Static Analysis of Endian Portability
by Abstract Interpretation
SAS 2021

David Delmas\textsuperscript{1,2}  Abdelraouf Ouadjaout\textsuperscript{2}  Antoine Miné\textsuperscript{2}

\textsuperscript{1}Airbus – Avionics Software
\textsuperscript{2}Sorbonne Université – LIP6

17 October 2021
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.

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.
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**.
Agenda

1 Motivating example
2 Syntax and concrete semantics
3 Memory model
4 Evaluation
Reading multi-byte input in network byte-order

Big-endian version

```c
1  u16 x, y;  // or u32, or u64
2  read_from_network((u8 *)&x, sizeof(x));
3
4
5
6
7  y = x;
8
9  // read y
```

![Diagram](image)
Reading multi-byte input in network byte-order

Big-endian version

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

y = x;

// read y
```

![Diagram](diagram.png)
Reading multi-byte input in network byte-order

Big-endian version

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

y = x;

// read y
```

![Diagram showing big-endian byte order]
Reading multi-byte input in network byte-order

Big-endian version

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

y = x;

// read y
```

![Byte order diagram]

```plaintext
xB: 0 1
yB: 0 1
```
Reading multi-byte input in network byte-order

Big-endian version

1. `u16 x, y;  // or u32, or u64`
2. `read_from_network((u8 *)&x, sizeof(x));`
3. `y = x;`
4. `// read y`

\[
\begin{array}{c|c|c|c|c}
\hline
0 & 1 & 0 & 1 \\
\hline
x_B & & & \\
\hline
y_B & & & \\
\hline
\end{array}
\]

\[
1 = 0 \times 2^8 + 1 = y_B
\]
Reading multi-byte input in network byte-order

Big-endian version on little-endian machine

```
1   u16 x, y;  // or u32, or u64
2   read_from_network((u8 *)&x, sizeof(x));
3
4
5
6
7   y = x;
8
9   // read y
```

\[
x_L = \begin{array}{cc}
0 & 1 \\
\end{array}
\quad y_L = \begin{array}{cc}
0 & 1 \\
\end{array}
\]

\[
y_L = 0 + 1 \times 2^8 = 256
\]

\[
x_B = \begin{array}{cc}
0 & 1 \\
\end{array}
\quad y_B = \begin{array}{cc}
0 & 1 \\
\end{array}
\]

\[
1 = 0 \times 2^8 + 1 = y_B
\]
Reading multi-byte input in network byte-order
Big-endian version on little-endian machine

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

y = x;

// read y
```

\[
x_L \quad y_L \\
0 \quad 1 \\
0 \quad 1
\]

\[
x_B \quad y_B \\
0 \quad 1 \\
0 \quad 1
\]

\[
y_L = 0 + 1 \times 2^8 = 256 \\
\neq \\
1 = 0 \times 2^8 + 1 = y_B
\]
Reading multi-byte input in network byte-order
Porting to little-endian

1. `u16 x, y; // or u32, or u64`
2. `read_from_network((u8 *)&x, sizeof(x));`
3. `u8 *px = (u8 *)&x, *py = (u8 *)&y;`
4. `for (int i=0; i<sizeof(x); i++) py[i] = px[offsetof(x)-i-1];`
5. `// read y`

\[ X_L \] \[ Y_L \]
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

\[ \begin{array}{cc}
  x_L & y_L \\
\end{array} \]
Reading multi-byte input in network byte-order

Porting to little-endian

```c
1  u16 x, y; // or u32, or u64
2  read_from_network((u8 *)&x, sizeof(x));
3
4  u8 *px = (u8 *)&x, *py = (u8 *)&y;
5  for (int i=0; i<sizeof(x); i++) py[i] = px[ sizeof(x)-i-1 ];
6
7
8
9  // read y
```

\[ x_\mathcal{L} \quad \begin{array}{|c|c|} \hline 0 & 1 \end{array} \quad y_\mathcal{L} \]
Reading multi-byte input in network byte-order

