging.

Solid foundations for solving remaining issues. Very promising.

Future — Mixed Languages

Mixed semantics depends on both languages:

- memory model fitted for both languages,
- interaction model.

Seems hard to solve in general:

- defines memory models as an abstraction of the lower-level memory model (x86),
- interaction still ad hoc.