manuscript: www.di.ens.fr/~delmas

Static Analysis of Program Portability by Abstract Interpretation

PhD defense

David Delmas

Airbus – Avionics Software

Sorbonne Université – LIP6

28 November 2022

The role of software and the cost of bugs

Safety-critical software

- flight-by-wire
- engine and breaks
- power plants
- pacemakers
- inertial systems

The role of software and the cost of bugs

Safety-critical software

- ✈ flight-by-wire
- 🚗 engine and breaks
- ☢ power plants
- pacemakers
- 🚀 inertial systems

Software bugs

serious consequences

The role of software and the cost of bugs

Safety-critical software

- flight-by-wire
- engine and breaks
- power plants
- pacemakers
- inertial systems

Software bugs

serious consequences

Evolving software

Bugs can be introduced in

- initial development
- later version **regression**
- new environment **portability error**

The role of software and the cost of bugs

Safety-critical software

- ✈ flight-by-wire
- 🚗 engine and breaks
- ☢ power plants
- 💼 pacemakers
- 🚀 inertial systems

Evolving software

Bugs can be introduced in

- initial development
- later version **regression**
- new environment **portability error**

Software bugs

serious consequences

Ariane 5.01 maiden flight

- reuse of Ariane 4 software
- different environment

The role of software and the cost of bugs

Photo Service Optique - CSG - © ESA - CNES

Ariane 5.01 maiden flight **failure**

- reuse of Ariane 4 software
- different environment
- direct cost: **500,000,000 $**

The role of software and the cost of bugs

Safety-critical software

- flight-by-wire
- engine and breaks
- power plants
- pacemakers
- inertial systems

Software bugs

serious consequences

Software verification

is mandatory

Evolving software

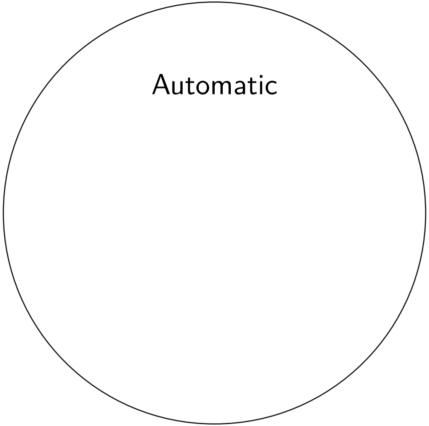
Bugs can be introduced in

- initial development
- later version **regression**
- new environment **portability error**

Ariane 5.01 maiden flight **failure**

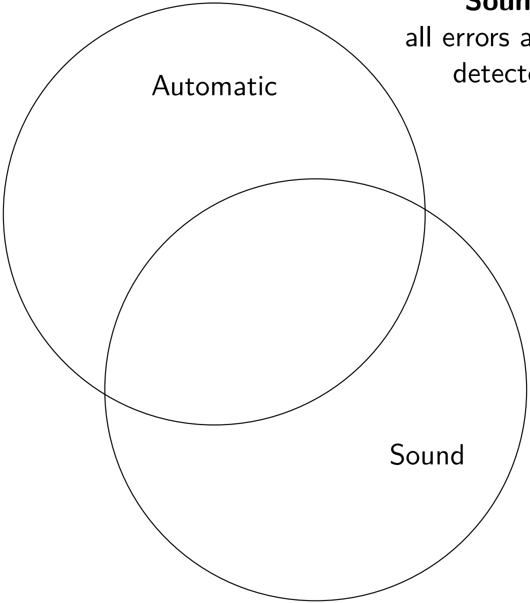
- reuse of Ariane 4 software
- different environment
- direct cost: **500,000,000 $**

Program verification techniques



Automatic

Program verification techniques



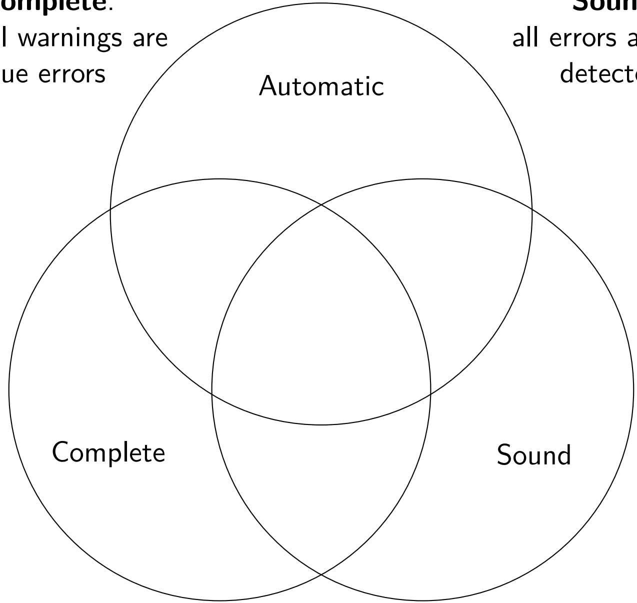
Sound:
all errors are
detected

Program verification techniques

© M. Journault

Complete:
all warnings are
true errors

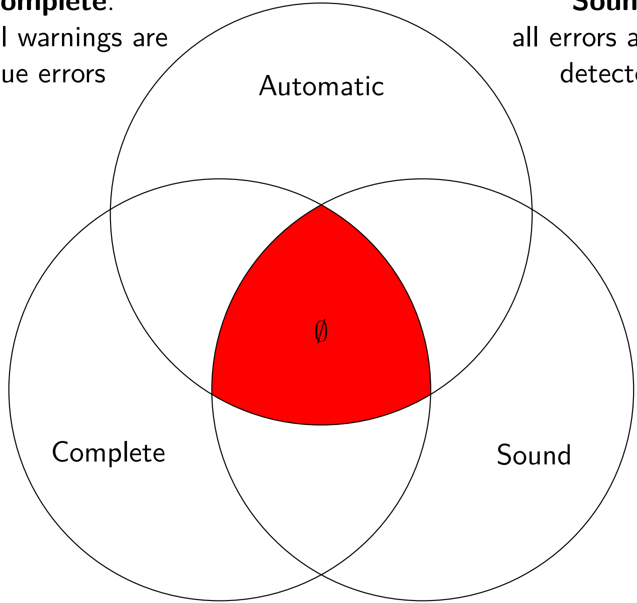
Sound:
all errors are
detected



Program verification techniques

Complete:
all warnings are
true errors

Sound:
all errors are
detected

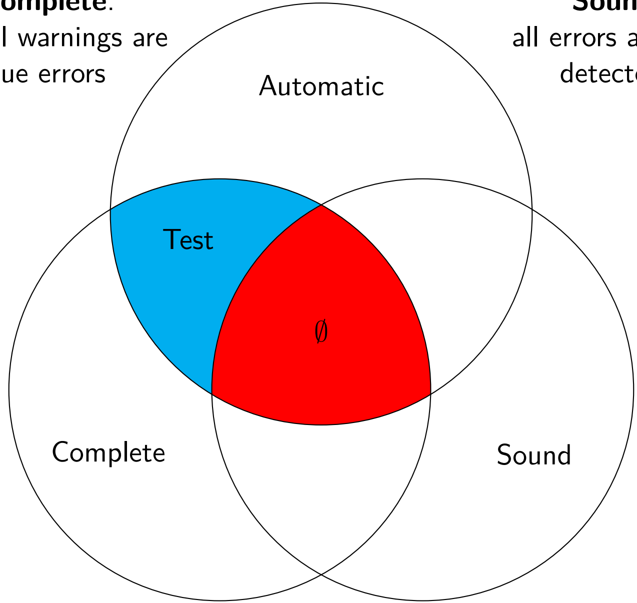


Program verification techniques

© M. Journault

Complete:
all warnings are
true errors

Sound:
all errors are
detected

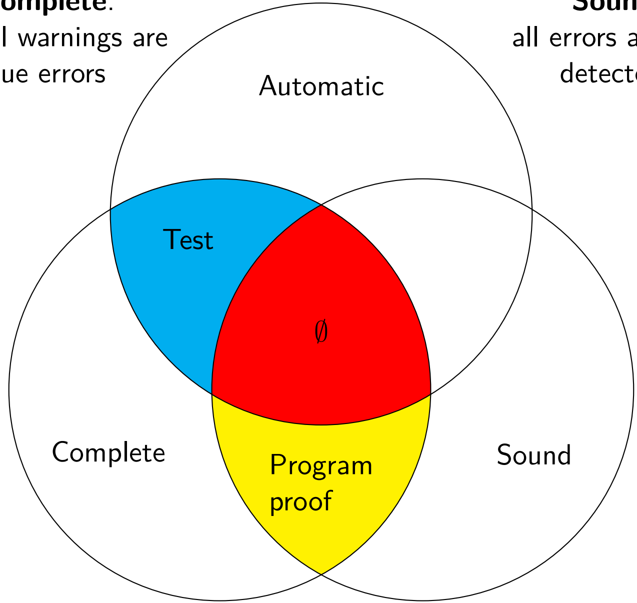


Program verification techniques

© M. Journault

Complete:
all warnings are
true errors

Sound:
all errors are
detected

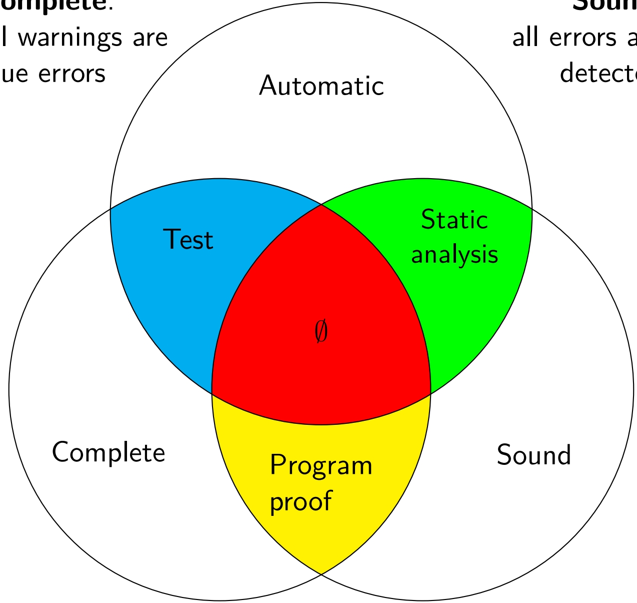


Program verification techniques

© M. Journault

Complete:
all warnings are
true errors

Sound:
all errors are
detected



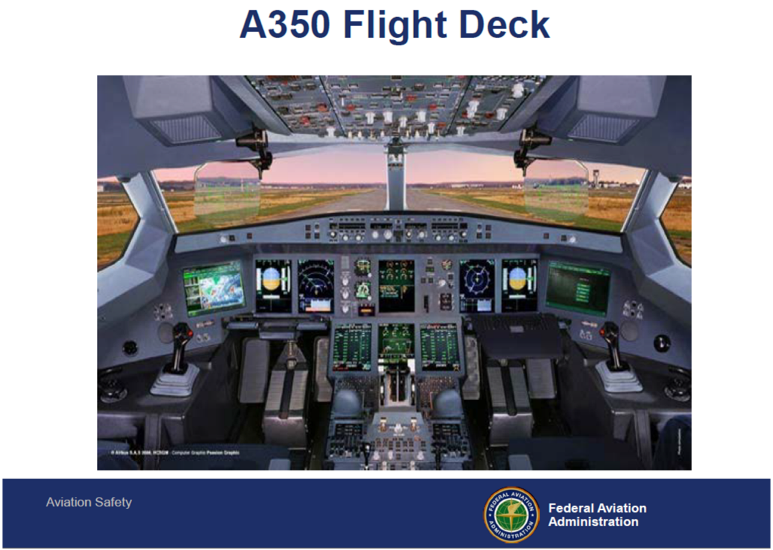
Aircraft functions transferred **from hardware**

de Havilland DH 106 Comet - 1949

Aviation Safety

Federal Aviation Administration

Aircraft functions transferred from hardware **to software**



A350 Flight Deck

Aviation Safety

Federal Aviation Administration

Software inside civil aircraft

Avionics software

- critical components of embedded systems
- e.g. flight-by-wire control systems
- major impact on safety
- widely used inside modern aircraft

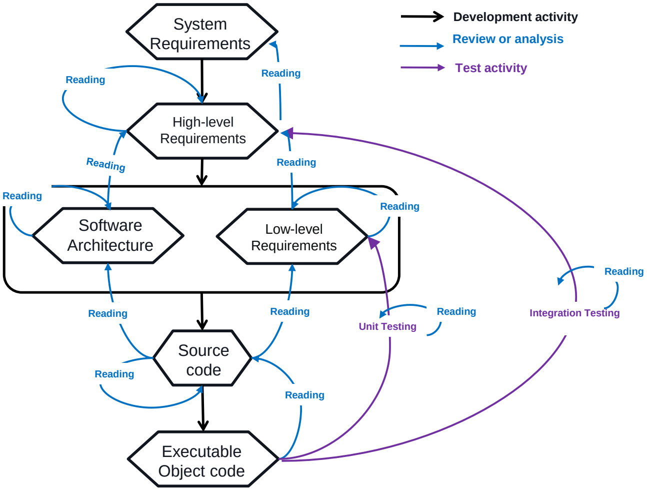
Certification

- by third parties on behalf of Authorities (FAA, EASA)
- stringent rules on **development and verification processes**
- DO-178/ED-12 international standard

Traditional process-based assurance **informal verification**

Large verification effort

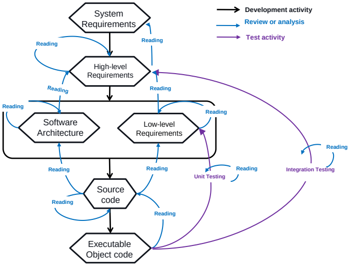
- intellectual **reviews**
- unit and integration **tests**

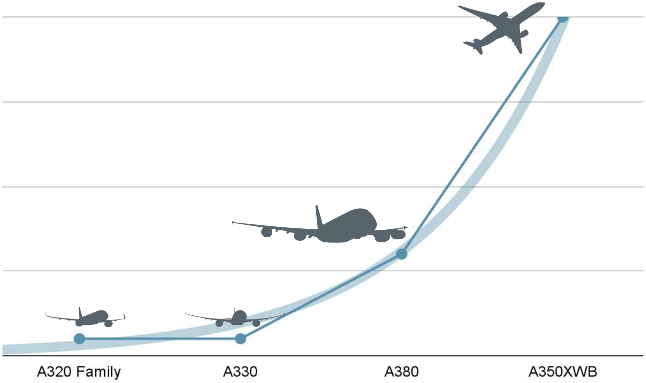


Traditional process-based assurance   **informal verification**

Large verification effort

- intellectual **reviews**
- unit and integration **tests**



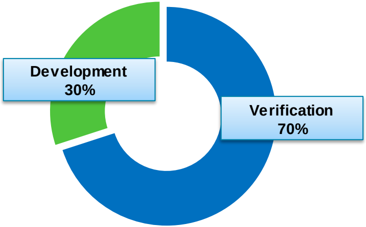


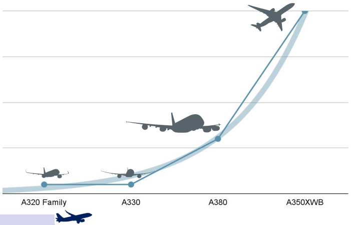
© V. Soumier

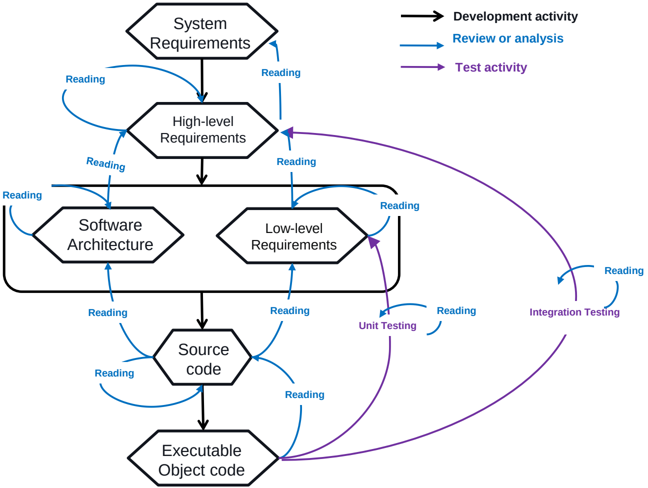
Traditional process-based assurance **informal verification**

Large verification effort

- intellectual **reviews**
- unit and integration **tests**







Automated process leveraging **formal verification**

Static analysis by AI

- absence of *run-time error*
- numerical accuracy
- stack usage
- WCET

Program proof

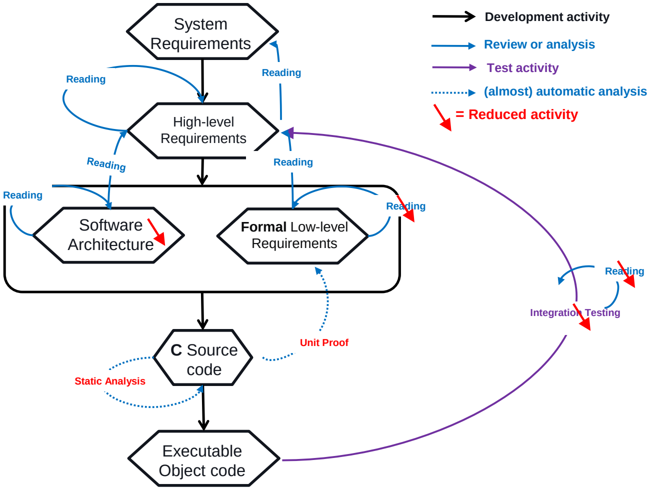
to replace unit testing

Source code verification

formally verified compiler

Industrial efficiency

cost savings in LLR processes



Principle of formal verification by abstract interpretation

Define the concrete semantics of your program

concrete semantics $\equiv$ mathematical model of the set of all its possible behaviours in all possible environments

can be constructed from semantics of commands of the programming language

Principle of formal verification by abstract interpretation

Define the concrete semantics of your program

concrete semantics ≡ mathematical model of the set of all its possible behaviours in all possible environments

can be constructed from semantics of commands of the programming language

Define a specification

specification ≡ subset of possible behaviours

Principle of formal verification by abstract interpretation

Define the concrete semantics of your program

concrete semantics ≡ mathematical model of the set of all its possible behaviours in all possible environments

can be constructed from semantics of commands of the programming language

Define a specification

specification ≡ subset of possible behaviours

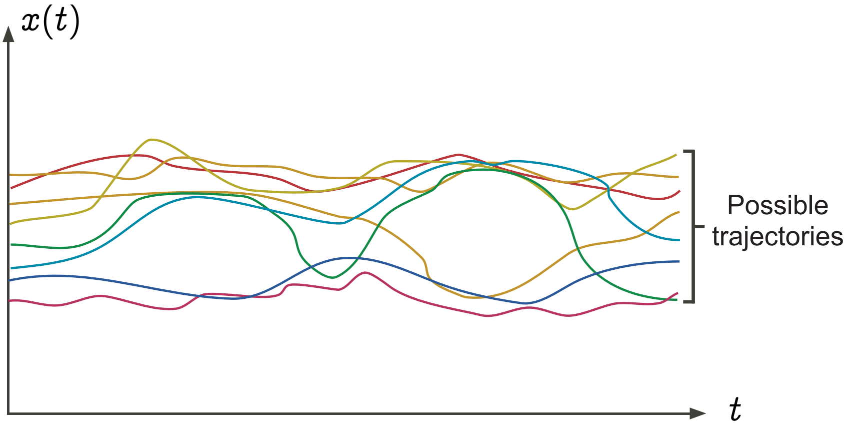
Conduct a formal proof

that the concrete semantics meets the specification

use computers to automate the proof

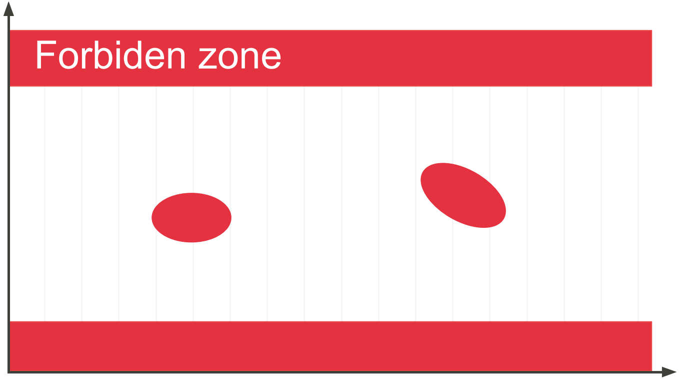
Concrete semantics of program P

© Cousot and Cousot [2010]