Porting to little-endian

```
1  u16 x, y;  // or u32, or u64
2  read_from_network((u8 *)&x, sizeof(x));
3
4  u8 *px = (u8 *)&x, *py = (u8 *)&y;
5  for (int i=0; i<sizeof(x); i++) py[i] = px[sizeof(x)-i-1];
6
7
8
9  // read y
```

```
|   |   |
|---|---|
| 0 | 1 |

x_L

|   |   |
|---|---|
|   |   |

y_L

Delmas, Ouadiaout, Miné  Endian Portability Analysis
Reading multi-byte input in network byte-order

Porting to little-endian

```c
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

| x_L | y_L |
|-----|-----|
| 0   | 1   |
| 1   | 0   |
```
Reading multi-byte input in network byte-order

Porting to little-endian

```c
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
```

\[
x_L = \begin{array}{cc}
0 & 1 \\
1 & 0 \\
\end{array}
\]

\[
y_L = 1 + 0 \times 2^8 = 1
\]
Reading multi-byte input in network byte-order
Both versions, with conditional inclusion

```c
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: \( y_L = y_B \)

\[
\begin{array}{c|c}
\hline
0 & 1 \\
\hline
x_L & y_L \\
\hline
\end{array}
\]

\[
\begin{array}{c|c|c|c}
\hline
0 & 1 & 0 & 1 \\
\hline
x_B & y_B \\
\hline
\end{array}
\]

\[
y_L = 1 + 0 \times 2^8 = 1
\]

\[
1 = 0 \times 2^8 + 1 = y_B
\]
```
Reading multi-byte input in network byte-order
Both versions, with conditional inclusion

```c
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_L ?
```

// read y: \( y_L = y_B \)

\[
\begin{array}{|c|c|}
\hline
0 & 1 \\
\hline
\end{array}
\quad \begin{array}{|c|c|}
\hline
1 & 0 \\
\hline
\end{array}
\quad \begin{array}{|c|c|}
\hline
0 & 1 \\
\hline
\end{array}
\quad \begin{array}{|c|c|}
\hline
0 & 1 \\
\hline
\end{array}
\]

\( y_L = 1 + 0 \times 2^8 = 1 \) \quad = \quad 1 = 0 \times 2^8 + 1 = y_B
Reading multi-byte input in network byte-order
Both versions, with bitwise arithmetics

```c
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)); // see paper
#else
    y = x;
#endif

// read y: \( y_{\text{L}} \stackrel{?}{=} y_{\text{B}} \)

\[
x_{\text{L}} \quad 0 \quad 1 \\
| \quad 1 \quad 0 \\
\]
\[
y_{\text{L}} = 1 + 0 \times 2^8 = 1
\]

\[
x_{\text{B}} \quad 0 \quad 1 \\
| \quad 0 \quad 1 \\
\]
\[
y_{\text{B}} = 1 = 0 \times 2^8 + 1 = y_{\text{B}}
\]
A program is called **endian portable** if two **endian-specific versions** thereof compute **equal** outputs when run on **equal** inputs on their respective **platforms**.

**This talk**

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

1. Motivating example
2. Syntax and concrete semantics
3. Memory model
4. Evaluation
Agenda

1. Motivating example
2. Syntax and concrete semantics
3. Memory model
4. Evaluation
Simple and double programs

C-like syntax

dstat ::= stat
     | stat || stat
     | if dcond then dstat else dstat
     | ...
     | assert_sync(expr)

stat ::= *scalar-type expr ← expr
     | if cond then stat else stat
     | ...

dcond ::= cond
     | cond || cond

dstat ::= stat

cond ::= expr ⊥ 0 ⊥ ∈ {≤, ≥, =, ≠, <, >}

eexpr ::= *scalar-type expr
    | & V
    | [c1, c2] c1, c2 ∈ ℤ
    | ◦ expr
    | expr ◦ expr

scalar-type ::= int-type | ptr

int-type ::= u8 | u16 | u32 | u64
       | s8 | s16 | s32 | s64

◆ ::= − | ~ | (scalar-type)
◆ ::= + − | * | / | % | & | | ^ | >> | <<
Lifting simple program semantics to double programs

Double program \( P \)

Simple states in \( \mathcal{E} \triangleq \mathcal{V} \rightarrow \mathcal{V} \)

Double states in \( \mathcal{D} \triangleq \mathcal{E} \times \mathcal{E} \)

