Field-Sensitive Value Analysis of Embedded C Programs with Union Types and Pointer Arithmetics

Antoine Miné

École Normale Supérieure, Paris, France
Introduction
Introduction

**Goal:** Static value-analysis

Determine the set of values each variable can take at each program point

- statically (at compile-time),
- automatically (no user intervention),
- at the source level (not a model),
- in a **sound, exhaustive** way (but losing completeness).

**Main Application:**

Check for the absence of **Run-Time Errors** (RTE):

- arithmetic errors: overflows, divisions by zero, invalid floating-point numbers,
- memory errors: invalid pointers, out-of-bound array accesses.

Due to incompleteness we may raise **false alarms**.

The analysis should be customizable to achieve zero false alarms on specific codes.
Target Programs

We focus on large embedded critical C code.

Simplifying Hypotheses:

- No dynamic memory allocation.
- No recursivity.
- No call to libraries.

Difficult Points Already addressed in previous work [Astrée: Blanchet et al. 03]

- Large programs (100 KLoc), huge state (10 K variables), and loops ($10^6$ iterations).
- Floating-point and modular integer computations.

New Difficult Points Tackled with here:

- Union types, pointer arithmetics, pointer casts.
- Dependency towards the low-level representation of data in memory.
Previous Work on Numerical Value Analysis

To find numerical invariants, we rely on **numerical abstract domains**. Able to abstract sets of environments in \( \mathbb{R}^n \).

\[
\bigwedge_i X_i \in [a_i, b_i] \\
\text{intervals [Cousot et al. 77]}
\]

\[
\bigwedge_{ij} X_i \pm X_j \leq c_{ij} \\
\text{octagons [Miné 01]}
\]

**Main Requirement**

There must exist a partition of the memory into \( \mathbb{R} \)-dimensions.

- Works with integer & floating-point types (one variable = one dimension).
- Can be extended to arrays and structures types (recursive splitting).
- Can cope with references (teaming with an alias analysis).
- Does not work when pointer casts or union types occur.
**Issues: Union Types**

**Problem:** union types induce complex aliasing at the byte level.

**Example:** register structure of an x86 Intel processor.

```c
union {
    struct { uint8 al, ah, bl, bh; } b;
    struct { uint16 ax, bx; } w;
} r;

r.w.ax = 0; r.b.ah = 2; ··
```

- `r.w.ax`, `r.b.al` and `r.b.ah` are not independent:
  - Modifying `ax` modifies `ah` and `al` (at ·, `al = 0`).
  - Modifying `ah` modifies `ax`: (at ·, `ax = 512`).

⇒ fields cannot be assigned dimensions in a numerical abstract domain.

**Note:** The code makes unportable assumptions about the memory layout. These are formalized in Application Binary Interfaces (ABI).
Issues: Pointer Arithmetics and Casts

Problem: pointer arithmetics breaks array bound-checking.

Example: arrays embedded within structures.

```c
struct { char a[2]; char b; } x, y;
*(x.a+2) = 1; // OK
*(x.a+3) = 1; // Error!
```

Problem: contiguous spans of memory can be given an arbitrary type dynamically.

Example: polymorphic memory copy function, one byte at a time.

```c
memcpy(char* d, char* s, int n) {
    for (int i=0; i<n; i++) d[i] = s[i];
}
unsigned char b[20];
struct { short p; ... } x;
memcpy(&x, b+1, sizeof(x));
x.p++;
```
Proposed Solutions

Solutions already exist, but are not adapted to our purpose.

♦ **Restrict the input language** (e.g., CCured [Necula et al. 02]). Not acceptable by all end-users.

♦ **Field-insensitive** or partially field-sensitive analyses with
  • static partitioning [Venet 04], or
  • dynamic partitioning [Yong et al.99] (when aliasing is detected).
  Not precise enough for exhaustive RTE checking.