Semantics⟦ *P* ⟧

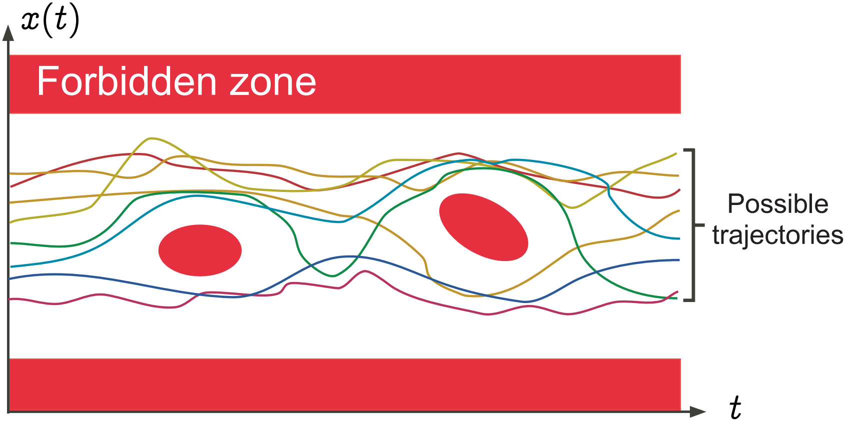
Specification of P (e.g. safety property) © Cousot and Cousot [2010]



$\mathit{Specification}[\![\, P \,]\!]$

Formal proof of P

© Cousot and Cousot [2010]

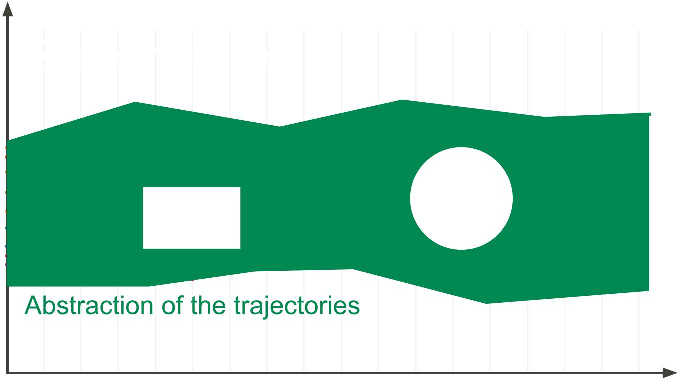


$\mathit{Semantics}[\![P]\!] \subseteq \mathit{Specification}[\![P]\!]$

Abstract semantics for P

© Cousot and Cousot [2010]

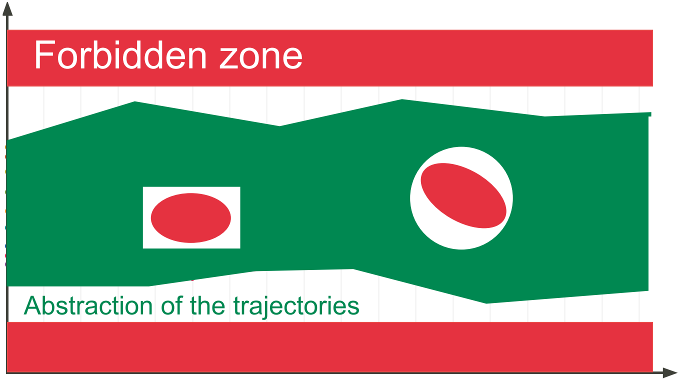
Semantics⟦ P ⟧ is uncomputable



Abstraction(*Semantics*⟦ P ⟧)

Proof by abstract interpretation

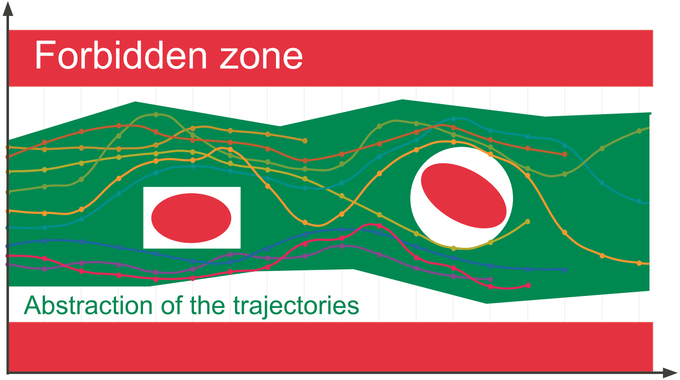
© Cousot and Cousot [2010]



$$\mathit{Abstraction}(\mathit{Semantics}[\![\, P \,]\!]) \subseteq \mathit{Specification}[\![\, P \,]\!]$$

Soundness of abstract interpretation

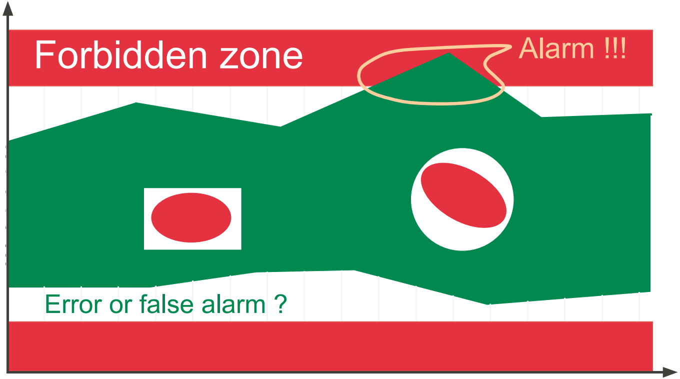
© Cousot and Cousot [2010]



$$Semantics[\![P]\!] \subseteq Abstraction(Semantics[\![P]\!]) \subseteq Specification[\![P]\!]$$

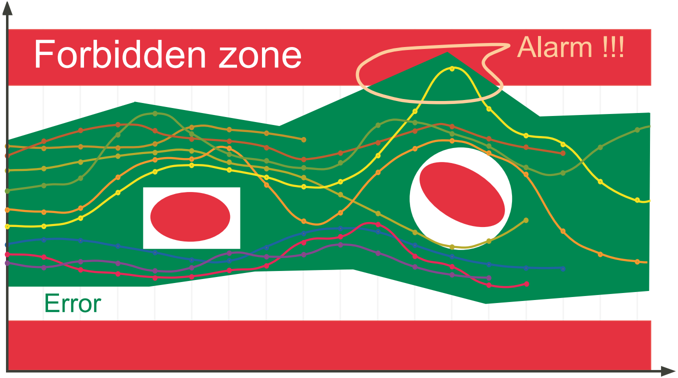
Static analyis by abstract interpretation © Cousot and Cousot [2010]

Alarms



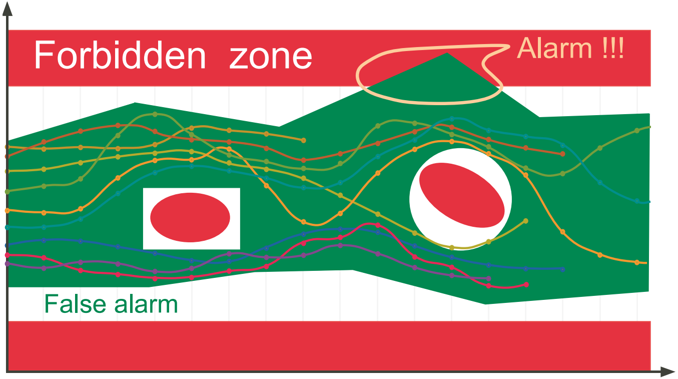
Static analyis by abstract interpretation © Cousot and Cousot [2010]

True error



Static analyis by abstract interpretation © Cousot and Cousot [2010]

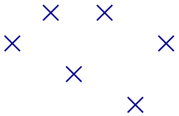
Incompleteness ⇒ false alarms



Static analyis by abstract interpretation

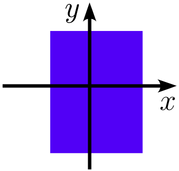
Numerical abstract domains

Bertrane et al. [2010]



Concrete values

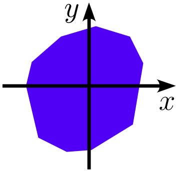
uncomputable



Intervals

$x, y \in [a, b]$

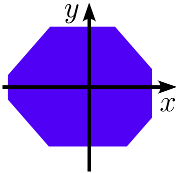
linear cost



Polyhedra

$\bigwedge \sum_i a_i x_i \leq b$

exponential cost



Octagons

$\bigwedge \pm x \pm y \leq c$

cubic cost

Abstract domains

- **sound** approximations of the concrete semantics
- trade-off between **cost** and **precision**

Goal of the thesis

Apply static analysis to two **program equivalence** problems

Regression verification

Objective: program change does not add undesirable behaviors

Patch analysis: inferring that two syntactically close versions of a program compute equal outputs when run on equal inputs in the **same environment**.

Portability verification

Objective: environment change does not add undesirable behaviors.

Portability analysis: inferring that two syntactically close versions of a program compute equal outputs when run on equal inputs in **their respective environments**.

Agenda

1. Introduction
2. Patch analysis for numerical programs
3. Patch analysis for C and structure layout portability
4. Endian portability analysis for C programs
5. Conclusion

Agenda

1. Introduction
2. Patch analysis for numerical programs
3. Patch analysis for C and structure layout portability
4. Endian portability analysis for C programs
5. Conclusion

Running example

Unchloop from Trostanetski et al. [2017]

Original program P_1

```
a = input(0,10);
b = input(0,10);
c = 1;

i=0;
while (i<a) {
  c=c+b;
  i=i+1;
}

r = c;
output(r);
```

Running example

Unchloop from Trostanetski et al. [2017]

Original and patched program versions P_1 and P_2

```
a = input(0,10);
b = input(0,10);
c = 1;

i=0;
while (i<a) {
  c=c+b;
  i=i+1;
}

r = c;
output(r);
```

```
a = input(0,10);
b = input(0,10);
c = 0;

i=0;
while (i<a) {
  c=c+b;
  i=i+1;
}

r = c+1;
output(r);
```

Running example

`Unchloop` from Trostanetski et al. [2017]

Original and patched program versions P_1 and P_2

assume: $a_1 = a_2 \wedge b_1 = b_2$ (equal inputs)

```
a = input(0,10);
b = input(0,10);
c = 1;

i=0;
while (i<a) {
  c=c+b;
  i=i+1;
}

r = c;
output(r);
```

```
a = input(0,10);
b = input(0,10);
c = 0;

i=0;
while (i<a) {
  c=c+b;
  i=i+1;
}

r = c+1;
output(r);
```

Running example

Unchloop from Trostanetski et al. [2017]

Original and patched program versions P_1 and P_2

assume: $a_1 = a_2 \land b_1 = b_2$ (equal inputs)

```
a = input(0,10);
b = input(0,10);
c = 1;

i=0;
while (i<a) {
  c=c+b;
  i=i+1;
}

r = c;
output(r);
```

```
a = input(0,10);
b = input(0,10);
c = 0;

i=0;
while (i<a) {
  c=c+b;
  i=i+1;
}

r = c+1;
output(r);
```

prove: $r_1 \overset{?}{=} r_2$ (equal outputs)

Running example

`Unchloop` from Trostanetski et al. [2017]

Invariants of program versions P_1 and P_2

assume: $a_1 = a_2 \wedge b_1 = b_2$ (equal inputs)

```
a = input(0,10);
b = input(0,10);
c = 1;

i=0;
while (i<a) {
  c=c+b;
  i=i+1;
}

r = c;
output(r);
```

$a_1 \in [0, 10]$

$b_1 \in [0, 10]$

$c_1 = b_1 \times i_1 + 1$

$c_1 = a_1 \times b_1 + 1$

$r_1 = a_1 \times b_1 + 1$

```
a = input(0,10);
b = input(0,10);
c = 0;

i=0;
while (i<a) {
  c=c+b;
  i=i+1;
}

r = c+1;
output(r);
```

$a_2 \in [0, 10]$

$b_2 \in [0, 10]$

$c_2 = b_2 \times i_2$

$c_2 = a_2 \times b_2$

$r_2 = a_2 \times b_2 + 1$

prove: $r_1 \overset{?}{=} r_2$ (equal outputs)

Running example

`Unchloop` from Trostanetski et al. [2017]

Invariants of program versions P_1 and P_2

assume: $a_1 = a_2 \wedge b_1 = b_2$ (equal inputs)

```
a = input(0,10);
b = input(0,10);
c = 1;

i=0;
while (i<a) {
  c=c+b;
  i=i+1;
}

r = c;
output(r);
```

$a_1 \in [0, 10]$

$b_1 \in [0, 10]$

$c_1 = b_1 \times i_1 + 1$

$c_1 = a_1 \times b_1 + 1$

$r_1 = a_1 \times b_1 + 1$

```
a = input(0,10);
b = input(0,10);
c = 0;

i=0;
while (i<a) {
  c=c+b;
  i=i+1;
}

r = c+1;
output(r);
```

$a_2 \in [0, 10]$

$b_2 \in [0, 10]$

$c_2 = b_2 \times i_2$

$c_2 = a_2 \times b_2$

$r_2 = a_2 \times b_2 + 1$

prove: $r_1 = r_2$ (equal outputs)

Running example

Unchloop from Trostanetski et al. [2017]

Proving the equivalence of program versions P_1 and P_2

assume: $a_1 = a_2 \wedge b_1 = b_2$ (equal inputs)

`output(r);` $r_1 = a_1 \times b_1 + 1$	`output(r);` $r_2 = a_2 \times b_2 + 1$

prove: $r_1 = r_2$ (equal outputs)

Proof of equivalence

from **separate** analyses of P_1 and P_2

requires inferring **expressive** relational invariants *(non linear)*

$\Longrightarrow$ **costly** numerical abstraction *(beyond polyhedra)*

Our approach

Joint analysis of program versions P_1 and P_2

First construct a double program P

from the AST of P_1 and P_2

using edit distance algorithms

with dynamic programming

```
a = input(0,10);
b = input(0,10);
c = 1;
i=0;
while (i<a) {
  c=c+b;
  i=i+1;
}
r = c;
output(r);
```

```
a = input(0,10);
b = input(0,10);
c = 0;
i=0;
while (i<a) {
  c=c+b;
  i=i+1;
}
r = c+1;
output(r);
```

Our approach

Joint analysis of program versions P_1 and P_2

First construct a double program P

from the AST of P_1 and P_2

using edit distance algorithms

with dynamic programming

```
a = input(0,10);
b = input(0,10);
c = 1;
i=0;
while (i<a) {
  c=c+b;
  i=i+1;
}
r = c;
output(r);
```

```
a = input(0,10);
b = input(0,10);
c = 1 || 0;
i=0;
while (i<a) {
  c=c+b;
  i=i+1;
}
r = c || c+1;
output(r);
```

```
a = input(0,10);
b = input(0,10);
c = 0;
i=0;
while (i<a) {
  c=c+b;
  i=i+1;
}
r = c+1;
output(r);
```

Our approach

Joint analysis of program versions P_1 and P_2

First construct a double program P

from the AST of P_1 and P_2

using edit distance algorithms

with dynamic programming

Left version: $P_1 = \pi_1(P)$

$$\pi_1(s_1 \parallel s_2) \triangleq s_1$$
$$\pi_1(\mathsf{c} = 1 \parallel 0) = \mathsf{c} = 1$$
$$\pi_1(\mathsf{r} = \mathsf{c} \parallel \mathsf{c}+1) = \mathsf{r} = \mathsf{c}$$

```
a = input(0,10);
b = input(0,10);
c = 1;
i=0;
while (i<a) {
  c=c+b;
  i=i+1;
}
r = c;
output(r);
```

```
a = input(0,10);
b = input(0,10);
c = 1 ‖ 0;
i=0;
while (i<a) {
  c=c+b;
  i=i+1;
}
r = c ‖ c+1;
output(r);
```

Our approach

Joint analysis of program versions P_1 and P_2

First construct a double program P

from the AST of P_1 and P_2

using edit distance algorithms

with dynamic programming

Right version: $P_2 = \pi_2(P)$

$$\begin{aligned} \pi_2(s_1 \parallel s_2) &\triangleq s_2 \\ \pi_2(\mathsf{c} = 1 \parallel 0) &= \mathsf{c} = 0 \\ \pi_2(\mathsf{r} = \mathsf{c} \parallel \mathsf{c}{+}1) &= \mathsf{r} = \mathsf{c}{+}1 \end{aligned}$$

```
a = input(0,10);
b = input(0,10);
c = 1 ‖ 0;
i=0;
while (i<a) {
  c=c+b;
  i=i+1;
}
r = c ‖ c+1;
output(r);
```

```
a = input(0,10);
b = input(0,10);
c = 0;
i=0;
while (i<a) {
  c=c+b;
  i=i+1;
}
r = c+1;
output(r);
```

Our approach

Joint analysis of program versions P_1 and P_2

First construct a double program P

from the AST of P_1 and P_2

using edit distance algorithms

with dynamic programming

Then analyze the double program P

using double program semantics

relating variables of P_1 and P_2

with **less expressive** invariants *(linear)*

```
a = input(0,10);
b = input(0,10);
c = 1 || 0;
i=0;
while (i<a) {
  c=c+b;
  i=i+1;
}
r = c || c+1;
output(r);
```

$a_1 = a_2 \in [0, 10]$

$b_1 = b_2 \in [0, 10]$

$c_1 = 1 \wedge c_2 = 0$

$c_1 = c_2 + 1$

$r_1 = r_2$

Lifting simple program semantics to double programs

Concrete domain of simple programs

Simple programs P_1 and P_2

Simple states in $\mathcal{E} \triangleq \mathcal{V} \to \mathbb{Z}$

Semantics $\mathbb{S}[\![s]\!] \in \mathcal{P}(\mathcal{E}) \to \mathcal{P}(\mathcal{E})$

Lifting simple program semantics to double programs

Concrete domain of simple programs and double programs

Simple programs P_1 and P_2

Simple states in $\mathcal{E} \triangleq \mathcal{V} \to \mathbb{Z}$

Semantics $\mathbb{S}[\![s]\!] \in \mathcal{P}(\mathcal{E}) \to \mathcal{P}(\mathcal{E})$

Double program P

Double states in $\mathcal{D} \triangleq \mathcal{E} \times \mathcal{E}$

Semantics $\mathbb{D}[\![s]\!] \in \mathcal{P}(\mathcal{D}) \to \mathcal{P}(\mathcal{D})$

Lifting simple program semantics to double programs

Patch, input, output, assignment and bloc statements

Simple programs P_1 and P_2

Simple states in $\mathcal{E} \triangleq \mathcal{V} \to \mathbb{Z}$

Semantics $\mathbb{S}[\![\, s \,]\!] \in \mathcal{P}(\mathcal{E}) \to \mathcal{P}(\mathcal{E})$

Double program P

Double states in $\mathcal{D} \triangleq \mathcal{E} \times \mathcal{E}$

Semantics $\mathbb{D}[\![\, s \,]\!] \in \mathcal{P}(\mathcal{D}) \to \mathcal{P}(\mathcal{D})$

$$\mathbb{D}[\![\, s_1 \parallel s_2 \,]\!]X \qquad \triangleq \bigcup_{(\rho_1,\rho_2)\in X} \{\, (\rho_1', \rho_2') \mid \rho_1' \in \mathbb{S}[\![\, s_1 \,]\!]\{\, \rho_1 \,\} \wedge \rho_2' \in \mathbb{S}[\![\, s_2 \,]\!]\{\, \rho_2 \,\} \,\}$$

Lifting simple program semantics to double programs

Patch, input, output, assignment and bloc statements

Simple programs P_1 and P_2

Simple states in $\mathcal{E} \triangleq \mathcal{V} \to \mathbb{Z}$

Semantics $\mathbb{S}[\![\, s \,]\!] \in \mathcal{P}(\mathcal{E}) \to \mathcal{P}(\mathcal{E})$

Double program P

Double states in $\mathcal{D} \triangleq \mathcal{E} \times \mathcal{E}$

Semantics $\mathbb{D}[\![\, s \,]\!] \in \mathcal{P}(\mathcal{D}) \to \mathcal{P}(\mathcal{D})$

$$\mathbb{D}[\![\, s_1 \parallel s_2 \,]\!]X \triangleq \bigcup_{(\rho_1,\rho_2)\in X} \{\, (\rho'_1, \rho'_2) \mid \rho'_1 \in \mathbb{S}[\![\, s_1 \,]\!]\{\, \rho_1 \,\} \land \rho'_2 \in \mathbb{S}[\![\, s_2 \,]\!]\{\, \rho_2 \,\} \,\}$$

$$\mathbb{D}[\![\, V \leftarrow e_1 \parallel e_2 \,]\!] \triangleq \mathbb{D}[\![\, V \leftarrow e_1 \parallel V \leftarrow e_2 \,]\!]$$

$$\mathbb{D}[\![\, V \leftarrow e \,]\!] \triangleq \mathbb{D}[\![\, V \leftarrow e \parallel V \leftarrow e \,]\!]$$