Endianness w.l.o.g. \((\mathcal{L}, \mathcal{B})\)

Semantics \( \mathbb{D}[s] \in \mathcal{P}(\mathcal{D}) \rightarrow \mathcal{P}(\mathcal{D}) \)
Lifting simple program semantics to double programs

**Double program** $P$

- **Simple states** in $\mathcal{E} \triangleq \mathcal{V} \rightarrow \mathcal{V}$
- **Double states** in $\mathcal{D} \triangleq \mathcal{E} \times \mathcal{E}$
- **Endianness** w.l.o.g. $(\mathcal{L}, \mathcal{B})$
- **Semantics** $\mathcal{D}[s] \in \mathcal{P}(\mathcal{D}) \rightarrow \mathcal{P}(\mathcal{D})$

**Transfer functions**

Delmas and Miné [2019a,b]

\[
\mathcal{D}[s_1 \parallel s_2]X \triangleq \bigcup_{(\rho_L, \rho_B) \in X} (\mathcal{S}_L[s_1]\{\rho_L\} \times \mathcal{S}_B[s_2]\{\rho_B\})
\]
Lifting simple program semantics to double programs

**Double program** $P$

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

**Double states** in $D \triangleq E \times E$

**Endianness** w.l.o.g. $(\mathcal{L}, \mathcal{B})$

**Semantics** $D[s] \in \mathcal{P}(D) \rightarrow \mathcal{P}(D)$

**Transfer functions**

Delmas and Miné [2019a,b]

$D[s_1 || s_2]X \triangleq \bigcup_{(\rho_L, \rho_B)} (\mathcal{S}_L[s_1]\{\rho_L\} \times \mathcal{S}_B[s_2]\{\rho_B\})$

$D[\text{if } e_1 \triangleright 0 || e_2 \triangleright 0 \text{ then } s \text{ else } t] \triangleq D[\ ] \circ F[e_1 \triangleright 0 || e_2 \triangleright 0]$

$\bigcup D[\ ] \circ F[e_1 \triangleright 0 || e_2 \triangleright 0]$

$\bigcup D[\ ] \circ F[e_1 \triangleright 0 || e_2 \triangleright 0]$

$\bigcup D[\ ] \circ F[e_1 \triangleright 0 || e_2 \triangleright 0]$

where $F[e_1 \triangleright 0 || e_2 \triangleright 0]X \triangleq \{(\rho_L, \rho_B) \in X \mid \exists v_1 \in \mathcal{E}_L[e] \rho_L : \exists v_2 \in \mathcal{E}_B[e] \rho_B : v_1 \triangleright 0 \land v_2 \triangleright 0\}$
Lifting simple program semantics to double programs

**Double program** $P$

- **Simple states** in $E \triangleq \mathcal{V} \rightarrow \mathcal{V}$
- **Double states** in $D \triangleq E \times E$
- **Endianness** w.l.o.g. $(\mathcal{L}, \mathcal{B})$
- **Semantics** $D[s] \in \mathcal{P}(D) \rightarrow \mathcal{P}(D)$

**Transfer functions**

\[
D[s_1 \parallel s_2]X \triangleq \bigcup_{(\rho_L, \rho_B) \in X} (S_L[s_1] \{ \rho_L \} \times S_B[s_2] \{ \rho_B \})
\]

\[
D[\text{if } e_1 \triangleleft 0 \parallel e_2 \triangleleft 0 \text{ then } s \text{ else } t] \triangleq D[\ldots] \circ F[e_1 \triangleleft 0 \parallel e_2 \triangleleft 0]
\]

\[
\bar{D}[\ldots] \circ F[e_1 \triangleright 0 \parallel e_2 \triangleright 0]
\]

\[
\bar{D}[\ldots] \circ F[e_1 \nrightarrow 0 \parallel e_2 \nrightarrow 0]
\]

where $F[e_1 \triangleleft 0 \parallel e_2 \triangleleft 0]X \triangleq \{(\rho_L, \rho_B) \in X | \exists v_1 \in E_L[e] \rho_L : \exists v_2 \in E_B[e] \rho_B : v_1 \triangleleft 0 \land v_2 \triangleleft 0\}$
Lifting simple program semantics to double programs

Double program \( P \)