♦ **One** memory layout is **fixed**; memory accesses that do not fit can be
  • abstracted as “don’t know” [Balakrishnan et al. 04] (not precise enough to deal with polymorphism)
  • or synthesized for existing information (stresses numerical abstract domains)

**Our Solution:**

• Fully **field-sensitive** analysis, **parameterized** by a numerical abstract domain.
• Discovers memory layouts **dynamically**, allowing **overlapping** fields in layouts.
• Performs **reductions** to account for byte-level aliasing between overlapping cells.
Plan

♦ Informal presentation of the analysis.

♦ Formalization (sketch) within the Abstract Interpretation framework.

♦ Preliminary experimental results.
Principles

**Abstract Environments** pairs composed of:

- a set \( C \) of **cells** of some **scalar type** \( \tau \)

  \[
  \tau ::= \text{(unsigned)} \text{char} | \text{short} | \text{int} | \text{long} | \text{long long} | \text{float} | \text{double} | \text{long double}
  \]

  spawning (possible overlapping) bytes within a variable \( V \in \mathcal{V} \)

- a **numerical abstract element**, abstracting \( \mathcal{P}(C \rightarrow \mathbb{R}) \).

**Abstraction Operations** three steps:

- cell realization: enrich \( C \),
- evaluation within the numerical abstract domain,
- side-effects: prune \( C \).
Reading **creates** cells, with optional **reduction**.

```c
union { struct { uint8 al, ah, bl, bh; } b; struct { uint16 ax, bx; } w; } r;

r.w.ax = X;
if (!r.b.ah) r.b.bl = r.b.al; else r.b.bh = r.b.al;
r.b.al = Y;
```

- enrich $C$ with a cell $ah$ of type unsigned char at offset 1,
- initialize $ah$ by reduction with existing cell $ax$:
  $$ah \leftarrow ax/256$$  \hspace{2cm} (little-endian ABI)
- evaluate the guard $ah = 0$

$\implies$ we get $X \in [0, 255]$ at ●  \hspace{1cm} (given a sufficiently precise abstract domain)

**When cells overlap, both invariants are true!**
Running Example – Control Flow Joins

Cell-sets are unified by enrichment (with reduction).

```c
union { struct { uint8 al,ah,bl,bh; } b; struct { uint16 ax,bx; } w; } r;
r.w.ax = X;
if (!r.b.ah) r.b.bl = r.b.al; else r.b.bh = r.b.al; •
r.b.al = Y;
```

- enrich left branch argument with bh, right branch argument with bl,
- perform control flow join on unified numerical abstract elements.

Loss of precision because \((x \land y) \lor (x' \land y') \neq (x \lor x') \land (y \lor y')\) in general.
Running Example – Memory Writes

Writes have **destructive** side-effects on overlapping cells.

```
union { struct { uint8 al,ah,bl,bh; } b; struct { uint16 ax,bx; } w; } r;
r.w.ax = X;
if (!r.b.ah) r.b.bl = r.b.al; else r.b.bh = r.b.al;
r.b.al = Y;
```

- enrich $C$ with target cells, if necessary (as for read)
- perform assignment on the numerical abstract element,
- destroy $ax$ that is modified by the assignment, but not targeted.

**Note:** $ax$ will be re-synthesized from $al$ and $ah$ accurately when next read.
Pointers and Pointer Arithmetics

**Pointer Semantics**  
pointer values as pairs:

- a *base* variable, \( V \in \mathcal{V} \),
- a numerical *offset* in \([0, \text{sizeof}(V)]\).

**Pointer Analysis**

Performed *together* with numerical value analysis.

- Pointer arithmetics is broken statically to the **byte level**, reduced to integer arithmetics  
  e.g.: \( p + i \rightarrow p + i \times \text{sizeof}(\ast p) \)

- We consider one new type of cells \( \text{ptr} \) representing uniformly all pointer types.

- The two components of \( \text{ptr} \) cells are abstracted separately:
  - the *base* component is represented as a set of variables,
  - the *offset* is allocated a *dimension* in the numerical abstract domain.