Lifting simple program semantics to double programs

Patch, input, output, assignment and bloc statements

Simple programs P_1 and P_2

Simple states in $\mathcal{E} \triangleq \mathcal{V} \to \mathbb{Z}$

Semantics $\mathbb{S}[\![\, s \,]\!] \in \mathcal{P}(\mathcal{E}) \to \mathcal{P}(\mathcal{E})$

Double program P

Double states in $\mathcal{D} \triangleq \mathcal{E} \times \mathcal{E}$

Semantics $\mathbb{D}[\![\, s \,]\!] \in \mathcal{P}(\mathcal{D}) \to \mathcal{P}(\mathcal{D})$

$$\mathbb{D}[\![\, s_1 \parallel s_2 \,]\!]X \triangleq \bigcup_{(\rho_1,\rho_2)\in X}\{\, (\rho'_1, \rho'_2) \mid \rho'_1 \in \mathbb{S}[\![\, s_1 \,]\!]\{\, \rho_1 \,\} \wedge \rho'_2 \in \mathbb{S}[\![\, s_2 \,]\!]\{\, \rho_2 \,\} \,\}$$

$$\mathbb{D}[\![\, V \leftarrow e_1 \parallel e_2 \,]\!] \triangleq \mathbb{D}[\![\, V \leftarrow e_1 \parallel V \leftarrow e_2 \,]\!]$$

$$\mathbb{D}[\![\, V \leftarrow e \,]\!] \triangleq \mathbb{D}[\![\, V \leftarrow e \parallel V \leftarrow e \,]\!]$$

$$\mathbb{D}[\![\, V \leftarrow \mathbf{input}(a,b) \,]\!]X \triangleq \{\, (\rho_1[V \mapsto v], \rho_2[V \mapsto v]) \mid v \in [a,b] \wedge (\rho_1,\rho_2) \in X \,\}$$

$$\mathbb{D}[\![\, \mathbf{output}(V) \,]\!]X \triangleq \{\, (\rho_1,\rho_2) \in X \mid \rho_1(V) = \rho_2(V) \,\}$$

Lifting simple program semantics to double programs

Patch, input, output, assignment and bloc statements

Simple programs P_1 and P_2

Simple states in $\mathcal{E} \triangleq \mathcal{V} \rightarrow \mathbb{Z}$

Semantics $\mathbb{S}[\![\, s \,]\!] \in \mathcal{P}(\mathcal{E}) \rightarrow \mathcal{P}(\mathcal{E})$

Double program P

Double states in $\mathcal{D} \triangleq \mathcal{E} \times \mathcal{E}$

Semantics $\mathbb{D}[\![\, s \,]\!] \in \mathcal{P}(\mathcal{D}) \rightarrow \mathcal{P}(\mathcal{D})$

$$\mathbb{D}[\![\, s_1 \parallel s_2 \,]\!]X \triangleq \bigcup_{(\rho_1,\rho_2)\in X} \{\, (\rho'_1, \rho'_2) \mid \rho'_1 \in \mathbb{S}[\![\, s_1 \,]\!]\{\, \rho_1 \,\} \wedge \rho'_2 \in \mathbb{S}[\![\, s_2 \,]\!]\{\, \rho_2 \,\} \,\}$$

$$\mathbb{D}[\![\, V \leftarrow e_1 \parallel e_2 \,]\!] \triangleq \mathbb{D}[\![\, V \leftarrow e_1 \parallel V \leftarrow e_2 \,]\!]$$

$$\mathbb{D}[\![\, V \leftarrow e \,]\!] \triangleq \mathbb{D}[\![\, V \leftarrow e \parallel V \leftarrow e \,]\!]$$

$$\mathbb{D}[\![\, V \leftarrow \mathbf{input}(a, b) \,]\!]X \triangleq \{\, (\rho_1[V \mapsto v], \rho_2[V \mapsto v]) \mid v \in [a, b] \wedge (\rho_1, \rho_2) \in X \,\}$$

$$\mathbb{D}[\![\, \mathbf{output}(V) \,]\!]X \triangleq \{\, (\rho_1, \rho_2) \in X \mid \rho_1(V) = \rho_2(V) \,\}$$

$$\mathbb{D}[\![\, s_1;\; s_2 \,]\!] \triangleq \mathbb{D}[\![\, s_2 \,]\!] \circ \mathbb{D}[\![\, s_1 \,]\!]$$

Lifting simple program semantics to double programs

if statement

Simple programs P_1 and P_2

Simple states in $\mathcal{E} \triangleq \mathcal{V} \to \mathbb{Z}$

Semantics $\mathbb{S}[\![s]\!] \in \mathcal{P}(\mathcal{E}) \to \mathcal{P}(\mathcal{E})$

Conditions $\mathbb{C}[\![c]\!] \in \mathcal{P}(\mathcal{E}) \to \mathcal{P}(\mathcal{E})$

Double program P

Double states in $\mathcal{D} \triangleq \mathcal{E} \times \mathcal{E}$

Semantics $\mathbb{D}[\![s]\!] \in \mathcal{P}(\mathcal{D}) \to \mathcal{P}(\mathcal{D})$

Conditions $\mathbb{F}[\![c_1 \parallel c_2]\!] \in \mathcal{P}(\mathcal{D}) \to \mathcal{P}(\mathcal{D})$

Lifting simple program semantics to double programs

if statement

Simple programs P_1 and P_2

Simple states in $\mathcal{E} \triangleq \mathcal{V} \to \mathbb{Z}$

Semantics $\mathbb{S}[\![s]\!] \in \mathcal{P}(\mathcal{E}) \to \mathcal{P}(\mathcal{E})$

Conditions $\mathbb{C}[\![c]\!] \in \mathcal{P}(\mathcal{E}) \to \mathcal{P}(\mathcal{E})$

Double program P

Double states in $\mathcal{D} \triangleq \mathcal{E} \times \mathcal{E}$

Semantics $\mathbb{D}[\![s]\!] \in \mathcal{P}(\mathcal{D}) \to \mathcal{P}(\mathcal{D})$

Conditions $\mathbb{F}[\![c_1 \parallel c_2]\!] \in \mathcal{P}(\mathcal{D}) \to \mathcal{P}(\mathcal{D})$

$$\mathbb{F}[\![c_1 \parallel c_2]\!]X \triangleq \{\, (\rho_1, \rho_2) \in X \mid \mathbb{C}[\![c_1]\!]\{\, \rho_1 \,\} \neq \emptyset \neq \mathbb{C}[\![c_2]\!]\{\, \rho_2 \,\} \,\}$$

Lifting simple program semantics to double programs

if statement

Simple programs P_1 and P_2

Simple states in $\mathcal{E} \triangleq \mathcal{V} \to \mathbb{Z}$

Semantics $\mathbb{S}[\![\, s \,]\!] \in \mathcal{P}(\mathcal{E}) \to \mathcal{P}(\mathcal{E})$

Conditions $\mathbb{C}[\![\, c \,]\!] \in \mathcal{P}(\mathcal{E}) \to \mathcal{P}(\mathcal{E})$

Double program P

Double states in $\mathcal{D} \triangleq \mathcal{E} \times \mathcal{E}$

Semantics $\mathbb{D}[\![\, s \,]\!] \in \mathcal{P}(\mathcal{D}) \to \mathcal{P}(\mathcal{D})$

Conditions $\mathbb{F}[\![\, c_1 \parallel c_2 \,]\!] \in \mathcal{P}(\mathcal{D}) \to \mathcal{P}(\mathcal{D})$

$$\mathbb{F}[\![\, c_1 \parallel c_2 \,]\!]X \triangleq \{\, (\rho_1, \rho_2) \in X \mid \mathbb{C}[\![\, c_1 \,]\!]\{\, \rho_1 \,\} \neq \emptyset \neq \mathbb{C}[\![\, c_2 \,]\!]\{\, \rho_2 \,\} \,\}$$

$$\begin{aligned}
\mathbb{D}[\![\, \textbf{if } c_1 \parallel c_2 \textbf{ then } s \textbf{ else } t \,]\!] \triangleq\; & \mathbb{D}[\![\qquad \dots \qquad]\!] \circ \mathbb{F}[\![\, c_1 \parallel c_2 \,]\!] \\
\dot\cup\; & \mathbb{D}[\![\qquad \dots \qquad]\!] \circ \mathbb{F}[\![\, \neg c_1 \parallel \neg c_2 \,]\!] \\
\dot\cup\; & \mathbb{D}[\![\qquad \dots \qquad]\!] \circ \mathbb{F}[\![\, c_1 \parallel \neg c_2 \,]\!] \\
\dot\cup\; & \mathbb{D}[\![\qquad \dots \qquad]\!] \circ \mathbb{F}[\![\, \neg c_1 \parallel c_2 \,]\!]
\end{aligned}$$

Lifting simple program semantics to double programs

if statement

Simple programs P_1 and P_2

Simple states in $\mathcal{E} \triangleq \mathcal{V} \to \mathbb{Z}$

Semantics $\mathbb{S}[\![\, s \,]\!] \in \mathcal{P}(\mathcal{E}) \to \mathcal{P}(\mathcal{E})$

Conditions $\mathbb{C}[\![\, c \,]\!] \in \mathcal{P}(\mathcal{E}) \to \mathcal{P}(\mathcal{E})$

Double program P

Double states in $\mathcal{D} \triangleq \mathcal{E} \times \mathcal{E}$

Semantics $\mathbb{D}[\![\, s \,]\!] \in \mathcal{P}(\mathcal{D}) \to \mathcal{P}(\mathcal{D})$

Conditions $\mathbb{F}[\![\, c_1 \parallel c_2 \,]\!] \in \mathcal{P}(\mathcal{D}) \to \mathcal{P}(\mathcal{D})$

$$\mathbb{F}[\![\, c_1 \parallel c_2 \,]\!]X \triangleq \{\, (\rho_1, \rho_2) \in X \mid \mathbb{C}[\![\, c_1 \,]\!]\{\, \rho_1 \,\} \neq \emptyset \neq \mathbb{C}[\![\, c_2 \,]\!]\{\, \rho_2 \,\} \,\}$$

$$\begin{array}{rcccl} \mathbb{D}[\![\, \textbf{if } c_1 \parallel c_2 \textbf{ then } s \textbf{ else } t \,]\!] & \triangleq & & \mathbb{D}[\![\, s \,]\!] & \circ\ \mathbb{F}[\![\, c_1 \parallel c_2 \,]\!] \\ & \dot\cup & & \mathbb{D}[\![\, t \,]\!] & \circ\ \mathbb{F}[\![\, \neg c_1 \parallel \neg c_2 \,]\!] \\ & \dot\cup & \mathbb{D}[\![& \ldots &]\!] \circ\ \mathbb{F}[\![\, c_1 \parallel \neg c_2 \,]\!] \\ & \dot\cup & \mathbb{D}[\![& \ldots &]\!] \circ\ \mathbb{F}[\![\, \neg c_1 \parallel c_2 \,]\!] \end{array}$$

Lifting simple program semantics to double programs

if statement

Simple programs P_1 and P_2

Simple states in $\mathcal{E} \triangleq \mathcal{V} \to \mathbb{Z}$

Semantics $\mathbb{S}[\![\, s \,]\!] \in \mathcal{P}(\mathcal{E}) \to \mathcal{P}(\mathcal{E})$

Conditions $\mathbb{C}[\![\, c \,]\!] \in \mathcal{P}(\mathcal{E}) \to \mathcal{P}(\mathcal{E})$

Double program P

Double states in $\mathcal{D} \triangleq \mathcal{E} \times \mathcal{E}$

Semantics $\mathbb{D}[\![\, s \,]\!] \in \mathcal{P}(\mathcal{D}) \to \mathcal{P}(\mathcal{D})$

Conditions $\mathbb{F}[\![\, c_1 \parallel c_2 \,]\!] \in \mathcal{P}(\mathcal{D}) \to \mathcal{P}(\mathcal{D})$

$$\mathbb{F}[\![\, c_1 \parallel c_2 \,]\!]X \triangleq \{\, (\rho_1, \rho_2) \in X \mid \mathbb{C}[\![\, c_1 \,]\!]\{\, \rho_1 \,\} \neq \emptyset \neq \mathbb{C}[\![\, c_2 \,]\!]\{\, \rho_2 \,\} \,\}$$

$$\begin{aligned}
\mathbb{D}[\![\, \textbf{if } c_1 \parallel c_2 \textbf{ then } s \textbf{ else } t \,]\!] \triangleq\ & \mathbb{D}[\![\, s \,]\!] \circ \mathbb{F}[\![\, c_1 \parallel c_2 \,]\!] \\
\dot{\cup}\ & \mathbb{D}[\![\, t \,]\!] \circ \mathbb{F}[\![\, \neg c_1 \parallel \neg c_2 \,]\!] \\
\dot{\cup}\ & \mathbb{D}[\![\, \pi_1(s) \parallel \pi_2(t) \,]\!] \circ \mathbb{F}[\![\, c_1 \parallel \neg c_2 \,]\!] \\
\dot{\cup}\ & \mathbb{D}[\![\, \pi_1(t) \parallel \pi_2(s) \,]\!] \circ \mathbb{F}[\![\, \neg c_1 \parallel c_2 \,]\!]
\end{aligned}$$

Lifting simple program semantics to double programs

while statement

Double program P

Double states in $\mathcal{D} \triangleq \mathcal{E} \times \mathcal{E}$

Semantics $\mathbb{D}[\![\, s \,]\!] \in \mathcal{P}(\mathcal{D}) \to \mathcal{P}(\mathcal{D})$

Conditions $\mathbb{F}[\![\, c_1 \parallel c_2 \,]\!] \in \mathcal{P}(\mathcal{D}) \to \mathcal{P}(\mathcal{D})$

Lifting simple program semantics to double programs

while statement

Double program P

Double states in $\mathcal{D} \triangleq \mathcal{E} \times \mathcal{E}$

Semantics $\mathbb{D}[\![\, s \,]\!] \in \mathcal{P}(\mathcal{D}) \to \mathcal{P}(\mathcal{D})$

Conditions $\mathbb{F}[\![\, c_1 \parallel c_2 \,]\!] \in \mathcal{P}(\mathcal{D}) \to \mathcal{P}(\mathcal{D})$

$$\mathbb{D}[\![\, \textbf{while } c_1 \parallel c_2 \textbf{ do } s \,]\!]X \triangleq \mathbb{F}[\![\, \neg c_1 \parallel \neg c_2 \,]\!](\mathrm{lfp}\ H^X)$$

Lifting simple program semantics to double programs

while statement

Double program P

Double states in $\mathcal{D} \triangleq \mathcal{E} \times \mathcal{E}$

Semantics $\mathbb{D}[\![\, s \,]\!] \in \mathcal{P}(\mathcal{D}) \to \mathcal{P}(\mathcal{D})$

Conditions $\mathbb{F}[\![\, c_1 \parallel c_2 \,]\!] \in \mathcal{P}(\mathcal{D}) \to \mathcal{P}(\mathcal{D})$

$$\mathbb{D}[\![\, \textbf{while } c_1 \parallel c_2 \textbf{ do } s \,]\!]X \triangleq \mathbb{F}[\![\, \neg c_1 \parallel \neg c_2 \,]\!](\text{lfp } H^X)$$

$$H^X(Y) \triangleq X \cup \begin{pmatrix} \mathbb{D}[\![\, s \,]\!] \circ \mathbb{F}[\![\, c_1 \parallel c_2 \,]\!]Y \cup \\ \mathbb{D}[\![\, \pi_1(s) \parallel \textbf{skip} \,]\!] \circ \mathbb{F}[\![\, c_1 \parallel \neg c_2 \,]\!]Y \cup \\ \mathbb{D}[\![\, \textbf{skip} \parallel \pi_2(s) \,]\!] \circ \mathbb{F}[\![\, \neg c_1 \parallel c_2 \,]\!]Y \end{pmatrix}$$

Construct a double program from a pair of program versions

First merge identical statements

```
$first \leftarrow \textbf{input}(0, 100);$
$last \leftarrow \textbf{input}(0, 100);$
$break \leftarrow \text{false};$




$i \leftarrow 0;$
$\textbf{while}\ (\neg break)\ \{$
  $x \leftarrow first + i \times 2;$
  $\textbf{if}\ (last < x)$
  $\textbf{then}\ break \leftarrow \text{true}$
  $\textbf{else}\ r \leftarrow x;$

  $i \leftarrow i + 1$
$\}$

$\textbf{output}(r)$
```

```
$first \leftarrow \textbf{input}(0, 100);$
$last \leftarrow \textbf{input}(0, 100);$
$break \leftarrow \text{false};$
$out \leftarrow (last < first);$
$\textbf{if}\ (\neg out)\ \{$
  $x \leftarrow first;$
  $i \leftarrow 1;$
  $\textbf{while}\ (\neg break)\ \{$
    $r \leftarrow x;$
    $\textbf{if}\ (out)$
    $\textbf{then}\ break \leftarrow \text{true}$
    $\textbf{else}\ \{\ x \leftarrow first + i \times 2;\ \ out \leftarrow (last < x);$
          $\textbf{if}\ (out \wedge \neg more)\ \textbf{then}\ break \leftarrow \text{true}\ \};$
    $i \leftarrow i + 1$
  $\}$
$\}$
$\textbf{output}(r)$
```

Construct a double program from a pair of program versions

Then align similar control structures

```
$first \leftarrow$ input$(0, 100)$;
$last \leftarrow$ input$(0, 100)$;
$break \leftarrow$ false;
                                   ||   $out \leftarrow (last < first)$;
                                   ||   if $(\neg out)$ {
                                   ||     $x \leftarrow first$;
$i \leftarrow 0$;                  ||     $i \leftarrow 1$;
while $(\neg break)$ {             ||     while $(\neg break)$ {
  $x \leftarrow first + i \times 2$;  ||       $r \leftarrow x$;
  if $(last < x)$                  ||       if $(out)$
  then $break \leftarrow$ true     ||       then $break \leftarrow$ true
  else $r \leftarrow x$;           ||       else { $x \leftarrow first + i \times 2$;  $out \leftarrow (last < x)$;
                                   ||              if $(out \wedge \neg more)$ then $break \leftarrow$ true };
  $i \leftarrow i + 1$             ||       $i \leftarrow i + 1$
}                                  ||     }
                                   ||   }
output$(r)$
```

Construct a double program from a pair of program versions

Then align similar control structures

```
$first \leftarrow$ input$(0, 100)$;
$last \leftarrow$ input$(0, 100)$;
$break \leftarrow$ false;
$i \leftarrow 0$; || $out \leftarrow (last < first)$;
                                   ||   if $(\neg out)$ {
                                   ||     $x \leftarrow first$;
                                   ||     $i \leftarrow 1$;
while $(\neg break)$ {             ||     while $(\neg break)$ {
  $x \leftarrow first + i \times 2$;     ||       $r \leftarrow x$;
  if $(last < x)$                  ||       if $(out)$
  then $break \leftarrow$ true         ||       then $break \leftarrow$ true
  else $r \leftarrow x$;               ||       else { $x \leftarrow first + i \times 2$;  $out \leftarrow (last < x)$;
                                   ||                if $(out \wedge \neg more)$ then $break \leftarrow$ true };
  $i \leftarrow i + 1$                 ||       $i \leftarrow i + 1$
}                                  ||     }
                                   ||   }
output$(r)$
```

Construct a double program from a pair of program versions

Then align similar control structures using simple program transformations

```
                         $first \leftarrow$ input$(0, 100)$;
                         $last \leftarrow$ input$(0, 100)$;
                         $break \leftarrow$ false;
                         $i \leftarrow 0;$ ∥ $out \leftarrow (last < first)$;
if (true) {                              ∥   if ($\neg out$) {
                                         ∥     $x \leftarrow first$;
                                         ∥     $i \leftarrow 1$;
  while ($\neg break$) {                 ∥     while ($\neg break$) {
    $x \leftarrow first + i \times 2$;   ∥       $r \leftarrow x$;
    if ($last < x$)                      ∥       if ($out$)
    then $break \leftarrow$ true         ∥       then $break \leftarrow$ true
    else $r \leftarrow x$;               ∥       else { $x \leftarrow first + i \times 2$; $out \leftarrow (last < x)$;
                                         ∥              if ($out \wedge \neg more$) then $break \leftarrow$ true };
    $i \leftarrow i + 1$                 ∥       $i \leftarrow i + 1$
  }                                      ∥     }
}                                        ∥   }
                         output$(r)$
```