- **Simple states** in \( \mathcal{E} \triangleq \mathcal{V} \rightarrow \mathcal{V} \)
- **Double states** in \( \mathcal{D} \triangleq \mathcal{E} \times \mathcal{E} \)
- **Endianness** w.l.o.g. \( (\mathcal{L}, \mathcal{B}) \)
- **Semantics** \( \mathcal{D}[s] \in \mathcal{P}(\mathcal{D}) \rightarrow \mathcal{P}(\mathcal{D}) \)

**Transfer functions**

\[
\begin{align*}
\mathcal{D}[s_1 \parallel s_2]X & \triangleq \bigcup_{(\rho_L, \rho_B) \in X} \left( \mathcal{S}_L[s_1] \{ \rho_L \} \times \mathcal{S}_B[s_2] \{ \rho_B \} \right) \\
\mathcal{D}[\text{if } e_1 \triangleright 0 \parallel e_2 \triangleright 0 \text{ then } s \text{ else } t] & \triangleq \mathcal{D}[s] \circ \mathcal{F}[e_1 \triangleright 0 \parallel e_2 \triangleright 0] \\
\bigcup \mathcal{D}[\ldots] & \circ \mathcal{F}[e_1 \triangleright 0 \parallel e_2 \triangleright 0] \\
\bigcup \mathcal{D}[\ldots] & \circ \mathcal{F}[e_1 \ntriangleright 0 \parallel e_2 \ntriangleright 0]
\end{align*}
\]

where \( \mathcal{F}[e_1 \triangleright 0 \parallel e_2 \triangleright 0]X \triangleq \{(\rho_L, \rho_B) \in X \mid \exists v_1 \in \mathcal{E}_L[e] \rho_L : \exists v_2 \in \mathcal{E}_B[e] \rho_B : v_1 \triangleright 0 \land v_2 \triangleright 0\} \)

Delmas, Ouadjaout, Miné [2019a,b]
### Lifting simple program semantics to double programs

#### Double program $P$

| Simple states | in $\mathcal{E} \triangleq \mathcal{V} \rightarrow \mathcal{V}$ |
|---------------|-------------------------------------------------|
| Double states | in $\mathcal{D} \triangleq \mathcal{E} \times \mathcal{E}$ |
| Endianness   | w.l.o.g. $(\mathcal{L}, \mathcal{B})$ |
| Semantics    | $\mathcal{D}[s] \in \mathcal{P}(\mathcal{D}) \rightarrow \mathcal{P}(\mathcal{D})$ |

#### Transfer functions

| $\mathcal{D}[s_1 || s_2]X$ | $\triangleq \bigcup_{(\rho_L, \rho_B) \in X} (\mathcal{S}_L[s_1] \{ \rho_L \} \times \mathcal{S}_B[s_2] \{ \rho_B \})$ |
|-----------------------------|------------------------------------------------------------------|
| $\mathcal{D}[\text{if } e_1 \perp 0 || e_2 \perp 0 \text{ then } s \text{ else } t]$ | $\triangleq \mathcal{D}[s] \circ \mathcal{F}[e_1 \perp 0 || e_2 \perp 0]$ |
|                            | $\lor \mathcal{D}[t] \circ \mathcal{F}[e_1 \perp 0 || e_2 \perp 0]$ |
|                            | $\lor \mathcal{D}[s_L || s_B] \circ \mathcal{F}[e_1 \perp 0 || e_2 \perp 0]$ |
|                            | $\lor \mathcal{D}[t_L || t_B] \circ \mathcal{F}[e_1 \perp 0 || e_2 \perp 0]$ |

where

| $\mathcal{F}[e_1 \perp 0 || e_2 \perp 0]X$ | $\triangleq \{(\rho_L, \rho_B) \in X | \exists v_1 \in E_L[e] \rho_L : \exists v_2 \in E_B[e] \rho_B : v_1 \perp 0 \land v_2 \perp 0\}$ |
|------------------------------------------|------------------------------------------------------------------|
| $s_L, s_B$                               | $\triangleq$ little-endian and big-endian versions of $s$ |
Lifting simple program semantics to double programs