Allows relationships between pointer and integer variables (relational abstract domains). But not between bases and offsets...
Memory Equality Predicates

Memory Copy: puts pressure on the numerical abstract domain

```c
memcpy(char* d, char* s, int n) {
    for (int i=0; i<n; i++) d[i] = s[i];
}
short x, y = 0;
memcpy(&x, &y, sizeof(x));
... = x;
```

At ⊗, we have: 
\[ x = 256 \times \left( y/256 \right) + \left( y \mod 256 \right) \]
The numerical domain may not know that this means \( x = y \)!

**Solution:** helper domain tracking memory equality predicates

- track equality predicates between spans of memory:
  \[ P(x, y, a, b, o) := \forall i \in [a, b], ((\text{char}*)x)[i] = ((\text{char}*)y)[i + o] \]
- perform extra reductions from \( y \) to \( x \).

Field-Sensitive Value Analysis of Embedded C Programs with Union Types and Pointer Arithmetics 13/23
Formalization (sketch)
Methodology

Abstract Interpretation [Cousot 77]:

- general theory of the approximation of the semantics of programs,
- allows deriving static analyses that are sound by construction,
- possible to compose abstractions (e.g., memory domain and existing numerical ones).

Methodology

♦ Define a concrete semantics on a concrete domain \( D \)
  - very precise: contains all the properties of interest,
  - undecidable,
  for value analysis, it collects reachable states.

♦ Define an abstract domain, i.e.:
  - a set of computer-representable abstract values \( D^\# \) with a partial order \( \sqsubseteq^\# \),
  - a monotonic concretization \( \gamma : D^\# \to D \),
  - computable versions \( F^\# : D^\# \to D^\# \) of concrete operators \( F : D \to D \),
  - respect the soundness condition: \((F \circ \gamma)(X^\#) \subseteq (\gamma \circ F^\#)(X^\#)\).
Non-Standard Concrete Memory Domain

**Definitions:**

- **variables** \( \mathcal{V} \)
- **scalar types** \( \tau \) ::= \( \text{int} \mid \text{float} \mid \text{ptr} \mid \cdots \)
- **values of type** \( \tau \) \( \mathcal{V}_\tau \subseteq \mathbb{R} \cup (\mathcal{V} \times \mathbb{N}) \cup \{\emptyset, \omega\} \)
- **byte locations** \( \mathcal{B} \) = \{ (V, i) \mid V \in \mathcal{V}, 0 \leq i < \text{sizeof}(V) \}\)
- **byte values** \( \mathbb{V} \) = \{ (\tau, b, v) \mid 0 \leq b < \text{sizeof}(\tau), v \in \mathcal{V}_\tau \}\)
  - \( b \)-th byte of value \( v \) encoded in type \( \tau \)
- **memory state** \( \mathcal{M} \) = \( \mathcal{B} \mapsto \mathbb{V} \)
- **concrete domain** \( \mathcal{D} \) = \( \mathcal{P}(\mathcal{M}) \)

**Features:** low-level, but not too low-level

- ♦ byte-based representation: allows mapping low-level C constructs,
- ♦ high-level byte values (at the lowest-level \( \mathbb{V} = [0, 255] \))
  - allows abstracting away some semantic features (e.g., absolute addresses of variables),
  - keeps typing information that will be useful in the abstract.
Concrete Memory Operations

**Memory Writes**  (easy)
Write value $x \in \mathbb{V}_\tau$ of type $\tau$ at location $(V, o)$:

$$\text{store } (\tau, b, x) \text{ at location } (V, o + b) \text{ for } 0 \leq b < \text{sizeof}(\tau)$$

**Memory Reads**  (more involved)
To read a data of type $\tau$ at location $(V, o)$:

- first, fetch byte values at positions $(V, o + b), 0 \leq b < \text{sizeof}(\tau)$
- then, use a **recomposition function** $\phi_\tau : \mathbb{V}^{\text{sizeof}(\tau)} \rightarrow \mathcal{P}(\mathbb{V}_\tau)$:

<table>
<thead>
<tr>
<th>$\phi_\tau \langle (\tau_0, b_0, v_0), \ldots \rangle$</th>
<th>$\phi_{\text{unsigned char}} \langle (\tau, b, v) \rangle$</th>
<th>$\phi_{\text{unsigned t}} \langle x_0, \ldots \rangle$</th>
<th>$\phi_{\text{signed t}} \langle x \rangle$</th>
<th>$\phi_\tau \langle \cdots \rangle$</th>
</tr>
</thead>
<tbody>
<tr>
<td>${ v }$ if $\forall i, v_i = v, \tau_i = \tau, b_i = i$</td>
<td>${ v/(256^b) \mod 256 }$ if $\tau$ is integer</td>
<td>${ \sum_i 2^{256 \times i} \times y_i \mid y_i \in \phi_{\text{unsigned char}} \langle x_i \rangle }$</td>
<td>${ w \mid w + 2^{\text{sizeof}(t)} \mathbb{Z} \cap \phi_{\text{unsigned t}} \langle x \rangle \neq \emptyset, w \in [-2^{\text{sizeof}(t)} - 1, 2^{\text{sizeof}(t)} - 1] }$</td>
<td>$\mathbb{V}_\tau$ in all other cases</td>
</tr>
</tbody>
</table>

The $\phi_\tau$ are **non-deterministic**, allowing definitions with **various precision & generality**:

- we could refine $\phi_\tau$ to take into account the binary encoding of floats,
- we could coarsen $\phi_\tau$ to make it endianess-independent.
Abstract Memory Semantics

Cell-Based Abstraction

- Let $D_V^\#(C)$ be a numerical domain, enriched with base pointer information.
- $C_{all} \overset{\text{def}}{=} \{(V, i, \tau) \mid V \in V, 0 \leq i, i + \text{sizeof}(\tau) \leq \text{sizeof}(V)\}$
- $D^\# \overset{\text{def}}{=} \{(C, X^\#) \mid C \subseteq C_{all}, X^\# \in D_V^\#(C)\}$
- $\gamma(\{(V_1, i_1, \tau_1), \ldots, (V_n, i_n, \tau_n), X^\#\}) \overset{\text{def}}{=} \{ \rho \in D \mid \forall x_1 \in \phi_{\tau_1} \langle \rho(V_1, i_1), \ldots, (V_1, i_1 + \text{sizeof}(\tau_1) - 1) \rangle, \ldots, \forall x_n \in \phi_{\tau_n} \langle \rho(V_n, i_n), \ldots, (V_n, i_n + \text{sizeof}(\tau_n) - 1) \rangle, (x_1, \ldots, x_n) \in \gamma_V(X^\#) \}$
- $(C_1, X_1^\#, B_1^\#) \sqsubseteq^\# (C_2, X_2^\#, B_2^\#) \iff C_1 = C_2 \land X_1^\# \sqsubseteq^\#_n X_2^\#$

Notes:

- we have an **intersection semantics** on overlapping cells,
- cells can be soundly removed from $C$ (results in an overapproximation)
- adding a cell to $C$ may restrict the byte values; it must be initialized according to the values of existing cells and $\phi$. 
Abstract Memory Semantics

Memory Equality Predicates

- \( D^\# \overset{\text{def}}{=} V \rightarrow ((\mathbb{N} \times V \times \mathbb{N} \times \mathbb{N}) \cup \{ \top^\# \}) \)

- \( \gamma(\epsilon) \overset{\text{def}}{=} \{ \rho \in D \mid \forall V \in V, \epsilon(V) = (s, W, d, l) \implies \forall 0 \leq i < l, \rho(V, s + i) = \rho(W, d + i) \} \)

- \( X^\# \sqsubseteq^\# Y^\# \iff \forall V \in V, Y^\#(V) = \top^\# \lor \{ X^\# = (s, W, d, l) \land Y^\# = (s', W, d', l') \land s - d = s' - d' \land [s, s + l] \subseteq [s', s' + l'] \} \)
Results
The Astrée Analyzer

**Original Astrée**

- Static analyzer developed at the ENS
- Checks for **run-time errors** in embedded reactive C code.
- Aimed towards 0 alarm by manual specialization of the analyzer.
- The original memory model could only deal with:
  - well-structured memory (arrays, struct, no union),
  - limited pointers (aliases, no arithmetics, no casts).