Construct a double program from a pair of program versions

The double program obtained allows for successful patch analysis with linear invariants

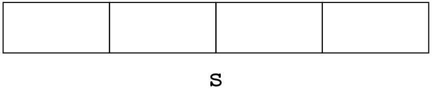
```
$\mathit{first} \leftarrow \textbf{input}(0, 100);$
$\mathit{last} \leftarrow \textbf{input}(0, 100);$
$\mathit{break} \leftarrow \text{false};$
$i \leftarrow 0 \quad \| \; \mathit{out} \leftarrow (\mathit{last} < \mathit{first});$
$\textbf{if}\ (\text{true} \;\|\; \neg \mathit{out})\ \{$
    $\textbf{skip} \quad \| \quad x \leftarrow \mathit{first};$
                 $i \leftarrow 1;$
    $\textbf{while}\ (\neg \mathit{break})\ \{$
        $x \leftarrow \mathit{first} + i \times 2 \quad \| \quad r \leftarrow x;$
        $\textbf{if}\ (\mathit{last} < x \;\|\; \mathit{out})$
        $\textbf{then}\ \mathit{break} \leftarrow \text{true}$
        $\textbf{else}$
            $r \leftarrow x \quad \| \quad x \leftarrow \mathit{first} + i \times 2;\ \mathit{out} \leftarrow (\mathit{last} < x);$
                           $\textbf{if}\ (\mathit{out} \wedge \neg \mathit{more})\ \textbf{then}\ \mathit{break} \leftarrow \text{true}$
        $i \leftarrow i + 1$
    $\}$
$\}$
$\textbf{output}(r)$
```

Agenda

1. Introduction
2. Patch analysis for numerical programs
3. Patch analysis for C and structure layout portability
4. Endian portability analysis for C programs
5. Conclusion

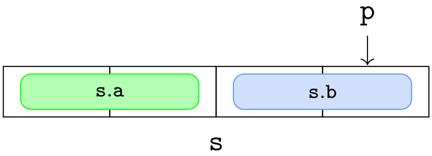
Low-level C programs

```
struct { u16 a; u16 b; } s;

s.b = input(0,1000);

u8 *p = (u8 *) &s + 1;

p += sizeof(u16);

output(*p);
```



Low-level C programs

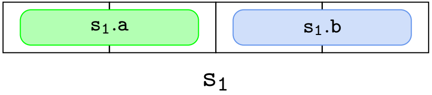
```
struct { u16 a; u16 b; } s;

s.b = input(0,1000);

u8 *p = (u8 *) &s + 1;

p += sizeof(u16);

output(*p);
```



Low-level C programs

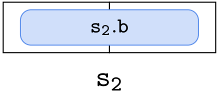
Patching a C data structure

```
struct { u16 a; u16 b; } s;

s.b = input(0,1000);

u8 *p = (u8 *) &s + 1;

p += sizeof(u16);

output(*p);
```



Removing unused field a

```
struct { ~~u16 a;~~ u16 b; } s;

s.b = input(0,1000);

u8 *p = (u8 *) &s + 1;

p += sizeof(u16);

output(*p);
```



Low-level C programs

Patching a C data structure

```
struct { u16 a; u16 b; } s;

s.b = input(0,1000); ●

u8 *p = (u8 *) &s + 1;

p += sizeof(u16);

output(*p);
```

		0	1

s_1

Removing unused field a

```
struct { ~~u16 a;~~ u16 b; } s;

s.b = input(0,1000); ●

u8 *p = (u8 *) &s + 1;

p += sizeof(u16);

output(*p);
```

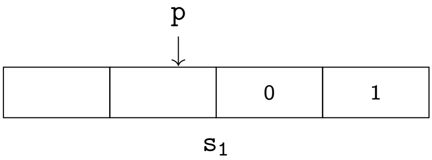
0	1

s_2

Low-level C programs

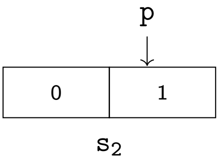
Patching a C data structure

```
struct { u16 a; u16 b; } s;

s.b = input(0,1000);

u8 *p = (u8 *) &s + 1; ●

p += sizeof(u16);

output(*p);
```



Removing unused field a

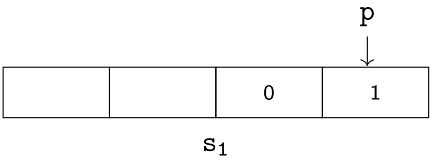
```
struct { ~~u16 a;~~ u16 b; } s;

s.b = input(0,1000);

u8 *p = (u8 *) &s + 1; ●

p += sizeof(u16);

output(*p);
```



Low-level C programs

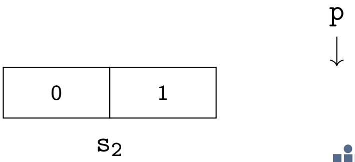
Patching a C data structure

```
struct { u16 a; u16 b; } s;

s.b = input(0,1000);

u8 *p = (u8 *) &s + 1;

p += sizeof(u16); ●

output(*p);
```



Removing unused field a

```
struct { ~~u16 a;~~ u16 b; } s;

s.b = input(0,1000);

u8 *p = (u8 *) &s + 1;

p += sizeof(u16); ●

output(*p);
```



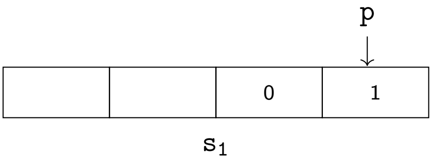
Low-level C programs

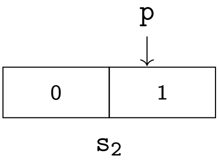
Patching a C data structure

Removing unused field a

```
struct { u16 a; u16 b; } s;

s.b = input(0,1000);

u8 *p = (u8 *) &s + 1;

p += sizeof(u16); ●

output(*p);
```

```
struct { ~~u16 a;~~ u16 b; } s;

s.b = input(0,1000);

u8 *p = (u8 *) &s + 1;

~~p += sizeof(u16);~~

output(*p);
```





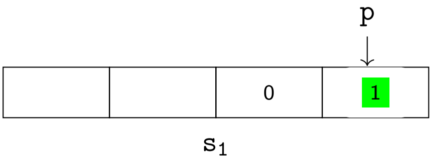
Low-level C programs

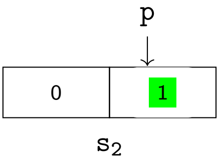
Patching a C data structure

```
struct { u16 a; u16 b; } s;

s.b = input(0,1000);

u8 *p = (u8 *) &s + 1;

p += sizeof(u16);

output(*p); ●
```

Removing unused field a

```
struct { ~~u16 a;~~ u16 b; } s;

s.b = input(0,1000);

u8 *p = (u8 *) &s + 1;

~~p += sizeof(u16);~~

output(*p); ●
```

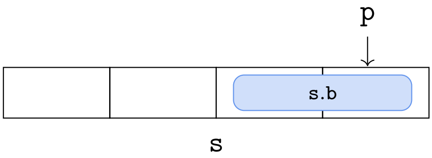




Low-level C programs

The Cell memory model

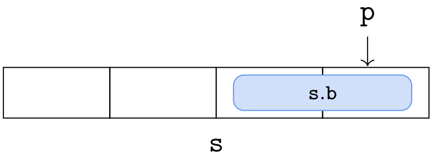
```
struct { u16 a; u16 b; } s;

s.b = input(0,1000);

u8 *p = (u8 *) &s + 1;

p += sizeof(u16);

output(*p);
```



Low-level C programs

The Cell memory model

```
struct { u16 a; u16 b; } s;

s.b = input(0,1000);

u8 *p = (u8 *) &s + 1;

p += sizeof(u16);

output(*p);
```



Memory model

- Concrete level

  the program holds values for individual bytes

- Low-level C programs

  multi-byte access to memory
  numerical invariants } ⇒ need for scalar cells

  byte-level access to encoding
  abuse unions and pointers } ⇒ cells may overlap

Low-level C programs

The Cell memory model

```
struct { u16 a; u16 b; } s;

s.b = input(0,1000);

u8 *p = (u8 *) &s + 1;

p += sizeof(u16);

output(*p);
```

Memory model

- Concrete level

  the program holds values for individual bytes

- Low-level C programs

  $\left.\begin{array}{l}\text{multi-byte access to memory}\\ \text{numerical invariants}\end{array}\right\} \Rightarrow$ need for scalar cells

  $\left.\begin{array}{l}\text{byte-level access to encoding}\\ \text{abuse unions and pointers}\end{array}\right\} \Rightarrow$ cells may overlap

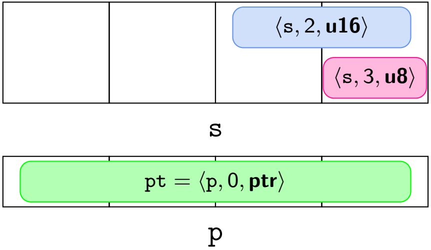
The Cells abstract domain — Miné [2006a, 2013]

- Memory as a dynamic collection of cells
  - synthetic scalar variables $\langle V, o, \tau \rangle \in \mathit{Cell} \subseteq \mathcal{V} \times \mathbb{N} \times \mathit{scalar\text{-}type}$
  - holding values for memory dereferences discovered during analysis
- Analysis with numerical domain (1 dimension / cell)

Low-level C programs

The Cell memory model

```
struct { u16 a; u16 b; } s;

s.b = input(0,1000);

u8 *p = (u8 *) &s + 1;

p += sizeof(u16);

output(*p);
```


$\langle \mathrm{s}, 2, \mathbf{u16} \rangle$
$\langle \mathrm{s}, 3, \mathbf{u8} \rangle$
s
$\mathrm{pt} = \langle \mathrm{p}, 0, \mathbf{ptr} \rangle$
p

The Cells abstract domain — Miné [2006a, 2013]

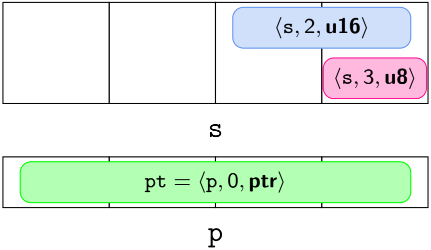
- Memory as a dynamic collection of cells
  - synthetic scalar variables $\langle V, o, \tau \rangle \in \mathit{Cell} \subseteq \mathcal{V} \times \mathbb{N} \times \mathit{scalar\text{-}type}$
  - holding values for memory dereferences discovered during analysis
- Analysis with numerical domain (1 dimension / cell)

Low-level C programs

The Cell memory model

$byte(n,k) = \lfloor n/2^{8k} \rfloor \mod 2^8$

```
struct { u16 a; u16 b; } s;

s.b = input(0,1000);

u8 *p = (u8 *) &s + 1;

p += sizeof(u16);

output(*p);
```


$\langle \mathrm{s}, 2, \mathbf{u16} \rangle$
$\langle \mathrm{s}, 3, \mathbf{u8} \rangle$
s
$\mathrm{pt} = \langle \mathrm{p}, 0, \mathbf{ptr} \rangle$
p

🟦 $\in [0, 1000]$

🟪 $= byte($🟦$, 1)$

The Cells abstract domain — Miné [2006a, 2013]

- Memory as a dynamic collection of cells
  - synthetic scalar variables $\langle V, o, \tau \rangle \in \mathcal{C}ell \subseteq \mathcal{V} \times \mathbb{N} \times scalar\text{-}type$
  - holding values for memory dereferences discovered during analysis
- Analysis with numerical domain (1 dimension / cell)

Patch analysis for low-level C programs

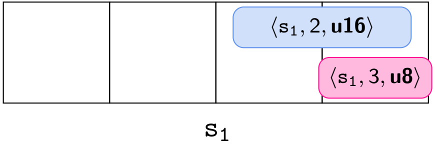
Lifting the Cell memory model

```
struct { u16 a; u16 b; } s; ||
struct {        u16 b; } s;

s.b = input(0,1000);

u8 *p = (u8 *) &s + 1;

p+=sizeof(u16) || skip;

output(*p);
```

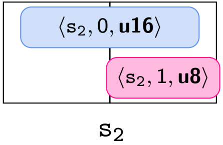
Patch analysis for low-level C programs

Lifting the Cell memory model

```
struct { u16 a; u16 b; } s; ‖
struct {        u16 b; } s;

s.b = input(0,1000);

u8 *p = (u8 *) &s + 1;

p+=sizeof(u16) ‖ skip;

output(*p);
```



$\langle s_1, 2, \mathbf{u16} \rangle$
$\langle s_1, 3, \mathbf{u8} \rangle$
s_1

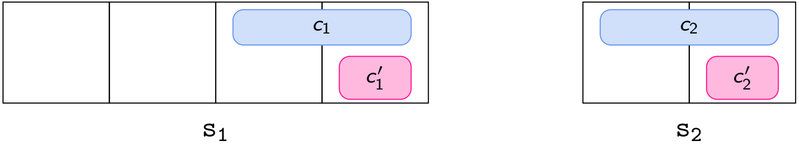


$\langle s_2, 0, \mathbf{u16} \rangle$
$\langle s_2, 1, \mathbf{u8} \rangle$
s_2

Patch analysis for low-level C programs

Lifting the Cell memory model

```
struct { u16 a; u16 b; } s; ||
struct {        u16 b; } s;

s.b = input(0,1000);

u8 *p = (u8 *) &s + 1;

p+=sizeof(u16) || skip;

output(*p);
```



Patch analysis for low-level C programs

Lifting the Cell memory model

$byte(n,k) = \lfloor n/2^{8k} \rfloor \mod 2^8$

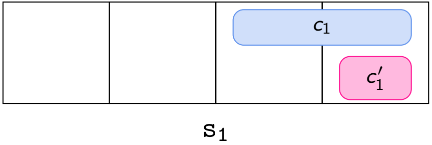
```
struct { u16 a; u16 b; } s; ‖
struct {        u16 b; } s;

s.b = input(0,1000);

u8 *p = (u8 *) &s + 1;

p+=sizeof(u16) ‖ skip;

output(*p);
```

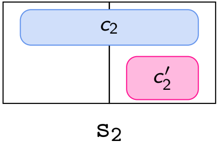
Program invariants and cell constraints

$c_1 = c_2 \in [0, 1000]$

$c_1' = byte(c_1, 1)$
$c_2' = byte(c_2, 1)$

$\boldsymbol{c_1' \overset{?}{=} c_2'}$





Optimizing the memory model for the common case

Complex invariants $\Longrightarrow$ expressive numerical domain?

- Program invariants and cell constraints

$$\left.\begin{array}{l} c_1' = \lfloor c_1/2^8 \rfloor \mod 2^8 \\ c_2' = \lfloor c_2/2^8 \rfloor \mod 2^8 \end{array}\right\} \wedge \quad c_1 = c_2 \quad \Longrightarrow \quad \boldsymbol{c_1' = c_2'}$$

- Common case: most multi-byte cells hold **equal values** in the memories of P_1 and P_2

Optimizing the memory model for the common case

Complex invariants $\Longrightarrow$ expressive numerical domain?

- Program invariants and cell constraints

$$\left.\begin{array}{l} c'_1 = \lfloor c_1/2^8 \rfloor \mod 2^8 \\ c'_2 = \lfloor c_2/2^8 \rfloor \mod 2^8 \end{array}\right\} \wedge \quad c_1 = c_2 \quad \Longrightarrow \quad \boldsymbol{c'_1 = c'_2}$$

- Common case: most multi-byte cells hold **equal values** in the memories of P_1 and P_2

Sharing cells in the memory environment

- **Single** representation for **two** cells
  - from **different** program **versions**
  - holding **equal values**
- A bi-cell is $\mathcal{B}icell \triangleq \widetilde{Cell} \cup (\widetilde{Cell} \times \widetilde{Cell})$

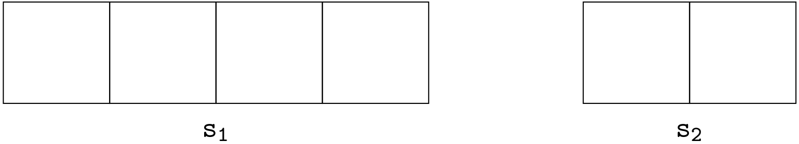
either a single cell $\widetilde{Cell} \triangleq Cell_1 \uplus Cell_2$

or a pair of cells holding equal values (shared bi-cell)

Patch analysis for low-level C programs

From cells to bi-cells

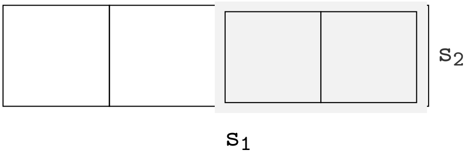
```
struct { u16 a; u16 b; } s; ‖
struct {        u16 b; } s;

s.b = input(0,1000);

u8 *p = (u8 *) &s + 1;

p+=sizeof(u16) ‖ skip;

output(*p);
```



Patch analysis for low-level C programs

From cells **to bi-cells**

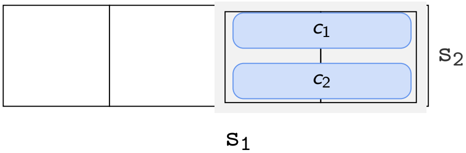
```
struct { u16 a; u16 b; } s; ‖
struct {        u16 b; } s;

s.b = input(0,1000);

u8 *p = (u8 *) &s + 1;

p+=sizeof(u16) ‖ skip;

output(*p);
```



Patch analysis for low-level C programs

From cells **to bi-cells**

```
struct { u16 a; u16 b; } s; ‖
struct {        u16 b; } s;

s.b = input(0,1000); ●

u8 *p = (u8 *) &s + 1;

p+=sizeof(u16) ‖ skip;

output(*p);
```



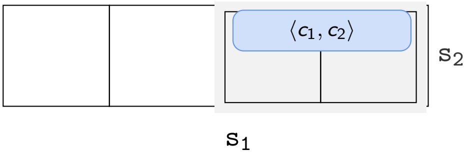
Program invariants and bi-cell constraints

$c_1 \overset{?}{=} c_2$

Patch analysis for low-level C programs

From cells **to bi-cells**

```
struct { u16 a; u16 b; } s; ‖
struct {        u16 b; } s;

s.b = input(0,1000); ●

u8 *p = (u8 *) &s + 1;

p+=sizeof(u16) ‖ skip;

output(*p);
```



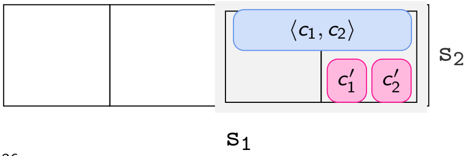
Program invariants and bi-cell constraints

$\langle c_1, c_2 \rangle \in \ [0, 1000]$

Patch analysis for low-level C programs

From cells **to bi-cells**

```
struct { u16 a; u16 b; } s; ||
struct {        u16 b; } s;

s.b = input(0,1000);

u8 *p = (u8 *) &s + 1;

p+=sizeof(u16) || skip;

output(*p); ●
```



Program invariants and bi-cell constraints

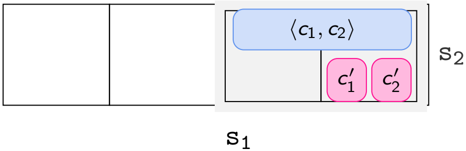
$\langle c_1, c_2\rangle \in [0, 1000]$

$c_1' \overset{?}{=} c_2'$

Patch analysis for low-level C programs

From cells **to bi-cells**

```
struct { u16 a; u16 b; } s; ‖
struct {        u16 b; } s;

s.b = input(0,1000);

u8 *p = (u8 *) &s + 1;

p+=sizeof(u16) ‖ skip;

output(*p); ●
```



Program invariants and bi-cell constraints

$$\langle c_1, c_2 \rangle \in [0, 1000]$$

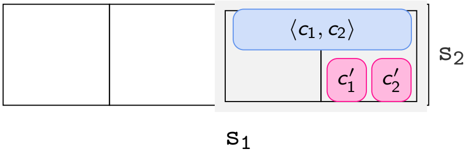
$$c'_1 \overset{?}{=} c'_2$$

Shared bi-cell synthesis

Patch analysis for low-level C programs

From cells **to bi-cells**

```
struct { u16 a; u16 b; } s; ‖
struct {        u16 b; } s;

s.b = input(0,1000);

u8 *p = (u8 *) &s + 1;

p+=sizeof(u16) ‖ skip;

output(*p); ●
```