**Double program** \( P \)

- **Simple states** in \( E \) \( \triangleq \mathcal{V} \to \mathcal{V} \)
- **Double states** in \( D \) \( \triangleq E \times E \)
- **Endianness** w.l.o.g. \( (L, B) \)
- **Semantics** \( D[s] \in \mathcal{P}(D) \to \mathcal{P}(D) \)

**Transfer functions**

\[
\begin{align*}
\mathcal{D}[s_1 \parallel s_2]X & \triangleq \bigcup_{(\rho_L, \rho_B) \in X}(\mathcal{S}_L[s_1] \{ \rho_L \} \times \mathcal{S}_B[s_2] \{ \rho_B \}) \\
\mathcal{D}[\text{if } e_1 \triangleright 0 \parallel e_2 \triangleright 0 \text{ then } s \text{ else } t] & \triangleq \bigcup_{e_1 \triangleright 0 \parallel e_2 \triangleright 0} \mathcal{D}[s] \circ \mathcal{F}[e_1 \triangleright 0 \parallel e_2 \triangleright 0] \cup \mathcal{D}[t] \circ \mathcal{F}[e_1 \triangleright 0 \parallel e_2 \triangleright 0] \\
\mathcal{D}[s_L \parallel s_B] & \triangleq \bigcup_{e_1 \triangleright 0 \parallel e_2 \triangleright 0} \mathcal{D}[\text{assert_sync}(e)] \\
\mathcal{D}[s_L, s_B] & \triangleq \{ (\rho_L, \rho_B) \in X | \exists v_1 \in E_L[e] \rho_L : \exists v_2 \in E_B[e] \rho_B : v_1 \triangleright 0 \land v_2 \triangleright 0 \}
\end{align*}
\]

**Delmas and Miné [2019a,b]**

\[ s_L, s_B \triangleq \text{little-endian and big-endian versions of } s, \text{ } \text{ } \text{ } \text{ } s_L, s_B \]

where \( \mathcal{F}[e_1 \triangleright 0 \parallel e_2 \triangleright 0]X \triangleq \{ (\rho_L, \rho_B) \in X | \exists v_1 \in E_L[e] \rho_L : \exists v_2 \in E_B[e] \rho_B : v_1 \triangleright 0 \land v_2 \triangleright 0 \} \]

\[ s_L, s_B \triangleq \{ (\rho_L, \rho_B) \in X | \exists v \in \mathcal{V} : E_L[e] \rho_L = \{ v \} = E_B[e] \rho_B \} \]
Agenda

1 Motivating example
2 Syntax and concrete semantics
3 Memory model
4 Evaluation
Low-level memory abstraction

Memory model

- **Concrete level**
  
  each program holds values for individual bytes

- **Low-level C programs**
  
  \[
  \begin{align*}
  \text{multi-byte access to memory} \\
  \text{numerical invariants} \\
  \text{byte-level access to encoding} \\
  \text{abuse unions and pointers}
  \end{align*}
  \]
  \[\Rightarrow\]
  need for **scalar** cells

  \[
  \begin{align*}
  \text{cells may overlap}
  \end{align*}
  \]
Low-level memory abstraction

**Memory model**

- **Concrete level**
  - each program holds values for individual bytes
- **Low-level C programs**
  - multi-byte access to memory
  - numerical invariants
  - byte-level access to encoding
  - abuse unions and pointers
  

  \[ \Rightarrow \text{need for scalar cells} \]

  \[ \Rightarrow \text{cells may overlap} \]

---

**The Cells abstract domain**

Miné [2006, 2013]

- Memory as a dynamic collection of cells
  - synthetic scalar variables
    \[ \langle V, o, \tau, \alpha, k \rangle \in \text{Cell} \triangleq V \times \mathbb{N} \times \text{scalar-type} \times \{ L, B \} \times \{ 1, 2 \} \]
  - holding values for memory dereferences discovered during analysis
- Analysis with numerical domain
  (1 dimension / cell)
Analyzing the motivating example with cells

```c
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
assert_sync(y);  // y1 = y2
```