**New Memory Abstraction** incorporated painlessly within Astrée.

- All numerical domains reused, a new one added for offsets (congruence).
- No change needed in iteration strategy.

Goals achieved:

- **non-regression** (in both efficiency and accuracy)
- prove RTE-absence on user-provided **small code samples** using unions and pointers
- start analysing **real-life codes** using unions and pointers
Experimental Results

Non-Regression

♦ Critical control-command software for A340 and A380 Airbus aircrafts.
♦ No union nor pointer arithmetics.

<table>
<thead>
<tr>
<th></th>
<th>old domain</th>
<th>new domain</th>
<th>both</th>
</tr>
</thead>
<tbody>
<tr>
<td>lines</td>
<td>time</td>
<td>memory</td>
<td>time</td>
</tr>
<tr>
<td>9 500</td>
<td>82 s</td>
<td>0.2 GB</td>
<td>99 s</td>
</tr>
<tr>
<td>70 000</td>
<td>62 mn</td>
<td>1.0 GB</td>
<td>63 mn</td>
</tr>
<tr>
<td>226 000</td>
<td>4 h 57</td>
<td>1.6 GB</td>
<td>4 h 42 mn</td>
</tr>
<tr>
<td>400 000</td>
<td>11 h 04 mn</td>
<td>3.0 GB</td>
<td>11 h 46 mn</td>
</tr>
</tbody>
</table>

Results:

• Similar time and memory consumption.

• Same number of alarms: absence of RTE still proved!
Experimental Results

Newly Analyzed Codes

Smaller but **more complex** codes, hand-written, use many unportable C features:

- union types implement dynamically typed, polymorphic sum types,
- access to memory-mapped I/O through pointers,
- custom allocation within large statically-allocated buffers,
- custom polymorphic memory initialization and copy functions.

<table>
<thead>
<tr>
<th>source</th>
<th>lines</th>
<th>time</th>
<th>memory</th>
<th>alarms</th>
</tr>
</thead>
<tbody>
<tr>
<td>samples</td>
<td>245</td>
<td>&lt; 1 s</td>
<td>3 MB</td>
<td>0</td>
</tr>
<tr>
<td>end-user #1</td>
<td>35 000</td>
<td>12 mn</td>
<td>212 MB</td>
<td>22</td>
</tr>
<tr>
<td></td>
<td>46 000</td>
<td>16 mn</td>
<td>271 MB</td>
<td>84</td>
</tr>
<tr>
<td>end-user #2</td>
<td>92 000</td>
<td>3 h 17 mn</td>
<td>3.2 GB</td>
<td>71</td>
</tr>
<tr>
<td></td>
<td>184 000</td>
<td>4 h 55 mn</td>
<td>1.1 GB</td>
<td>36</td>
</tr>
</tbody>
</table>

**Results:**

- No false alarms on representative code samples.
- Few enough alarms on real-life codes (could be checked manually).
Future Work

**Achieve Zero Alarms on Real-Life Codes**
Find the origin of alarms and refine the analyzer.

*Note:* alarms may not be related to our memory abstraction.

**Offset Analysis**

Current numerical domains include: octagons $\pm X \pm Y \leq c$, simple congruences $X \in a\mathbb{Z} + b$. Not expressive enough to find some common invariants on offsets!

*Note:* array slices give rise to offsets of the form: $\sum_i [a_i, b_i] \times c_i$.

**Memory Summarization**

- **collapse** cells in the abstract [Gopan et al. 05]
- **focus** cells **dynamically** to avoid weak updates,
- infer **abstract relationship** between offsets and memory contents (as in parametric predicate abstraction [Cousot 03])
Summary

We addressed the problem of field-sensitive value analysis:

• of C code featuring union types, pointer arithmetics and pointer casts,
• of code relying on fine hypotheses on the memory layout,
• with a level of precision that allows checking for RTE.

We presented:

♦ a non-standard concrete semantics of memory operations in C,
♦ precise and sound abstractions compatible with numerical abstract domains,
♦ encouraging preliminary experimental results within the Astrée analyzer.