Program invariants and bi-cell constraints

$$\langle c_1, c_2\rangle \in [0, 1000]$$

$$c_1' \overset{?}{=} c_2'$$

Shared bi-cell synthesis

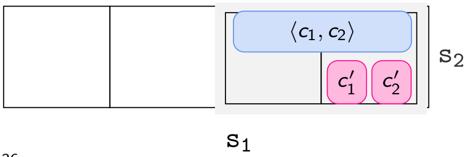
$\exists\langle c_1', c_2'\rangle$? ✗

Patch analysis for low-level C programs

From cells **to bi-cells**

$byte(n,k) = \lfloor n/2^{8k} \rfloor \mod 2^8$

```
struct { u16 a; u16 b; } s; ‖
struct {        u16 b; } s;

s.b = input(0,1000);

u8 *p = (u8 *) &s + 1;

p+=sizeof(u16) ‖ skip;

output(*p); ●
```



Program invariants and bi-cell constraints

$\langle c_1, c_2\rangle \in [0, 1000]$

$c'_1 \overset{?}{=} c'_2$

Shared bi-cell synthesis

$\exists\langle c'_1, c'_2\rangle$? ✗

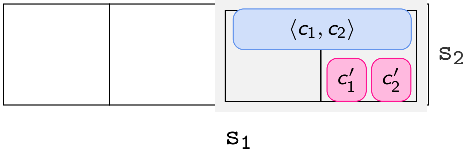
$\forall\rho : \rho(c'_1) = \rho(c'_2)$? $ > polyhedra

$$c'_1 = byte(c_1, 1)$$
$$c'_2 = byte(c_2, 1)$$

Patch analysis for low-level C programs

From cells **to bi-cells**

```
struct { u16 a; u16 b; } s; ‖
struct {        u16 b; } s;

s.b = input(0,1000);

u8 *p = (u8 *) &s + 1;

p+=sizeof(u16) ‖ skip;

output(*p); ●
```



Program invariants and bi-cell constraints

$\langle c_1, c_2\rangle \in \; [0, 1000]$

$c'_1 \overset{?}{=} c'_2$

Shared bi-cell synthesis

$\exists\langle c'_1, c'_2\rangle$? ✗

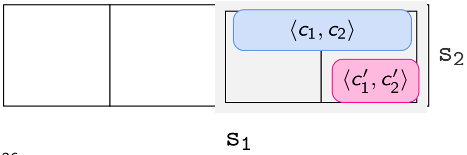
$\forall\rho : \; \rho(c'_1) = \rho(c'_2)$? $\$$ $>$ polyhedra

$$\left.\begin{array}{l}\exists(x_1, x_2, o) : \; x_1 = x_2 \; \wedge \\ c'_i \text{ at offset } o \text{ inside } x_i\end{array}\right\} ? \checkmark \; \begin{cases} x_i = c_i \\ o = 1 \end{cases}$$

Patch analysis for low-level C programs

From cells **to bi-cells**

```
struct { u16 a; u16 b; } s; ‖
struct {        u16 b; } s;

s.b = input(0,1000);

u8 *p = (u8 *) &s + 1;

p+=sizeof(u16) ‖ skip;

output(*p); ●
```



Program invariants and bi-cell constraints

$\langle c_1, c_2 \rangle \in \; [0, 1000]$

Shared bi-cell synthesis

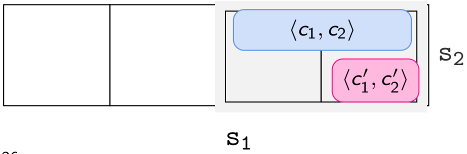
$\langle c_1', c_2' \rangle$ synthesized by pattern-matching

Patch analysis for low-level C programs

From cells **to bi-cells**

$byte(n, k) = \lfloor n/2^{8k} \rfloor \mod 2^8$

```
struct { u16 a; u16 b; } s; ||
struct {        u16 b; } s;

s.b = input(0,1000);

u8 *p = (u8 *) &s + 1;

p+=sizeof(u16) || skip;

output(*p); ●
```



Program invariants and bi-cell constraints

$$\langle c_1, c_2 \rangle \in [0, 1000]$$

$$\langle c_1', c_2' \rangle = byte(\langle c_1, c_2 \rangle, 1)$$

Shared bi-cell synthesis

Implementation

on top of Mopsa

MOPSA
analyzer

http://mopsa.lip6.fr/

Mopsa platform

- Modular development
- Precise static analyses
- Multiple languages
- Multiple properties

Prototype abstract interpreter

$\simeq$ 6,700 lines of OCaml source code

50%	**bi-cell** based memory **abstraction**
33%	double program **construction**
17%	double program **iterators** and utilities

The Mopsa leverage effect

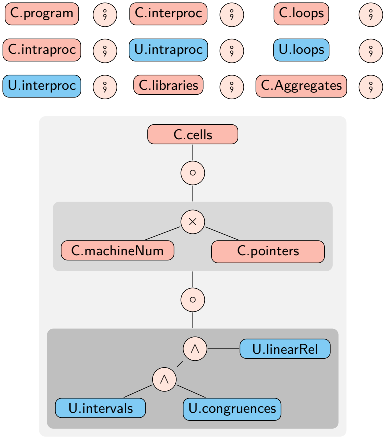
$\simeq$ 50,000 lines of Mopsa leveraged

38%	**parsers** and utilities
27%	common **framework** **iterators** and numeric **domains**

24%	specific for the C language
11%	generic for of all languages

Implementation

Analysis of C **programs** with **cells**



C.program ⨾ C.interproc ⨾ C.loops ⨾

C.intraproc ⨾ U.intraproc ⨾ U.loops ⨾

U.interproc ⨾ C.libraries ⨾ C.Aggregates ⨾

C.cells

∘

×

C.machineNum

C.pointers

∘

∧

U.linearRel

∧

U.intervals

U.congruences

⨾ Sequence

∧ Reduced product

× Cartesian product

∘ Composition

🔵 Universal

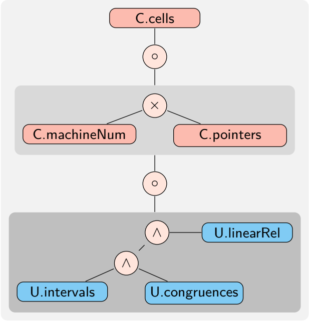
🔴 C specific

🟢 Double C

Implementation

Analysis of C **programs** with **cells**

C.program ⨟ C.interproc ⨟ C.loops ⨟
C.intraproc ⨟ U.intraproc ⨟ U.loops ⨟
U.interproc ⨟ C.libraries ⨟ C.Aggregates ⨟



⨟ Sequence

∧ Reduced product

× Cartesian product

∘ Composition

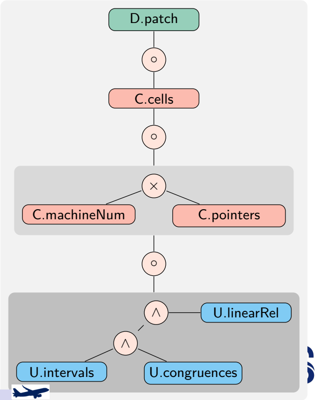
🔵 Universal

🔴 C specific

🟢 Double C

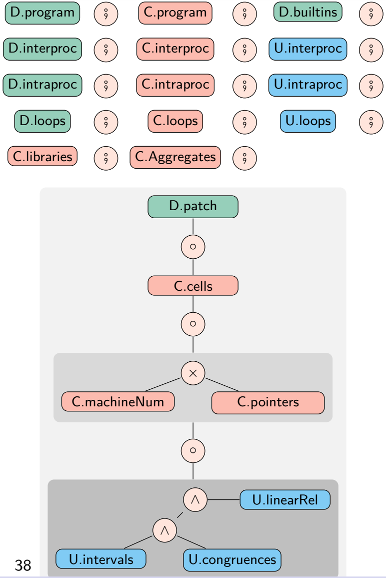
Analysis of C **patches** with **cells**

D.program ⨟ C.program ⨟ D.builtins ⨟
D.interproc ⨟ C.interproc ⨟ U.interproc ⨟
D.intraproc ⨟ C.intraproc ⨟ U.intraproc ⨟
D.loops ⨟ C.loops ⨟ U.loops ⨟
C.libraries ⨟ C.Aggregates ⨟



Implementation

Analysis of C **patches** with **cells**


D.program ⨾ C.program ⨾ D.builtins ⨾
D.interproc ⨾ C.interproc ⨾ U.interproc ⨾
D.intraproc ⨾ C.intraproc ⨾ U.intraproc ⨾
D.loops ⨾ C.loops ⨾ U.loops ⨾
C.libraries ⨾ C.Aggregates ⨾
D.patch
∘
C.cells
∘
×
C.machineNum
C.pointers
∘
∧
U.linearRel
∧
U.intervals
U.congruences

⨾ Sequence

∧ Reduced product

× Cartesian product

∘ Composition

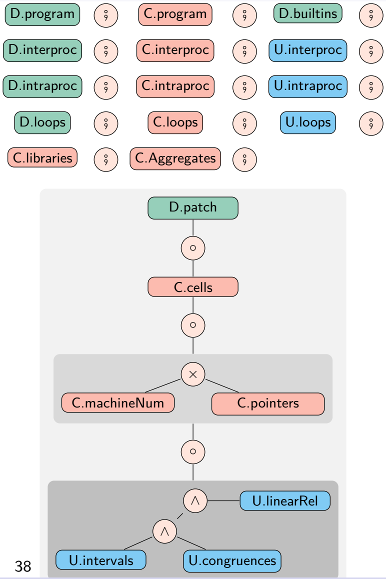
🟦 Universal

🟥 C specific

🟩 Double C

Implementation

Analysis of C **patches** with **cells**



D.program ⨾ C.program ⨾ D.builtins ⨾

D.interproc ⨾ C.interproc ⨾ U.interproc ⨾

D.intraproc ⨾ C.intraproc ⨾ U.intraproc ⨾

D.loops ⨾ C.loops ⨾ U.loops ⨾

C.libraries ⨾ C.Aggregates ⨾

D.patch

∘

C.cells

∘

×

C.machineNum

C.pointers

∘

∧

U.linearRel

∧

U.intervals

U.congruences

⨾ Sequence

∧ Reduced product

× Cartesian product

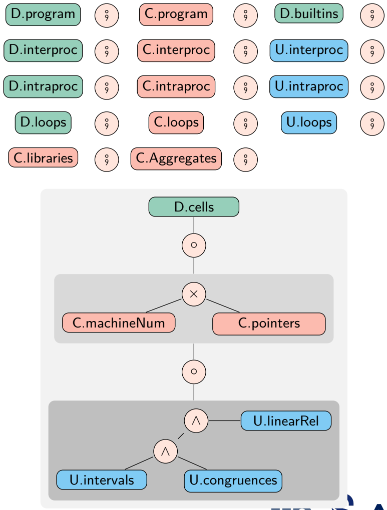
∘ Composition

🟦 Universal

🟥 C specific

🟩 Double C

Analysis of C **patches** with **bi-cells**



D.program ⨾ C.program ⨾ D.builtins ⨾

D.interproc ⨾ C.interproc ⨾ U.interproc ⨾

D.intraproc ⨾ C.intraproc ⨾ U.intraproc ⨾

D.loops ⨾ C.loops ⨾ U.loops ⨾

C.libraries ⨾ C.Aggregates ⨾

D.cells

∘

×

C.machineNum

C.pointers

∘

∧

U.linearRel

∧

U.intervals

U.congruences

Related works

Semantic patch analysis

Related work	Tool	Characteristics	Our approach
Symbolic execution Trostanetski et al. [2017]	ModDiff	Full path enumeration	Approximate fixpoint computation
Deductive methods Godlin and Strichman [2009] Lahiri et al. [2012] and Klebanov et al. [2018]	RVT SymDiff Rêve	SMT solvers	Abstract domains
Abstract interpretation Partush and Yahav [2013] Partush and Yahav [2014]	Dizy Score	Program transformation → correlating program speculative correlation	Concrete collecting semantics for double programs double program construction

Evaluation

Synthetic or simplified benchmarks from the related works

	Benchmark	LOC	#P	Related	Cell based abstraction				Bi-cell based abstraction					
				time	polyhedra		octagon		polyhedra		octagon		interval	
ModDiff	Comp	13	2	539 ms	48 ms	✓		✗	107 ms	✓	209 ms	✓		✗
	Const	9	3	541 ms	28 ms	✓		✗	38 ms	✓	49 ms	✓		✗
	Fig. 2	14	1	–	31 ms	✓	39 ms	✓	40 ms	✓	47 ms	✓	25 ms	✓
	LoopMult	14	2	49 s	166 ms	✓		✗	367 ms	✓		✗		✗
	LoopSub	15	2	1.2 s	60 ms	✓		✗	74 ms	✓		✗		✗
	UnchLoop	13	2	2.8 s[1]	69 ms	✓		✗	71 ms	✓		✗		✗
Rêve	loop	11	3	50 ms	43 ms	✓		✗	52 ms	✓		✗		✗
	while-if	11	3	80 ms	66 ms	✓	156 ms	✓	66 ms	✓	97 ms	✓		✗
	digits10	24	19	1.12 s	312 ms	✓		✗	207 ms	✓	313 ms	✓	47 ms	✓
	barthe	13	2	120 ms	93 ms	✓		✗	69 ms	✓		✗		✗
	barthe2	11	2	150 ms	81 ms	✓		✗	79 ms	✓		✗		✗
Score/Dizy	sign	12	2	–	29 ms	✓		✗	33 ms	✓		✗		✗
	sum	14	4	4 s	71 ms	✓		✗	162 ms	✓	349 ms	✓		✗
	copy[2]	37	1	2 s	132 ms	✓	373 ms	✓	156 ms	✓	189 ms	✓	30 ms	✓
	seq[2]	41	13	11 s	293 ms	✓		✗	326 ms	✓		✗		✗
	pr[2]	111	8	1149 s	2.686 s	✓	11.672 s	✓	4.410 s	✓	3.487 s	✓	87 ms	✓

[1] only 5 loop iterations

[2] Coreutils (simplified code)

Evaluation

Real patches from Coreutils and Linux

	Bench.	LOC	#P	Cell based abstraction		Bi-cell based abstraction		
				polyhedra	octagon	polyhedra	octagon	interval
Coreutils	copy	95	1	157 ms ✓	482 ms ✓	113 ms ✓	156 ms ✓	41 ms ✓
	seq	46	16	570 ms ✓	✗	442 ms ✓	✗	✗
	pr	114	8	1.421 s ✓	6.469 s ✓	4.642 s ✓	3.723 s ✓	88 ms ✓
	test	352	10	9.188 s ✓	✗	440 ms ✓	1.163 s ✓	96 ms ✓
Linux	kvm	248	1/11	2.707 s ✓	4.214 s ✓	1.426 s ✓	1.568 s ✓	96 ms ✓
	sched	194	7/12	65 ms ✓	✗	63 ms ✓	104 ms ✓	38 ms ✓
	dma	270	5/23	285 ms ✓	1.235 s ✓	216 ms ✓	584 ms ✓	76 ms ✓
	block	324	22/6	80 ms ✓	✗	67 ms ✓	121 ms ✓	31 ms ✓
	iucv	179	10/9	403 ms ✓	1.757 s ✓	7.721 s ✓	14.423 s ✓	426 ms ✓
	io_uring	1569	10/14	868.701 s ✓	✗	594.481 s ✓	4170.295s ✓	288 ms ✓

[2] simplified Coreutils benchmarks from Score/Dizy

Evaluation

Real patches from Coreutils and Linux

	Bench.	LOC	#P	Cell based abstraction				Bi-cell based abstraction					
				polyhedra		octagon		polyhedra		octagon		interval	
Coreutils	copy	95	1	157 ms	✓	482 ms	✓	113 ms	✓	156 ms	✓	41 ms	✓
	copy[2]	37	1	132 ms	✓	373 ms	✓	156 ms	✓	189 ms	✓	30 ms	✓
	seq	46	16	570 ms	✓		✗	442 ms	✓		✗		✗
	seq[2]	41	13	293 ms	✓		✗	326 ms	✓		✗		✗
	pr	114	8	1.421 s	✓	6.469 s	✓	4.642 s	✓	3.723 s	✓	88 ms	✓
	pr[2]	111	8	2.686 s	✓	11.672 s	✓	4.410 s	✓	3.487 s	✓	87 ms	✓
	test	352	10	9.188 s	✓		✗	440 ms	✓	1.163 s	✓	96 ms	✓
Linux	kvm	248	1/11	2.707 s	✓	4.214 s	✓	1.426 s	✓	1.568 s	✓	96 ms	✓
	sched	194	7/12	65 ms	✓		✗	63 ms	✓	104 ms	✓	38 ms	✓
	dma	270	5/23	285 ms	✓	1.235 s	✓	216 ms	✓	584 ms	✓	76 ms	✓
	block	324	22/6	80 ms	✓		✗	67 ms	✓	121 ms	✓	31 ms	✓
	iucv	179	10/9	403 ms	✓	1.757 s	✓	7.721 s	✓	14.423 s	✓	426 ms	✓
	io_uring	1569	10/14	868.701 s	✓		✗	594.481 s	✓	4170.295s	✓	288 ms	✓

[2] simplified Coreutils benchmarks from SCORE/DIZY

Agenda

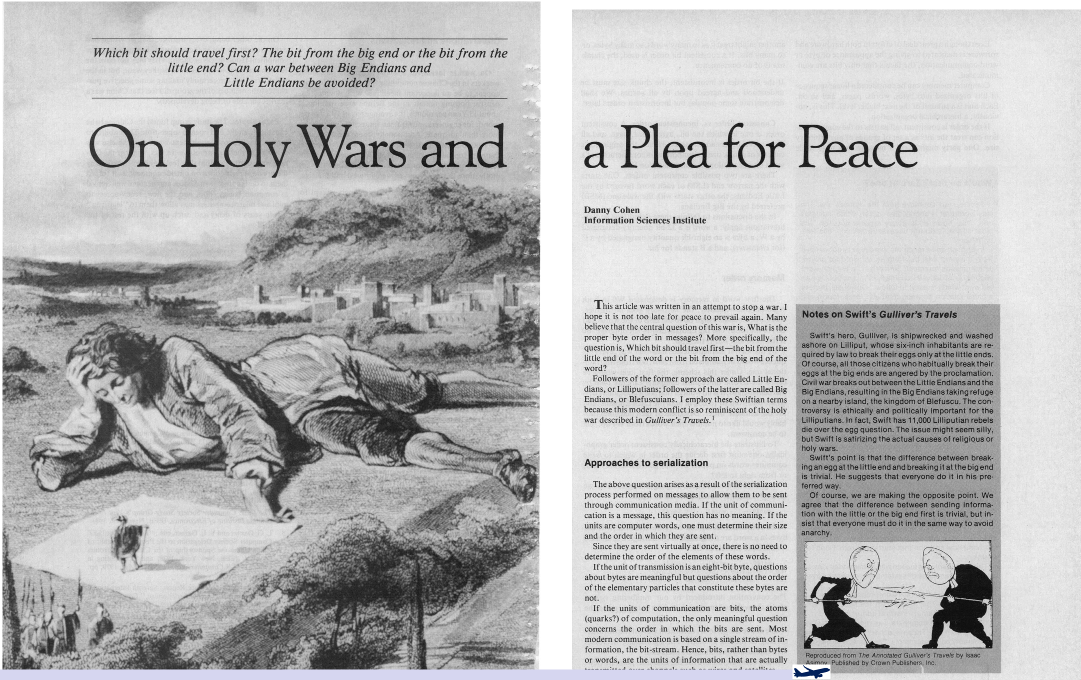
1. Introduction
2. Patch analysis for numerical programs
3. Patch analysis for C and structure layout portability
4. **Endian portability analysis for C programs**
5. Conclusion

Endianness

No consensus

Representation of multi-byte scalar values in memory

- Little-endian systems
  - least-significant byte at lowest address
  - Intel processors
- Big-endian systems
  - least-significant byte at highest address
  - internet protocols, legacy or embedded processors (e.g. SPARC, PowerPC)



Which bit should travel first? The bit from the big end or the bit from the little end? Can a war between Big Endians and Little Endians be avoided?

On Holy Wars and a Plea for Peace

Danny Cohen
Information Sciences Institute

This article was written in an attempt to stop a war. I hope it is not too late for peace to prevail again. Many believe that the central question of this war is, What is the proper byte order in messages? More specifically, the question is, Which bit should travel first—the bit from the little end of the word or the bit from the big end of the word?