![Diagram](image)
Analyzing the motivating example with cells

```c
u16 x, y;
read_from_network((u8 *)&x, sizeof(x)); // x_0 \leq x_1 \land x_1 \leq x_2
#if __BYTE_ORDER == __LITTLE_ENDIAN
((u8 *)&y)[0] = ((u8 *)&x)[1];
((u8 *)&y)[1] = ((u8 *)&x)[0];
#else
y = x;
#endif
assert_sync(y);  // y_1 \neq y_2
```

\[ x^n_i \triangleq \langle x, n, u8, L, 1 \rangle \]

\[ x^n_2 \triangleq \langle x, n, u8, B, 2 \rangle \]
Analyzing the motivating example with cells

1. `u16 x, y;`
2. `read_from_network((u8 *)&x, sizeof(x));`
3. `# if __BYTE_ORDER == __LITTLE_ENDIAN`
   4. `((u8 *)&y)[0] = ((u8 *)&x)[1];`
   5. `((u8 *)&y)[1] = ((u8 *)&x)[0];`
4. `# else`
   6. `y = x;`
7. `# endif`
8. `assert_sync(y);  // y_1 ? y_2`

\[
x_1^0 = x_2^0 \land x_1^1 = x_2^1
\]

\[
x_1^n \triangleq \langle x, n, \text{u8}, \mathcal{L}, 1 \rangle
\]

\[
x_2^n \triangleq \langle x, n, \text{u8}, \mathcal{B}, 2 \rangle
\]
Analyzing the motivating example with cells

1. `u16 x, y;`
2. `read_from_network((u8 *)&x, sizeof(x));`
3. `
   # if __BYTE_ORDER == __LITTLE_ENDIAN
   ((u8 *)&y)[0] = ((u8 *)&x)[1];
   ((u8 *)&y)[1] = ((u8 *)&x)[0];
   # endif
`
4. `
   # else
   y = x;
   # endif
`

5. `assert_sync(y); // y_1 \neq y_2`

\[ x_1^0 = x_2^0 \land x_1^1 = x_2^1 \]

\[ y_1^0 = x_1^1 \]
\[ y_1^1 = x_1^0 \]

\[ x_1^n \triangleq \langle x, n, u8, L, 1 \rangle \]
\[ x_2^n \triangleq \langle x, n, u8, B, 2 \rangle \]
Analyzing the motivating example with cells

```c
u16 x, y;
read_from_network((u8 *)&x, sizeof(x));

#define __BYTE_ORDER == __LITTLE_ENDIAN
((u8 *)&y)[0] = ((u8 *)&x)[1];
((u8 *)&y)[1] = ((u8 *)&x)[0];
#else
y = x;
#endif
assert_sync(y);  // y1 \neq y2
```

\[ x^n_1 \triangleq \langle x, n, u8, L, 1 \rangle \]

\[ x^n_2 \triangleq \langle x, n, u8, B, 2 \rangle \]

\[ x_2 \triangleq \langle x, 0, u16, B, 2 \rangle \]
Analyzing the motivating example with cells

1. \( \text{u16 } x, y; \)
2. \( \text{read_from_network((u8 *)}\& x, \text{sizeof}(x)); \)
3. \# if \_\_BYTE\_ORDER == \_\_LITTLE\__ENDIAN
4. \( ((\text{u8 *})\& y)[0] = ((\text{u8 *})\& x)[1]; \)
5. \( ((\text{u8 *})\& y)[1] = ((\text{u8 *})\& x)[0]; \)
6. \# else
7. \( y = x; \)
8. \# endif
9. \( \text{assert_sync(y); } // y_1 \equiv y_2 \)

\( x^0 = x_0^0 \land x_1^1 = x_2^1 \)

\( y_1^0 = x_1^1 \)
\( y_1^1 = x_1^0 \)
\( x_2 = 2^8 \times x_2^0 + x_2^1 \land y_2 = x_2 \)

\( x^n_1 \triangleq \langle x, n, u8, L, 1 \rangle \)
\( x^n_2 \triangleq \langle x, n, u8, B, 2 \rangle \)
\( x_2 \triangleq \langle x, 0, u16, B, 2 \rangle \)
Analyzing the motivating example with cells