Followers of the former approach are called Little Endians, or Lilliputians; followers of the latter are called Big Endians, or Blefuscuians. I employ these Swiftian terms because this modern conflict is so reminiscent of the holy war described in *Gulliver's Travels.*[1]

Approaches to serialization

The above question arises as a result of the serialization process performed on messages to allow them to be sent through communication media. If the unit of communication is a message, this question has no meaning. If the units are computer words, one must determine their size and the order in which they are sent.

Since they are sent virtually at once, there is no need to determine the order of the elements of these words.

If the unit of transmission is an eight-bit byte, questions about bytes are meaningful but questions about the order of the elementary particles that constitute these bytes are not.

If the units of communication are bits, the atoms (quarks?) of computation, the only meaningful question concerns the order in which the bits are sent. Most modern communication is based on a single stream of information, the bit-stream. Hence, bits, rather than bytes or words, are the units of information that are actually transmitted over channels such as wires and satellites.

Notes on Swift's *Gulliver's Travels*

Swift's hero, Gulliver, is shipwrecked and washed ashore on Lilliput, whose six-inch inhabitants are required by law to break their eggs only at the little ends. Of course, all those citizens who habitually break their eggs at the big ends are angered by the proclamation. Civil war breaks out between the Little Endians and the Big Endians, resulting in the Big Endians taking refuge on a nearby island, the kingdom of Blefuscu. The controversy is ethically and politically important for the Lilliputians. In fact, Swift has 11,000 Lilliputian rebels die over the egg question. The issue might seem silly, but Swift is satirizing the actual causes of religious or holy wars.

Swift's point is that the difference between breaking an egg at the little end and breaking it at the big end is trivial. He suggests that everyone do it in his preferred way.

Of course, we are making the opposite point. We agree that the difference between sending information with the little or the big end first is trivial, but insist that everyone must do it in the same way to avoid anarchy.

Reproduced from *The Annotated Gulliver's Travels* by Isaac Asimov. Published by Crown Publishers, Inc.

Endianness

No consensus

Representation of multi-byte scalar values in memory

- Little-endian systems
  - least-significant byte at lowest address
  - Intel processors
- Big-endian systems
  - least-significant byte at highest address
  - internet protocols, legacy or embedded processors (e.g. SPARC, PowerPC)

Endianness versus portability

Low-level C programs

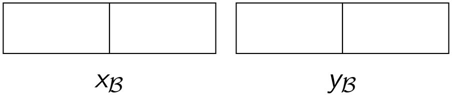
- typically rely on **assumptions** on endianness.

⇒ **Porting** to platform with opposite endianness is **challenging**.

Reading multi-byte input in network byte-order

Big-endian version

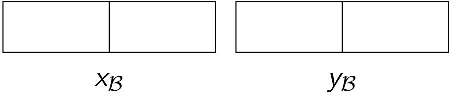
```
 u16 x, y;  // or u32, or u64
 read_from_network((u8 *)&x, sizeof(x));




 y = x;

// read y
```



Reading multi-byte input in network byte-order

Big-endian version

```
  u16 x, y;  // or u32, or u64
● read_from_network((u8 *)&x, sizeof(x));



  y = x;

 // read y
```

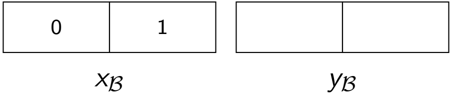


Reading multi-byte input in network byte-order

Big-endian version

```
u16 x, y;  // or u32, or u64
read_from_network((u8 *)&x, sizeof(x));
```

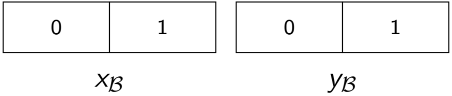
```
● y = x;

// read y
```



Reading multi-byte input in network byte-order

Big-endian version

```
u16 x, y;  // or u32, or u64
read_from_network((u8 *)&x, sizeof(x));




y = x;

● // read y
```



Reading multi-byte input in network byte-order

Big-endian version

```
u16 x, y;  // or u32, or u64
read_from_network((u8 *)&x, sizeof(x));




y = x;

// read y
```

0	1

$x_{\mathcal{B}}$

0	1

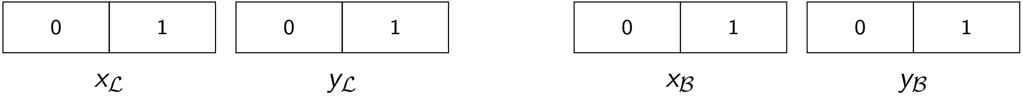
$y_{\mathcal{B}}$

$$1 = 0 \times 2^8 + 1 = y_{\mathcal{B}}$$

Reading multi-byte input in network byte-order

Big-endian version on little-endian machine

```
u16 x, y;  // or u32, or u64
read_from_network((u8 *)&x, sizeof(x));




y = x;

// read y
```

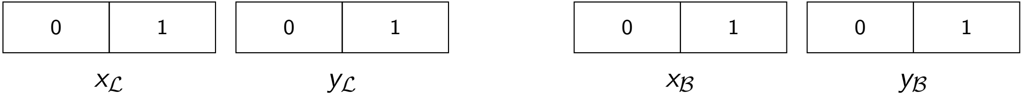


$$y_{\mathcal{L}} = 0 + 1 \times 2^8 = 256 \qquad 1 = 0 \times 2^8 + 1 = y_{\mathcal{B}}$$

Reading multi-byte input in network byte-order

Big-endian version on little-endian machine

```
u16 x, y;  // or u32, or u64
read_from_network((u8 *)&x, sizeof(x));




y = x;

// read y
```



0 1 0 1 0 1 0 1

$x_{\mathcal{L}}$ $y_{\mathcal{L}}$ $x_{\mathcal{B}}$ $y_{\mathcal{B}}$

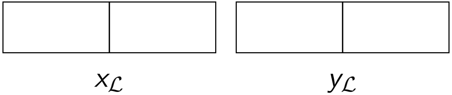
$$y_{\mathcal{L}} = 0 + 1 \times 2^8 = 256 \quad \neq \quad 1 = 0 \times 2^8 + 1 = y_{\mathcal{B}}$$

Reading multi-byte input in network byte-order

Porting to little-endian

```
u16 x, y;  // or u32, or u64
read_from_network((u8 *)&x, sizeof(x));

u8 *px = (u8 *)&x, *py = (u8 *)&y;
for (int i=0; i<sizeof(x); i++) py[i] = px[sizeof(x)-i-1];
```

```
// read y
```

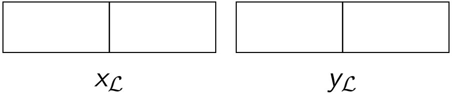


Reading multi-byte input in network byte-order

Porting to little-endian

```
u16 x, y;  // or u32, or u64
read_from_network((u8 *)&x, sizeof(x));

u8 *px = (u8 *)&x, *py = (u8 *)&y;
for (int i=0; i<sizeof(x); i++) py[i] = px[sizeof(x)-i-1];
```

```
// read y
```

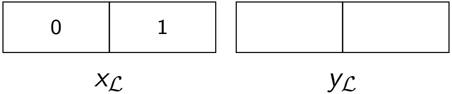


Reading multi-byte input in network byte-order

Porting to little-endian

```
  u16 x, y;  // or u32, or u64
  read_from_network((u8 *)&x, sizeof(x));

● u8 *px = (u8 *)&x, *py = (u8 *)&y;
  for (int i=0; i<sizeof(x); i++) py[i] = px[sizeof(x)-i-1];
```

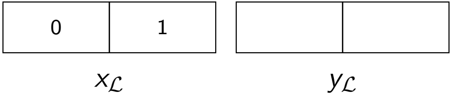
```
// read y
```



Reading multi-byte input in network byte-order

Porting to little-endian

```
  u16 x, y;  // or u32, or u64
  read_from_network((u8 *)&x, sizeof(x));

  u8 *px = (u8 *)&x, *py = (u8 *)&y;
● for (int i=0; i<sizeof(x); i++) py[i] = px[sizeof(x)-i-1];



 // read y
```

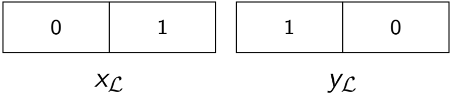


Reading multi-byte input in network byte-order

Porting to little-endian

```
u16 x, y;  // or u32, or u64
read_from_network((u8 *)&x, sizeof(x));

u8 *px = (u8 *)&x, *py = (u8 *)&y;
for (int i=0; i<sizeof(x); i++) py[i] = px[sizeof(x)-i-1];
```

● // read y



0	1	1	0

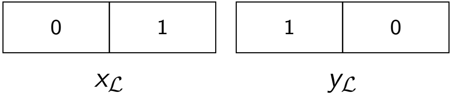
$x_{\mathcal{L}}$ $y_{\mathcal{L}}$

Reading multi-byte input in network byte-order

Porting to little-endian

```
u16 x, y;  // or u32, or u64
read_from_network((u8 *)&x, sizeof(x));

u8 *px = (u8 *)&x, *py = (u8 *)&y;
for (int i=0; i<sizeof(x); i++) py[i] = px[sizeof(x)-i-1];
```

```
// read y
```



0	1	1	0
$x_\mathcal{L}$		$y_\mathcal{L}$	

$$y_\mathcal{L} = 1 + 0 \times 2^8 = 1$$

Reading multi-byte input in network byte-order

Both versions, with conditional inclusion

```
  u16 x, y;  // or u32, or u64
  read_from_network((u8 *)&x, sizeof(x));
# if __BYTE_ORDER == __LITTLE_ENDIAN
  u8 *px = (u8 *)&x, *py = (u8 *)&y;
  for (int i=0; i<sizeof(x); i++) py[i] = px[sizeof(x)-i-1];
# else
  y = x;
# endif
```

// read y: $y_\mathcal{L} \overset{?}{=} y_\mathcal{B}$

0	1	1	0	0	1	0	1
$x_\mathcal{L}$		$y_\mathcal{L}$		$x_\mathcal{B}$		$y_\mathcal{B}$	

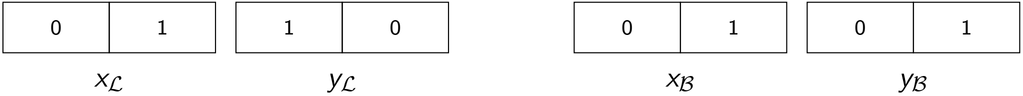
$$y_\mathcal{L} = 1 + 0 \times 2^8 = 1 \qquad 1 = 0 \times 2^8 + 1 = y_\mathcal{B}$$

Reading multi-byte input in network byte-order

Both versions, with conditional inclusion

```
  u16 x, y;  // or u32, or u64
  read_from_network((u8 *)&x, sizeof(x));
# if __BYTE_ORDER == __LITTLE_ENDIAN
  u8 *px = (u8 *)&x, *py = (u8 *)&y;
  for (int i=0; i<sizeof(x); i++) py[i] = px[sizeof(x)-i-1];
# else
  y = x;
# endif
```

// read y: $y_{\mathcal{L}} \overset{?}{=} y_{\mathcal{B}}$



$x_{\mathcal{L}}$		$y_{\mathcal{L}}$		$x_{\mathcal{B}}$		$y_{\mathcal{B}}$	
0	1	1	0	0	1	0	1

$$y_{\mathcal{L}} = 1 + 0 \times 2^8 = 1 \qquad = \qquad 1 = 0 \times 2^8 + 1 = y_{\mathcal{B}}$$

Reading multi-byte input in network byte-order

Both versions, with bitwise arithmetics

```
  u16 x, y;  // or u32, or u64
  read_from_network((u8 *)&x, sizeof(x));
# if __BYTE_ORDER == __LITTLE_ENDIAN
  y = (((x >> 8) & 0xff) | ((x & 0xff) << 8));  // bitwise arithmetic

# else
  y = x;
# endif
```

// read y: $y_{\mathcal{L}} \overset{?}{=} y_{\mathcal{B}}$

0	1	1	0	0	1	0	1
$x_{\mathcal{L}}$		$y_{\mathcal{L}}$		$x_{\mathcal{B}}$		$y_{\mathcal{B}}$	

$$y_{\mathcal{L}} = 1 + 0 \times 2^8 = 1 \qquad = \qquad 1 = 0 \times 2^8 + 1 = y_{\mathcal{B}}$$

Endian portability analysis

Endian portability

A program is called **endian portable**
if two endian-specific versions thereof

- compute equal outputs
- when run on equal inputs
- on their respective platforms.

Our approach

We present

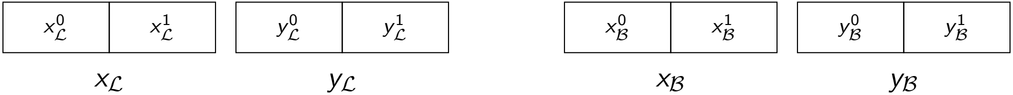
a **static analysis** by abstract interpretation
to infer the **endian portability**
of **large** real-world **low-level** C programs.

Semantics of simple endian-aware low-level C programs

Parameterizing the semantics with endianness

Memory model

The semantics of memory reads and writes depends on the endianness of the platform.



$x^0_{\mathcal{L}}$ $x^1_{\mathcal{L}}$ $y^0_{\mathcal{L}}$ $y^1_{\mathcal{L}}$ $x^0_{\mathcal{B}}$ $x^1_{\mathcal{B}}$ $y^0_{\mathcal{B}}$ $y^1_{\mathcal{B}}$

$x_{\mathcal{L}}$ $y_{\mathcal{L}}$ $x_{\mathcal{B}}$ $y_{\mathcal{B}}$

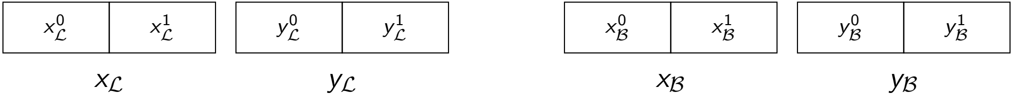
$$y_{\mathcal{L}} = y^0_{\mathcal{L}} + y^1_{\mathcal{L}} \times 2^8 \qquad y_{\mathcal{B}} = y^0_{\mathcal{B}} \times 2^8 + y^1_{\mathcal{B}}$$

Semantics of simple endian-aware low-level C programs

Parameterizing the semantics with endianness

Memory model

The semantics of memory reads and writes depends on the endianness of the platform.



$x_{\mathcal{L}}^0$ $x_{\mathcal{L}}^1$ $y_{\mathcal{L}}^0$ $y_{\mathcal{L}}^1$ $x_{\mathcal{B}}^0$ $x_{\mathcal{B}}^1$ $y_{\mathcal{B}}^0$ $y_{\mathcal{B}}^1$

$x_{\mathcal{L}}$ $y_{\mathcal{L}}$ $x_{\mathcal{B}}$ $y_{\mathcal{B}}$

$$y_{\mathcal{L}} = y_{\mathcal{L}}^0 + y_{\mathcal{L}}^1 \times 2^8 \qquad y_{\mathcal{B}} = y_{\mathcal{B}}^0 \times 2^8 + y_{\mathcal{B}}^1$$

Endian-aware cell-based memory model

Cells with endianness encoding ε

$$\langle V, o, \tau, \varepsilon \rangle \in \mathcal{C}ell \subseteq \mathcal{V} \times \mathbb{N} \times \textit{scalar-type} \times \{\, \mathcal{L}, \mathcal{B} \,\}$$

Semantics

Lifting (endian-aware) simple program semantics to (endian-diverse) double programs

Simple programs P_α $\alpha \in \{\, \mathcal{L}, \mathcal{B} \,\}$

Simple states in $\mathcal{E}_\alpha$ *(environments over cells)*

Statements $\mathbb{S}_\alpha[\![\, s \,]\!] \in \mathcal{P}(\mathcal{E}_\alpha) \to \mathcal{P}(\mathcal{E}_\alpha)$

Expressions $\mathbb{E}_\alpha[\![\, e \,]\!] \in \mathcal{E}_\alpha \to \mathcal{P}(\mathbb{V})$

Semantics

Lifting (endian-aware) simple program semantics to (endian-diverse) double programs

Simple programs P_α $\alpha \in \{ \mathcal{L}, \mathcal{B} \}$

Simple states in $\mathcal{E}_\alpha$ *(environments over cells)*

Statements $\mathbb{S}_\alpha[\![s]\!] \in \mathcal{P}(\mathcal{E}_\alpha) \to \mathcal{P}(\mathcal{E}_\alpha)$

Expressions $\mathbb{E}_\alpha[\![e]\!] \in \mathcal{E}_\alpha \to \mathcal{P}(\mathbb{V})$

Double program P

Double states in $\mathcal{D} \triangleq \mathcal{E}_\mathcal{L} \times \mathcal{E}_\mathcal{B}$ *(w.l.o.g.)*

Statements $\mathbb{D}[\![s]\!] \in \mathcal{P}(\mathcal{D}) \to \mathcal{P}(\mathcal{D})$

Conditions $\mathbb{F}[\![c_\mathcal{L} \parallel c_\mathcal{B}]\!] \in \mathcal{P}(\mathcal{D}) \to \mathcal{P}(\mathcal{D})$

Semantics

Lifting (endian-aware) simple program semantics to (endian-diverse) double programs

Simple programs P_α $\quad \alpha \in \{\, \mathcal{L}, \mathcal{B} \,\}$

Simple states in $\mathcal{E}_\alpha$ *(environments over cells)*

- Statements $\mathbb{S}_\alpha[\![\, s \,]\!] \in \mathcal{P}(\mathcal{E}_\alpha) \to \mathcal{P}(\mathcal{E}_\alpha)$
- Expressions $\mathbb{E}_\alpha[\![\, e \,]\!] \in \mathcal{E}_\alpha \to \mathcal{P}(\mathbb{V})$

Double program P

Double states in $\mathcal{D} \triangleq \mathcal{E}_\mathcal{L} \times \mathcal{E}_\mathcal{B}$ *(w.l.o.g.)*

- Statements $\mathbb{D}[\![\, s \,]\!] \in \mathcal{P}(\mathcal{D}) \to \mathcal{P}(\mathcal{D})$
- Conditions $\mathbb{F}[\![\, c_\mathcal{L} \parallel c_\mathcal{B} \,]\!] \in \mathcal{P}(\mathcal{D}) \to \mathcal{P}(\mathcal{D})$

Transfer functions

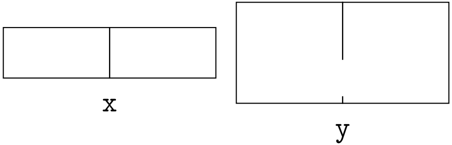
$$\mathbb{D}[\![\, s_\mathcal{L} \parallel s_\mathcal{B} \,]\!] X \triangleq \bigcup_{(\rho_\mathcal{L}, \rho_\mathcal{B}) \in X} (\mathbb{S}_\mathcal{L}[\![\, s_\mathcal{L} \,]\!]\{\, \rho_\mathcal{L} \,\} \times \mathbb{S}_\mathcal{B}[\![\, s_\mathcal{B} \,]\!]\{\, \rho_\mathcal{B} \,\})$$

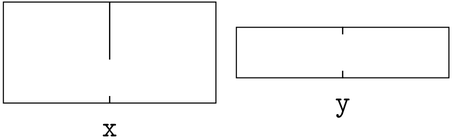
$$\begin{aligned}
\mathbb{D}[\![\, \textbf{if } e_\mathcal{L} \bowtie 0 \parallel e_\mathcal{B} \bowtie 0 \textbf{ then } s \textbf{ else } t \,]\!] \triangleq\ & \mathbb{D}[\![\, s \,]\!] \circ \mathbb{F}[\![\, e_\mathcal{L} \bowtie 0 \parallel e_\mathcal{B} \bowtie 0 \,]\!] \\
\dot\cup\ & \mathbb{D}[\![\, t \,]\!] \circ \mathbb{F}[\![\, e_\mathcal{L} \not\bowtie 0 \parallel e_\mathcal{B} \not\bowtie 0 \,]\!] \\
\dot\cup\ & \mathbb{D}[\![\, \pi_\mathcal{L}(s) \parallel \pi_\mathcal{B}(t) \,]\!] \circ \mathbb{F}[\![\, e_\mathcal{L} \bowtie 0 \parallel e_\mathcal{B} \not\bowtie 0 \,]\!] \\
\dot\cup\ & \mathbb{D}[\![\, \pi_\mathcal{L}(t) \parallel \pi_\mathcal{B}(s) \,]\!] \circ \mathbb{F}[\![\, e_\mathcal{L} \not\bowtie 0 \parallel e_\mathcal{B} \bowtie 0 \,]\!]
\end{aligned}$$