1. `u16 x, y;`
2. `read_from_network((u8 *)&x, sizeof(x));`
3. `# if __BYTE_ORDER == __LITTLE_ENDIAN`
4. `((u8 *)&y)[0] = ((u8 *)&x)[1];`
5. `((u8 *)&y)[1] = ((u8 *)&x)[0];`
6. `# else`
7. `y = x;`
8. `# endif`
9. `assert_sync(y); // y_1 = y_2`

\[
\begin{align*}
x_1^n & \triangleq \langle x, n, u8, L, 1 \rangle \\
y_1 & \triangleq \langle y, 0, u16, L, 1 \rangle \\
x_2^n & \triangleq \langle x, n, u8, B, 2 \rangle \\
x_2 & \triangleq \langle x, 0, u16, B, 2 \rangle
\end{align*}
\]
Optimizing the memory model for the common case

| Complex invariants | $\Rightarrow$ | Expressive numerical domain? |
|-------------------|----------------|----------------------------|
| Program invariants and cell constraints |
| $x_0^0 = x_0^2 = y_1^1$ | $x_1^1 = x_1^2 = y_1^0$ | $y_2 = x_2$ |
| $x_1 = x_0^0 + 2^8 x_1^1$ | $y_1 = y_0^0 + 2^8 y_1^1$ | $y_1 = y_2$ |
| $x_2 = 2^8 x_0^0 + x_1^1$ | $y_2 = 2^8 y_0^0 + y_1^1$ | $y_2 = y_2$ |

**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 $\implies$ expressive numerical domain?

- Program invariants and cell constraints

$$
\begin{align*}
x_1^0 &= x_2^0 = y_1^1 \\
x_1^1 &= x_2^1 = y_1^0 \\
x_1 &= x_1^0 + 2^8 x_1^1 \\
y_1 &= y_1^0 + 2^8 y_1^1 \\
x_2 &= 2^8 x_2^0 + x_2^1 \\
y_2 &= 2^8 y_2^0 + y_2^1
\end{align*}
$$

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

Sharing cells in the memory environment

- **Single** representation for two cells
  - from different program versions
  - at same memory location
  - holding equal values

- A bi-cell is $Bicell \triangleq Cell \cup (\overline{Cell} \times \overline{Cell})$
  - either a cell
  - or a pair of cells holding equal values (shared bi-cell)
Analyzing the motivating example: **from cells to bi-cells**

1. `u16 x, y;`
2. `read_from_network((u8 *)&x, sizeof(x));`
3. `# if __BYTE_ORDER == __LITTLE_ENDIAN`
   4. `((u8 *)&y)[0] = ((u8 *)&x)[1];`
   5. `((u8 *)&y)[1] = ((u8 *)&x)[0];`
4. `# else`
   7. `y = x;`
5. `# endif`
   8. `# endif`
9. `assert_sync(y); // y1 \not= y2`

\[ x^0 = x^0_1 \land x^1_1 = x^1_2 \]
\[ y^0_1 = x^1_1 \]
\[ y^1_1 = x^0_1 \]
\[ x_2 = 2^8 \times x^0_2 + x^1_2 \land y_2 = x_2 \]
\[ y_1 = y^0_1 + 2^8 \times y^1_1 \]

\[ x^n_1 \triangleq \langle x, n, u8, L, 1 \rangle \]
\[ y_1 \triangleq \langle y, 0, u16, L, 1 \rangle \]
\[ x^n_2 \triangleq \langle x, n, u8, B, 2 \rangle \]
\[ x_2 \triangleq \langle x, 0, u16, B, 2 \rangle \]
Analyzing the motivating example: from cells to bi-cells

```c
1 u16 x, y;
2 read_from_network((u8 *)&x, sizeof(x));
3 # if __BYTE_ORDER == __LITTLE_ENDIAN
4   ((u8 *)&y)[0] = ((u8 *)&x)[1];
5   ((u8 *)&y)[1] = ((u8 *)&x)[0];
6 # else
7   y = x;
8 # endif
9 assert_sync(y); // y1 \equiv y2
```

\[
x_1^0 \triangleq \langle x, n, u8, L, 1 \rangle
\]
\[
x_2^0 \triangleq \langle x, n, u8, B, 2 \rangle
\]
\[
x_2 \triangleq \langle x, 0, u16, B, 2 \rangle
\]
\[
y_1 \triangleq \langle y, 0, u16, L, 1 \rangle
\]
\[
y_2 \triangleq \langle y, 0, u16, B, 2 \rangle
\]
Analyzing the motivating example: from cells to bi-cells

```c
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
assert_sync(y); // y_1 \equiv y_2
```

\[
x_1^n \triangleq \langle x, n, \text{u8}, L, 1 \rangle
\]
\[
x_2^n \triangleq \langle x, n, \text{u8}, B, 2 \rangle
\]
\[
x_2 \triangleq \langle x, 0, \text{u16}, B, 2 \rangle
\]
\[
y_1 \triangleq \langle y, 0, \text{u16}, L, 1 \rangle
\]
\[
y_2 \triangleq \langle y, 0, \text{u16}, B, 2 \rangle
\]
Agenda

1 Motivating example
2 Syntax and concrete semantics
3 Memory model
4 Evaluation
Implementation

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

Prototype abstract interpreter
- 3,000 lines of OCaml source code
  - 19% double program management and iterators
  - 45% memory domain
  - 36% symbolic predicate domain (see paper)
- leverages 31,000 lines of Mopsa (excluding parsers)

http://mopsa.lip6.fr/
## Benchmarks

| Origin      | Name    | LOC  | Time   | Revision | Result |
|-------------|---------|------|--------|----------|--------|
| Open Source | GENEVE  | 218  | 1 s    | 2014-1   | ![Result](https://example.com/icon)
|             |         |      |        | 2014-2   | ![Result](https://example.com/icon) |
|             |         |      |        | 2016     | ![Result](https://example.com/icon) |
|             |         |      |        | 2017     | ![Result](https://example.com/icon) |
|             | MLX5    | 125  | 155 ms | 2017     | ![Result](https://example.com/icon) |
|             |         |      |        | 2020-1   | ![Result](https://example.com/icon) |
|             |         |      |        | 2020-2   | ![Result](https://example.com/icon) |
|             | Squashfs| 110  | 150 ms | 2020-1   | ![Result](https://example.com/icon) |
|             |         |      |        | 2020-2   | ![Result](https://example.com/icon) |
| Industrial  | Module S| 300 K| 9.7 h  | 2020     | ![Result](https://example.com/icon) |
|             | Module A| 1 M  | 20.4 h | 2020     | ![Result](https://example.com/icon) |
|             |         |      |        | 2021     | ![Result](https://example.com/icon) |

**Disclaimer:**

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

### Static analysis of Endian portability

1. **Novel concrete collecting semantics**
   - two versions of a program
   - platforms with different endiannesses

2. **Joint memory abstraction**
   - relations between little- and big-endian memories

3. **Prototype static analyzer**
   - scale to large industrial software
   - with zero false alarms

---

Delmas, Ouadjaout, Miné

Endian Portability Analysis
Conclusion

Static analysis of Endian portability

- Novel concrete collecting semantics
  - two versions of a program
  - platforms with different endiannesses
- Joint memory abstraction
  - relations between little- and big-endian memories
- Prototype static analyzer
  - scale to large industrial software
  - with zero false alarms

More in the paper: Symbolic predicate domain

- Relations between bytes of variables of the two programs
- Established by bitwise arithmetics
- Near-linear cost
Conclusion

**Static analysis of Endian portability**

1. Novel concrete collecting semantics
   - two versions of a program
   - platforms with different endianness

2. Joint memory abstraction
   - relations between little- and big-endian memories

3. Prototype static analyzer
   - scale to large industrial software
   - with zero false alarms

**Future work**

- **Industrialize** (certification of avionics simulation fidelity)
- **Extend**
  - **Portability** (layouts of C types, sizes of machine integers)
  - **Patches** modifying data-types
Backup slides
D. Delmas and A. Miné. Analysis of Program Differences with Numerical Abstract Interpretation. In PERR 2019, Prague, Czech Republic, Apr. 2019a.

D. Delmas and A. Miné. Analysis of Software Patches Using Numerical Abstract Interpretation. In B.-Y. E. Chang, editor, Proc. of the 26th International Static Analysis Symposium (SAS’19), volume 11822 of Lecture Notes in Computer Science, pages 225–246, Porto, Portugal, Oct. 2019b. Bor-Yuh Evan Chang, Springer.

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 2006.

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