Analyzing the motivating example with cells

```
  u16 x, y;
  read_from_network((u8 *)&x, sizeof(x));
# if __BYTE_ORDER == __LITTLE_ENDIAN
  ((u8 *)&y)[0] = ((u8 *)&x)[1];
  ((u8 *)&y)[1] = ((u8 *)&x)[0];
# else
  y = x;
# endif
  output(y);   // $y_{\mathcal{L}} \overset{?}{=} y_{\mathcal{B}}$
```

Invariants and cell constraints



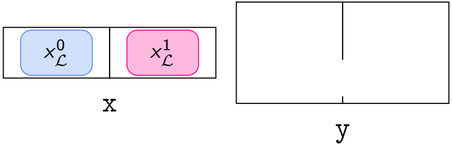


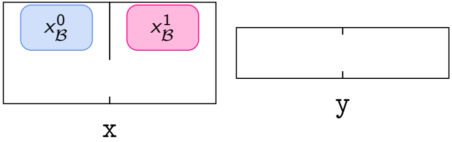
Analyzing the motivating example with cells

```
  u16 x, y;
  read_from_network((u8 *)&x, sizeof(x)); ●
# if __BYTE_ORDER == __LITTLE_ENDIAN
  ((u8 *)&y)[0] = ((u8 *)&x)[1];
  ((u8 *)&y)[1] = ((u8 *)&x)[0];
# else
  y = x;
# endif
  output(y);   // $y_{\mathcal{L}} \overset{?}{=} y_{\mathcal{B}}$
```

Invariants and cell constraints

$$x^0_{\mathcal{L}} = x^0_{\mathcal{B}} \land x^1_{\mathcal{L}} = x^1_{\mathcal{B}}$$





$$x^n_{\mathcal{B}} \triangleq \langle x, n, \mathbf{u8}, \mathcal{B} \rangle$$

$$x^n_{\mathcal{L}} \triangleq \langle x, n, \mathbf{u8}, \mathcal{L} \rangle$$

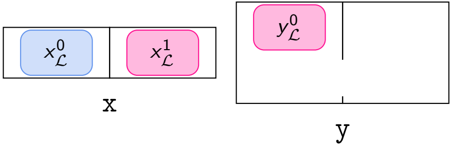
Analyzing the motivating example with cells

```
  u16 x, y;
  read_from_network((u8 *)&x, sizeof(x));
# if __BYTE_ORDER == __LITTLE_ENDIAN
  ((u8 *)&y)[0] = ((u8 *)&x)[1]; ●
  ((u8 *)&y)[1] = ((u8 *)&x)[0];
# else
  y = x;
# endif
  output(y);   // $y_\mathcal{L} \overset{?}{=} y_\mathcal{B}$
```

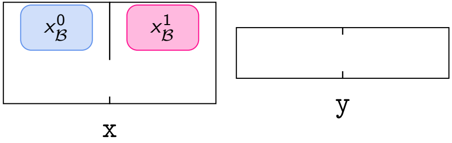
Invariants and cell constraints

$$x^0_\mathcal{L} = x^0_\mathcal{B} \ \wedge\ x^1_\mathcal{L} = x^1_\mathcal{B}$$

$$y^0_\mathcal{L} = x^1_\mathcal{L}$$



$x^n_\mathcal{L} \triangleq \langle x, n, \mathbf{u8}, \mathcal{L}\rangle$



$x^n_\mathcal{B} \triangleq \langle x, n, \mathbf{u8}, \mathcal{B}\rangle$

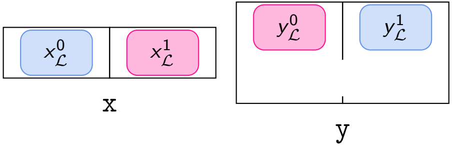
Analyzing the motivating example with cells

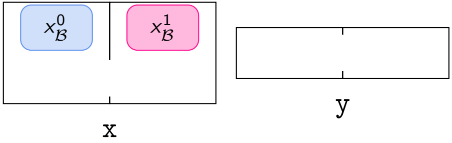
```
  u16 x, y;
  read_from_network((u8 *)&x, sizeof(x));
# if __BYTE_ORDER == __LITTLE_ENDIAN
  ((u8 *)&y)[0] = ((u8 *)&x)[1];
  ((u8 *)&y)[1] = ((u8 *)&x)[0]; ●
# else
  y = x;
# endif
  output(y);   // $y_{\mathcal{L}} \overset{?}{=} y_{\mathcal{B}}$
```

Invariants and cell constraints

$$x^0_{\mathcal{L}} = x^0_{\mathcal{B}} \;\wedge\; x^1_{\mathcal{L}} = x^1_{\mathcal{B}}$$

$$y^0_{\mathcal{L}} = x^1_{\mathcal{L}}$$
$$y^1_{\mathcal{L}} = x^0_{\mathcal{L}}$$





$x^n_{\mathcal{B}} \triangleq \langle x, n, \mathbf{u8}, \mathcal{B} \rangle$

$x^n_{\mathcal{L}} \triangleq \langle x, n, \mathbf{u8}, \mathcal{L} \rangle$

Analyzing the motivating example with cells

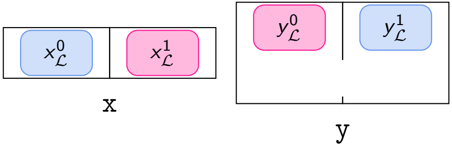
```
  u16 x, y;
  read_from_network((u8 *)&x, sizeof(x));
# if __BYTE_ORDER == __LITTLE_ENDIAN
  ((u8 *)&y)[0] = ((u8 *)&x)[1];
  ((u8 *)&y)[1] = ((u8 *)&x)[0];
# else
  y = x; ●
# endif
  output(y);   // $y_{\mathcal{L}} \overset{?}{=} y_{\mathcal{B}}$
```

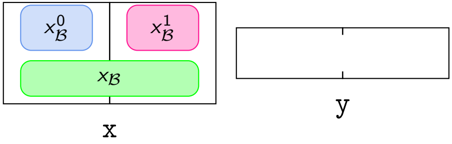
Invariants and cell constraints

$$x_{\mathcal{L}}^0 = x_{\mathcal{B}}^0 \ \wedge \ x_{\mathcal{L}}^1 = x_{\mathcal{B}}^1$$

$$y_{\mathcal{L}}^0 = x_{\mathcal{L}}^1$$
$$y_{\mathcal{L}}^1 = x_{\mathcal{L}}^0$$

$$x_{\mathcal{B}} = 2^8 \times x_{\mathcal{B}}^0 + x_{\mathcal{B}}^1$$





$$x_{\mathcal{L}}^n \triangleq \langle x, n, \mathbf{u8}, \mathcal{L} \rangle$$

$$x_{\mathcal{B}}^n \triangleq \langle x, n, \mathbf{u8}, \mathcal{B} \rangle$$
$$x_{\mathcal{B}} \triangleq \langle x, 0, \mathbf{u16}, \mathcal{B} \rangle$$

Analyzing the motivating example with cells

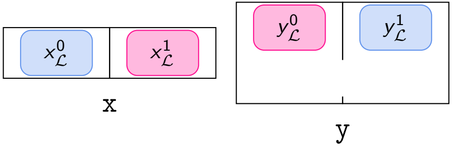
```
  u16 x, y;
  read_from_network((u8 *)&x, sizeof(x));
# if __BYTE_ORDER == __LITTLE_ENDIAN
  ((u8 *)&y)[0] = ((u8 *)&x)[1];
  ((u8 *)&y)[1] = ((u8 *)&x)[0];
# else
  y = x; ●
# endif
  output(y);   // $y_{\mathcal{L}} \overset{?}{=} y_{\mathcal{B}}$
```

Invariants and cell constraints

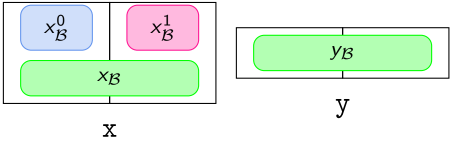
$$x^0_{\mathcal{L}} = x^0_{\mathcal{B}} \;\wedge\; x^1_{\mathcal{L}} = x^1_{\mathcal{B}}$$

$$y^0_{\mathcal{L}} = x^1_{\mathcal{L}}$$
$$y^1_{\mathcal{L}} = x^0_{\mathcal{L}}$$

$$x_{\mathcal{B}} = 2^8 \times x^0_{\mathcal{B}} + x^1_{\mathcal{B}} \;\wedge\; y_{\mathcal{B}} = x_{\mathcal{B}}$$



$x^n_{\mathcal{L}} \triangleq \langle x, n, \mathbf{u8}, \mathcal{L} \rangle$



$x^n_{\mathcal{B}} \triangleq \langle x, n, \mathbf{u8}, \mathcal{B} \rangle$

$x_{\mathcal{B}} \triangleq \langle x, 0, \mathbf{u16}, \mathcal{B} \rangle$

Analyzing the motivating example with cells

```
  u16 x, y;
  read_from_network((u8 *)&x, sizeof(x));
# if __BYTE_ORDER == __LITTLE_ENDIAN
  ((u8 *)&y)[0] = ((u8 *)&x)[1];
  ((u8 *)&y)[1] = ((u8 *)&x)[0];
# else
  y = x;
# endif
```

`output(y);` ● // $y_{\mathcal{L}} \overset{?}{=} y_{\mathcal{B}}$

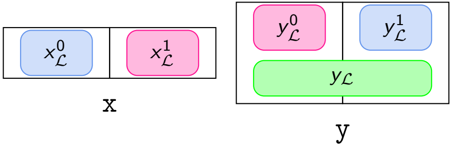
Invariants and cell constraints

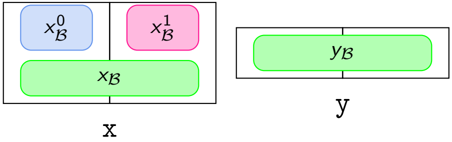
$$x_{\mathcal{L}}^0 = x_{\mathcal{B}}^0 \ \wedge \ x_{\mathcal{L}}^1 = x_{\mathcal{B}}^1$$

$$y_{\mathcal{L}}^0 = x_{\mathcal{L}}^1$$
$$y_{\mathcal{L}}^1 = x_{\mathcal{L}}^0$$

$$x_{\mathcal{B}} = 2^8 \times x_{\mathcal{B}}^0 + x_{\mathcal{B}}^1 \ \wedge \ y_{\mathcal{B}} = x_{\mathcal{B}}$$

$$y_{\mathcal{L}} = y_{\mathcal{L}}^0 + 2^8 \times y_{\mathcal{L}}^1$$





$x_{\mathcal{L}}^n \triangleq \langle x, n, \mathbf{u8}, \mathcal{L} \rangle$ $\quad$ $y_{\mathcal{L}} \triangleq \langle y, 0, \mathbf{u16}, \mathcal{L} \rangle$

$x_{\mathcal{B}}^n \triangleq \langle x, n, \mathbf{u8}, \mathcal{B} \rangle$

$x_{\mathcal{B}} \triangleq \langle x, 0, \mathbf{u16}, \mathcal{B} \rangle$

Optimizing the memory model for the common case

Complex invariants $\Longrightarrow$ expressive numerical domain?

- Program invariants and cell constraints

$$\begin{array}{llll} \boldsymbol{x^0_{\mathcal{L}} = x^0_{\mathcal{B}}} = y^1_{\mathcal{L}} & \boldsymbol{x^1_{\mathcal{L}} = x^1_{\mathcal{B}}} = y^0_{\mathcal{L}} & y_{\mathcal{B}} = x_{\mathcal{B}} & \boldsymbol{y_{\mathcal{L}} \overset{?}{=} y_{\mathcal{B}}} \\ x_{\mathcal{L}} = x^0_{\mathcal{L}} + 2^8 x^1_{\mathcal{L}} & y_{\mathcal{L}} = y^0_{\mathcal{L}} + 2^8 y^1_{\mathcal{L}} & x_{\mathcal{B}} = 2^8 x^0_{\mathcal{B}} + x^1_{\mathcal{B}} & y_{\mathcal{B}} = 2^8 y^0_{\mathcal{B}} + y^1_{\mathcal{B}} \end{array}$$

- Common case: most multi-byte cells hold **equal values**
  in the little- and big-endian memories

Optimizing the memory model for the common case

Complex invariants ⟹ expressive numerical domain?

- Program invariants and cell constraints

$$\begin{array}{llll}
\boldsymbol{x^0_\mathcal{L} = x^0_\mathcal{B}} = y^1_\mathcal{L} & \boldsymbol{x^1_\mathcal{L} = x^1_\mathcal{B}} = y^0_\mathcal{L} & y_\mathcal{B} = x_\mathcal{B} & \boldsymbol{y_\mathcal{L} \overset{?}{=} y_\mathcal{B}} \\
x_\mathcal{L} = x^0_\mathcal{L} + 2^8 x^1_\mathcal{L} & y_\mathcal{L} = y^0_\mathcal{L} + 2^8 y^1_\mathcal{L} & x_\mathcal{B} = 2^8 x^0_\mathcal{B} + x^1_\mathcal{B} & y_\mathcal{B} = 2^8 y^0_\mathcal{B} + y^1_\mathcal{B}
\end{array}$$

- Common case: most multi-byte cells hold **equal values** in the little- and big-endian memories

Extension of the bi-cell based memory model

- **Single** representation for **two** cells
  - from **different** program **versions**
  - holding **equal values**
  - representing **equalities**, or equalities **modulo byte-swapping**
- A bi-cell is $\mathcal{B}icell \triangleq \widetilde{Cell} \cup (\widetilde{Cell} \times \widetilde{Cell})$

either a single cell $\widetilde{Cell} \triangleq Cell_\mathcal{L} \uplus Cell_\mathcal{B}$

or a pair of cells holding equal values (shared bi-cell)

Analyzing the motivating example: **from cells** to bi-cells

```
  u16 x, y;
  read_from_network((u8 *)&x, sizeof(x));
# if __BYTE_ORDER == __LITTLE_ENDIAN
  ((u8 *)&y)[0] = ((u8 *)&x)[1];
  ((u8 *)&y)[1] = ((u8 *)&x)[0];
# else
  y = x;
# endif
  output(y); ● // $y_{\mathcal{L}} \overset{?}{=} y_{\mathcal{B}}$
```

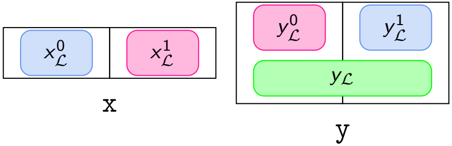
Invariants and cell constraints

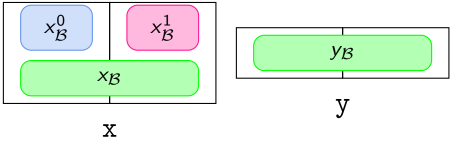
$$x^0_{\mathcal{L}} = x^0_{\mathcal{B}} \;\wedge\; x^1_{\mathcal{L}} = x^1_{\mathcal{B}}$$

$$y^0_{\mathcal{L}} = x^1_{\mathcal{L}}$$
$$y^1_{\mathcal{L}} = x^0_{\mathcal{L}}$$

$$x_{\mathcal{B}} = 2^8 \times x^0_{\mathcal{B}} + x^1_{\mathcal{B}} \;\wedge\; y_{\mathcal{B}} = x_{\mathcal{B}}$$

$$y_{\mathcal{L}} = y^0_{\mathcal{L}} + 2^8 \times y^1_{\mathcal{L}}$$





$x^n_{\mathcal{L}} \triangleq \langle x, n, \mathbf{u8}, \mathcal{L}\rangle$ $\quad y_{\mathcal{L}} \triangleq \langle y, 0, \mathbf{u16}, \mathcal{L}\rangle$

$x^n_{\mathcal{B}} \triangleq \langle x, n, \mathbf{u8}, \mathcal{B}\rangle$
$x_{\mathcal{B}} \triangleq \langle x, 0, \mathbf{u16}, \mathcal{B}\rangle$

Analyzing the motivating example: from cells **to bi-cells**

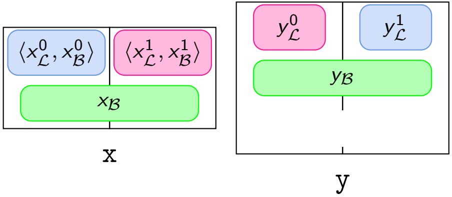
```
  u16 x, y;
  read_from_network((u8 *)&x, sizeof(x));
# if __BYTE_ORDER == __LITTLE_ENDIAN
  ((u8 *)&y)[0] = ((u8 *)&x)[1];
  ((u8 *)&y)[1] = ((u8 *)&x)[0];
# else
  y = x;
# endif
```

`output(y);` ● // $y_{\mathcal{L}} \overset{?}{=} y_{\mathcal{B}}$

Invariants and bi-cell constraints

$$y^0_{\mathcal{L}} = \langle x^1_{\mathcal{L}}, x^1_{\mathcal{B}} \rangle$$
$$y^1_{\mathcal{L}} = \langle x^0_{\mathcal{L}}, x^0_{\mathcal{B}} \rangle$$

$$y_{\mathcal{B}} = x_{\mathcal{B}} \ \wedge \ x_{\mathcal{B}} = \ldots$$



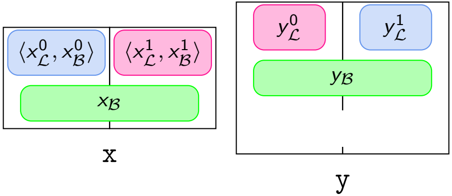
$\langle x^0_{\mathcal{L}}, x^0_{\mathcal{B}} \rangle$
$\langle x^1_{\mathcal{L}}, x^1_{\mathcal{B}} \rangle$
$x_{\mathcal{B}}$
x
$y^0_{\mathcal{L}}$
$y^1_{\mathcal{L}}$
$y_{\mathcal{B}}$
y

$$x^n_{\mathcal{L}} \triangleq \langle x, n, \mathbf{u8}, \mathcal{L}, \mathcal{L} \rangle$$
$$x^n_{\mathcal{B}} \triangleq \langle x, n, \mathbf{u8}, \mathcal{B}, \mathcal{B} \rangle$$
$$x_{\mathcal{B}} \triangleq \langle x, 0, \mathbf{u16}, \mathcal{B}, \mathcal{B} \rangle$$
$$y_{\mathcal{L}} \triangleq \langle y, 0, \mathbf{u16}, \mathcal{L}, \mathcal{L} \rangle$$
$$y_{\mathcal{B}} \triangleq \langle y, 0, \mathbf{u16}, \mathcal{B}, \mathcal{B} \rangle$$

Analyzing the motivating example: from cells **to bi-cells**

```
  u16 x, y;
  read_from_network((u8 *)&x, sizeof(x));
# if __BYTE_ORDER == __LITTLE_ENDIAN
  ((u8 *)&y)[0] = ((u8 *)&x)[1];
  ((u8 *)&y)[1] = ((u8 *)&x)[0];
# else
  y = x;
# endif
```

`output(y);` ● // $y_{\mathcal{L}} \overset{?}{=} y_{\mathcal{B}}$



$\langle x^0_{\mathcal{L}}, x^0_{\mathcal{B}} \rangle$
$\langle x^1_{\mathcal{L}}, x^1_{\mathcal{B}} \rangle$
$x_{\mathcal{B}}$
x
$y^0_{\mathcal{L}}$
$y^1_{\mathcal{L}}$
$y_{\mathcal{B}}$
y

Invariants and bi-cell constraints

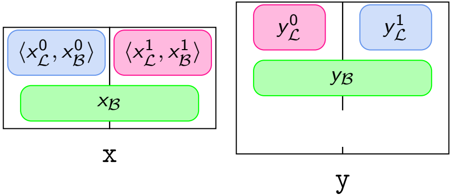
$$y^0_{\mathcal{L}} = \langle x^1_{\mathcal{L}}, x^1_{\mathcal{B}} \rangle$$
$$y^1_{\mathcal{L}} = \langle x^0_{\mathcal{L}}, x^0_{\mathcal{B}} \rangle$$

$$y_{\mathcal{B}} = x_{\mathcal{B}} \ \wedge \ x_{\mathcal{B}} = \dots$$

Shared bi-cell synthesis

Analyzing the motivating example: from cells **to bi-cells**

```
  u16 x, y;
  read_from_network((u8 *)&x, sizeof(x));
# if __BYTE_ORDER == __LITTLE_ENDIAN
  ((u8 *)&y)[0] = ((u8 *)&x)[1];
  ((u8 *)&y)[1] = ((u8 *)&x)[0];
# else
  y = x;
# endif
  output(y); ● // y_𝓛 ≟ y_𝓑
```



$\langle x^0_{\mathcal{L}}, x^0_{\mathcal{B}} \rangle$
$\langle x^1_{\mathcal{L}}, x^1_{\mathcal{B}} \rangle$
$x_{\mathcal{B}}$
x
$y^0_{\mathcal{L}}$
$y^1_{\mathcal{L}}$
$y_{\mathcal{B}}$
y

Invariants and bi-cell constraints

$$y^0_{\mathcal{L}} = \langle x^1_{\mathcal{L}}, x^1_{\mathcal{B}} \rangle$$
$$y^1_{\mathcal{L}} = \langle x^0_{\mathcal{L}}, x^0_{\mathcal{B}} \rangle$$

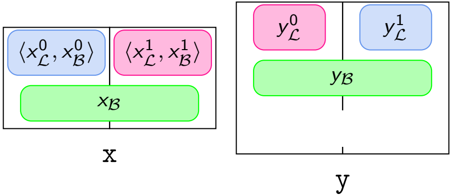
$$y_{\mathcal{B}} = x_{\mathcal{B}} \;\wedge\; x_{\mathcal{B}} = \ldots$$

Shared bi-cell synthesis

$\exists c : y_{\mathcal{L}} = c = y_{\mathcal{B}}$? $x_{\mathcal{B}}$ candidate

Analyzing the motivating example: from cells **to bi-cells**

```
  u16 x, y;
  read_from_network((u8 *)&x, sizeof(x));
# if __BYTE_ORDER == __LITTLE_ENDIAN
  ((u8 *)&y)[0] = ((u8 *)&x)[1];
  ((u8 *)&y)[1] = ((u8 *)&x)[0];
# else
  y = x;
# endif
  output(y); ● // $y_{\mathcal{L}} \overset{?}{=} y_{\mathcal{B}}$
```


$\langle x^0_{\mathcal{L}}, x^0_{\mathcal{B}} \rangle$
$\langle x^1_{\mathcal{L}}, x^1_{\mathcal{B}} \rangle$
$x_{\mathcal{B}}$
x
$y^0_{\mathcal{L}}$
$y^1_{\mathcal{L}}$
$y_{\mathcal{B}}$
y

Invariants and bi-cell constraints

$$y^0_{\mathcal{L}} = \langle x^1_{\mathcal{L}}, x^1_{\mathcal{B}} \rangle$$
$$y^1_{\mathcal{L}} = \langle x^0_{\mathcal{L}}, x^0_{\mathcal{B}} \rangle$$

$$y_{\mathcal{B}} = x_{\mathcal{B}} \ \wedge \ x_{\mathcal{B}} = \ldots$$

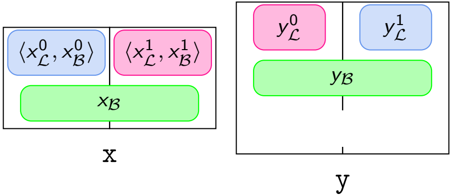
Shared bi-cell synthesis

$\exists c : \ y_{\mathcal{L}} = c = y_{\mathcal{B}}$? $\quad x_{\mathcal{B}}$ candidate

$y_{\mathcal{L}} = x_{\mathcal{B}}$?

Analyzing the motivating example: from cells **to bi-cells**

```
  u16 x, y;
  read_from_network((u8 *)&x, sizeof(x));
# if __BYTE_ORDER == __LITTLE_ENDIAN
  ((u8 *)&y)[0] = ((u8 *)&x)[1];
  ((u8 *)&y)[1] = ((u8 *)&x)[0];
# else
  y = x;
# endif
  output(y); ● // $y_{\mathcal{L}} \stackrel{?}{=} y_{\mathcal{B}}$
```



$\langle x^0_{\mathcal{L}}, x^0_{\mathcal{B}}\rangle$
$\langle x^1_{\mathcal{L}}, x^1_{\mathcal{B}}\rangle$
$x_{\mathcal{B}}$
x
$y^0_{\mathcal{L}}$
$y^1_{\mathcal{L}}$
$y_{\mathcal{B}}$
y

Invariants and bi-cell constraints

$$y^0_{\mathcal{L}} = \langle x^1_{\mathcal{L}}, x^1_{\mathcal{B}}\rangle$$
$$y^1_{\mathcal{L}} = \langle x^0_{\mathcal{L}}, x^0_{\mathcal{B}}\rangle$$

$$y_{\mathcal{B}} = x_{\mathcal{B}} \ \wedge \ x_{\mathcal{B}} = \ldots$$

Shared bi-cell synthesis

$\exists c : \ y_{\mathcal{L}} = c = y_{\mathcal{B}}$? $\quad x_{\mathcal{B}}$ candidate

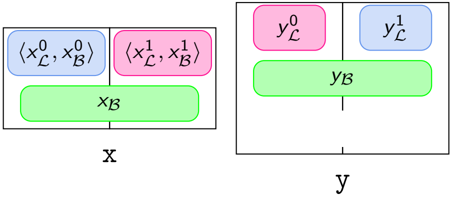
$y_{\mathcal{L}} = x_{\mathcal{B}}$?

$y^0_{\mathcal{L}} = x^1_{\mathcal{B}}$? ✓

$y^1_{\mathcal{L}} = x^0_{\mathcal{B}}$? ✓

Analyzing the motivating example: from cells **to bi-cells**

```
  u16 x, y;
  read_from_network((u8 *)&x, sizeof(x));
# if __BYTE_ORDER == __LITTLE_ENDIAN
  ((u8 *)&y)[0] = ((u8 *)&x)[1];
  ((u8 *)&y)[1] = ((u8 *)&x)[0];
# else
  y = x;
# endif
  output(y); ● // $y_{\mathcal{L}} \overset{?}{=} y_{\mathcal{B}}$
```



$\langle x^0_{\mathcal{L}}, x^0_{\mathcal{B}} \rangle$
$\langle x^1_{\mathcal{L}}, x^1_{\mathcal{B}} \rangle$
$x_{\mathcal{B}}$
x
$y^0_{\mathcal{L}}$
$y^1_{\mathcal{L}}$
$y_{\mathcal{B}}$
y

Invariants and bi-cell constraints

$$y^0_{\mathcal{L}} = \langle x^1_{\mathcal{L}}, x^1_{\mathcal{B}} \rangle$$
$$y^1_{\mathcal{L}} = \langle x^0_{\mathcal{L}}, x^0_{\mathcal{B}} \rangle$$

$$y_{\mathcal{B}} = x_{\mathcal{B}} \ \wedge \ x_{\mathcal{B}} = \ldots$$

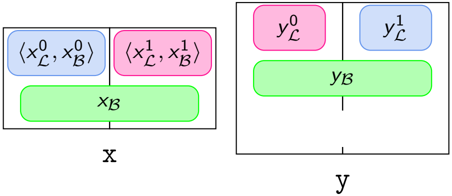
Shared bi-cell synthesis

$\exists c : \ y_{\mathcal{L}} = c = y_{\mathcal{B}}$? $x_{\mathcal{B}}$ candidate

$y_{\mathcal{L}} = x_{\mathcal{B}}$? ✓

Analyzing the motivating example: from cells **to bi-cells**

```
  u16 x, y;
  read_from_network((u8 *)&x, sizeof(x));
# if __BYTE_ORDER == __LITTLE_ENDIAN
  ((u8 *)&y)[0] = ((u8 *)&x)[1];
  ((u8 *)&y)[1] = ((u8 *)&x)[0];
# else
  y = x;
# endif
  output(y); ● // $y_\mathcal{L} \overset{?}{=} y_\mathcal{B}$
```



$\langle x^0_\mathcal{L}, x^0_\mathcal{B} \rangle$
$\langle x^1_\mathcal{L}, x^1_\mathcal{B} \rangle$
$x_\mathcal{B}$
x
$y^0_\mathcal{L}$
$y^1_\mathcal{L}$
$y_\mathcal{B}$
y

Invariants and bi-cell constraints

$$y^0_\mathcal{L} = \langle x^1_\mathcal{L}, x^1_\mathcal{B} \rangle$$
$$y^1_\mathcal{L} = \langle x^0_\mathcal{L}, x^0_\mathcal{B} \rangle$$

$$y_\mathcal{B} = x_\mathcal{B} \ \wedge \ x_\mathcal{B} = \ldots$$

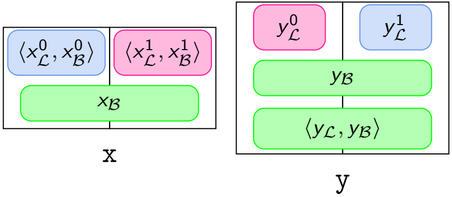
Shared bi-cell synthesis

$\exists c : y_\mathcal{L} = c = y_\mathcal{B}\,?$ ✓ $c = x_\mathcal{B}$

Analyzing the motivating example: from cells **to bi-cells**

```
  u16 x, y;
  read_from_network((u8 *)&x, sizeof(x));
# if __BYTE_ORDER == __LITTLE_ENDIAN
  ((u8 *)&y)[0] = ((u8 *)&x)[1];
  ((u8 *)&y)[1] = ((u8 *)&x)[0];
# else
  y = x;
# endif
```

`output(y);` ● // $y_{\mathcal{L}} \overset{?}{=} y_{\mathcal{B}}$



$\langle x^0_{\mathcal{L}}, x^0_{\mathcal{B}} \rangle$
$\langle x^1_{\mathcal{L}}, x^1_{\mathcal{B}} \rangle$
$x_{\mathcal{B}}$
x
$y^0_{\mathcal{L}}$
$y^1_{\mathcal{L}}$
$y_{\mathcal{B}}$
$\langle y_{\mathcal{L}}, y_{\mathcal{B}} \rangle$
y

Invariants and bi-cell constraints

$$y^0_{\mathcal{L}} = \langle x^1_{\mathcal{L}}, x^1_{\mathcal{B}} \rangle$$
$$y^1_{\mathcal{L}} = \langle x^0_{\mathcal{L}}, x^0_{\mathcal{B}} \rangle$$

$$y_{\mathcal{B}} = x_{\mathcal{B}} \ \wedge \ x_{\mathcal{B}} = \ldots$$

Shared bi-cell synthesis

$\langle y_{\mathcal{L}}, y_{\mathcal{B}} \rangle$ synthesized by

pattern-matching

$+$ simple equalities
(symbolic propagation)

The bit-slice numerical domain

Motivating example

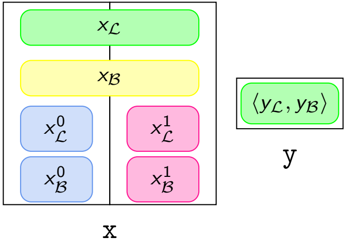
```
  u16 x; u8 *p = (u8 *)&x;
  u8 y = input(0,255);
# if __BYTE_ORDER == __LITTLE_ENDIAN
  x = y | 0xff00;
# else
  x = (y << 8) | 0xff;
# endif
  output(p[0]);
  output(p[1]);
```

Program invariants

$$x_{\mathcal{L}} = \langle y_{\mathcal{L}}, y_{\mathcal{B}} \rangle + 65280$$

$$x_{\mathcal{B}} = 256 \times \langle y_{\mathcal{L}}, y_{\mathcal{B}} \rangle + 255$$

$$\boldsymbol{x}^0_{\mathcal{L}} \overset{?}{=} \boldsymbol{x}^0_{\mathcal{B}}$$
$$\boldsymbol{x}^1_{\mathcal{L}} \overset{?}{=} \boldsymbol{x}^1_{\mathcal{B}}$$



Bi-cell constraints

$$\boldsymbol{x}^0_{\mathcal{L}} = byte(x_{\mathcal{L}}, 0) \qquad \boldsymbol{x}^0_{\mathcal{B}} = byte(x_{\mathcal{B}}, 1)$$
$$\boldsymbol{x}^1_{\mathcal{L}} = byte(x_{\mathcal{L}}, 1) \qquad \boldsymbol{x}^1_{\mathcal{B}} = byte(x_{\mathcal{L}}, 0)$$

The bit-slice numerical domain

Symbolic predicates (inspired by Miné [2006b], Miné [2012])

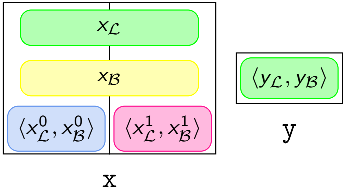
```
  u16 x; u8 *p = (u8 *)&x;
  u8 y = input(0,255);
# if __BYTE_ORDER == __LITTLE_ENDIAN
  x = y | 0xff00;
# else
  x = (y << 8) | 0xff;
# endif
  output(p[0]);
  output(p[1]);
```

Program invariants

$byte(x_{\mathcal{L}}, 0) = \langle y_{\mathcal{L}}, y_{\mathcal{B}} \rangle \quad byte(x_{\mathcal{L}}, 1) = 255$

$byte(x_{\mathcal{B}}, 0) = 255 \quad byte(x_{\mathcal{B}}, 1) = \langle y_{\mathcal{L}}, y_{\mathcal{B}} \rangle$

$$\boldsymbol{x^0_{\mathcal{L}} = x^0_{\mathcal{B}}}$$
$$\boldsymbol{x^1_{\mathcal{L}} = x^1_{\mathcal{B}}}$$



$x_{\mathcal{L}}$
$x_{\mathcal{B}}$
$\langle x^0_{\mathcal{L}}, x^0_{\mathcal{B}} \rangle$
$\langle x^1_{\mathcal{L}}, x^1_{\mathcal{B}} \rangle$
x
$\langle y_{\mathcal{L}}, y_{\mathcal{B}} \rangle$
y

Bi-cell constraints

$\boldsymbol{x^0_{\mathcal{L}}} = byte(x_{\mathcal{L}}, 0) \quad \boldsymbol{x^0_{\mathcal{B}}} = byte(x_{\mathcal{B}}, 1)$

$\boldsymbol{x^1_{\mathcal{L}}} = byte(x_{\mathcal{L}}, 1) \quad \boldsymbol{x^1_{\mathcal{B}}} = byte(x_{\mathcal{L}}, 0)$

Implementation

Extensions of prototype abstract interpreter

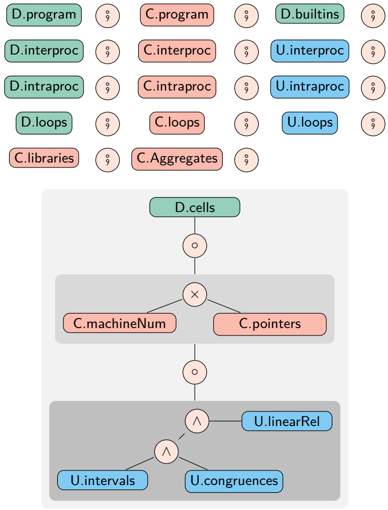
Compared to the previous version (6,700 lines of OCaml)

300 lines **updated** in the **bi-cell** memory domain *(8%)*

1,000 lines **added** for the **bit-slice** predicate domain

Implementation

Patch analysis (bi-cells)



Endian portability analysis (bi-cells and bit-slices)

⨟ Sequence

∧ Reduced product

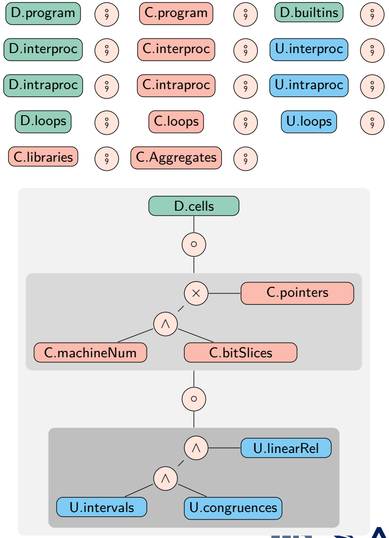
× Cartesian product

∘ Composition

🔵 Universal

🔴 C specific

🟢 Double C



Benchmarks

Origin	Name	LOC	Time	Revision	Result
Open Source	GENEVE	218	1 s	2014-1	🐞
				2014-2	✓
				2016	🐞
				2017	✓
	MLX5	125	155 ms	2017	🐞
				2020-1	🐞
				2020-2	✓
	Squashfs	110	150 ms	2020-1	🐞
				2020-2	✓
Industrial	Module S	300 K	9.7 h	2020	✓
	Module A	1 M	20.4 h	2020	🐞
				2021	✓

Disclaimer:

- Modules A and S are part of an early prototype, not in production yet.
- All findings have been incorporated into the development cycle.

Agenda

1. Introduction
2. Patch analysis for numerical programs
3. Patch analysis for C and structure layout portability
4. Endian portability analysis for C programs
5. Conclusion

Contributions

Double program semantics

- concrete semantics for two versions
- joint analysis by induction on syntax
- double program construction algorithm
- *support for unbounded input streams*

Bi-cell memory domain

- symbolic relations between memories
- scalable patch analyses
- scalable portability analyses

Numerical domains

- bit-slice domain
- *Delta domain*
- near-linear cost

Implementation and experimentation

- prototype analyzer on MOPSA
- small slices of open source software
- large real-world avionics software

Future work

Industrialization

- endian portability for simulation
- non regression for product-lines

Portability analysis

- 32-bit versus 64-bit
- different 64-bit data models
- porting from x86 or PowerPC to ARM
- changes in OS data types
- *Year 2038 problem*
- different ranges of inputs (Ariane 5.01)

Semantic differencing

- characterize semantic differences
- infer a semantic distance
- evaluate the cost of a patch
- infer an "improvement" property

Hyperproperties and information flow

- 2-safety properties
- prove secrecy and noninterference
- experiment on more complex programs

Summary

Topics

- patch analysis
- structure layout portability analysis
- endian portability analysis

Contributions

- Double program semantics
- Bi-cell memory domain
- Numerical domains
- Implementation and experimentation

Future work

- Industrialization
- Portability analysis
- Semantic differencing
- Hyperproperties and information flow

Thank you for your attention

Questions?

AIRBUS

Backup slides

References

J. Bertrane, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, and X. Rival. Static analysis and verification of aerospace software by abstract interpretation. In *AIAA Infotech@Aerospace*, number 2010-3385, pages 1–38. AIAA, Apr. 2010.

P. Cousot and R. Cousot. *A gentle introduction to formal verification of computer systems by abstract interpretation*, pages 1–29. NATO Science Series III: Computer and Systems Sciences. IOS Press, 2010.

B. Godlin and O. Strichman. Regression verification. In *Proceedings of DAC '09*, pages 466–471, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-497-3.

V. Klebanov, P. Rümmer, and M. Ulbrich. Automating regression verification of pointer programs by predicate abstraction. *Formal Methods in System Design*, 52(3):229–259, June 2018. ISSN 1572-8102. doi: 10.1007/s10703-017-0293-8.

S. K. Lahiri, C. Hawblitzel, M. Kawaguchi, and H. Rebêlo. Symdiff: A language-agnostic semantic diff tool for imperative programs. In *CAV*, pages 712–717, 2012. ISBN 978-3-642-31424-7.

A. Miné. Field-sensitive value analysis of embedded C programs with union types and pointer arithmetics. In *Proc. of the ACM SIGPLAN/SIGBED Conf. on Languages, Compilers, and Tools for Embedded Systems (LCTES'06)*, pages 54–63. ACM, June 2006a.

A. Miné. Symbolic methods to enhance the precision of numerical abstract domains. In *Proc. of the 7th Int. Conf. on Verification, Model Checking, and Abstract Interpretation (VMCAI'06)*, volume 3855 of *LNCS*, pages 348–363. Springer, Jan. 2006b.

A. Miné. Abstract domains for bit-level machine integer and floating-point operations. In *Proc. of the 4th Int. Workshop on Invariant Generation (WING'12)*, number HW-MACS-TR-0097, page 16. Computer Science, School of Mathematical and Computer Science, Heriot-Watt University, UK, Jun. 2012.

A. Miné. Static analysis by abstract interpretation of concurrent programs. Technical report, École normale supérieure, May 2013.

N. Partush and E. Yahav. Abstract semantic differencing for numerical programs. In *SAS*, pages 238–258, 2013. ISBN 978-3-642-38856-9.

N. Partush and E. Yahav. Abstract semantic differencing via speculative correlation. In *Proceedings of OOPSLA'14*, pages 811–828, 2014. ISBN 978-1-4503-2585-1.