Validation of Memory Accesses Through Symbolic Analyses

Laure Gonnord

Lyon1/LIP

12 mars 2015 - Aric Seminar

Inspiration : OOPSLA'14 slides :

VALIDATION OF MEMORY ACCESSES
THROUGH SYMBOLIC ANALYSES

Henrique Nazare
Izabela Maffra
Willer Santos
Leonardo Oliveira
Fernando Quintão
Laure Gonnord

http://homepages.dcc.ufmg.br/~fernando/publications/presentations/OOPSLA14.pdf

1. Motivation and big picture
   - Overview
2. Technical context LLVM
3. Symbolic Range Analysis
4. A bit on the other analyses
5. Experimental results
6. Conclusion

Goal : Safety

Prove that (some) memory accesses are safe :

```
int main() {
   int v[10];
   v[0] = 0;        ✓
   return v[20];    ✗
}
```

- Fight against bugs and overflow attacks.

Contributions (OOPSLA'14)

- A technique to prove that (some) memory accesses are safe :
  - Less need for additional guards.
  - Based on abstract interpretation.
  - Precision and cost compromise.
- Implemented in LLVM-compiler infrastructure :
  - Eliminate 50% of the guards inserted by AddressSanitizer
  - SPEC CPU 2006 17% faster

Our key insight : Symbolic (parametric) Analyses

```
int main(int argc, char** a) {
  char* p = malloc(argc);
  int i = 0;
  while (i < argc) {
    p[i] = 0;
    i++;
  }
  return 0;
}
```

W(p) = [0, argc − 1].

R(i) = [0, argc − 1]

- $R(i) \subseteq W(p)$ thus $p[i]$ is **safe**.

1 Motivation and big picture
- Overview

2 Technical context LLVM

3 Symbolic Range Analysis

4 A bit on the other analyses

5 Experimental results

6 Conclusion

A bit on sanitizing memory accesses

Different techniques : but all have an overhead.

Ex : Address Sanitizer

- Shadow every memory allocated : 1 byte $\rightarrow$ 1 bit (allocated or not).
- Guard every array access : check if its shadow bit is valid.
  - slows down SPEC CPU 2006 by 25%

- We want to **remove these guards**.

Green Arrays : overview 1/2

```
1.  int main(int argc, char** argv) {
2.    int size = argc + 1;
3.    char* buf = malloc(size);
4.    unsigned index = 0;
5.    scanf("%u", &index);
6.    if (index < argc) {
7.      buf[index] = 0;
8.    }
9.    return index;
10. }
```

Green Arrays : overview 2/2

Symbolic Range Analysis: finds the lower and upper values that variables can assume

Symbolic Region Analysis: finds the lower and upper values that a pointer can address

Integer Overflow Analysis: Which arithmetic operations can overflow?

Any address from buf + 0 to buf + argc is safe!

Inside the branch index is at least 0 and at most argc-1

We know that "argc - 1" is less than argc

As long as we do not have integer overflows!

1. Motivation and big picture
2. **Technical context LLVM**
3. Symbolic Range Analysis
4. A bit on the other analyses
5. Experimental results
6. Conclusion

A bit on LLVM

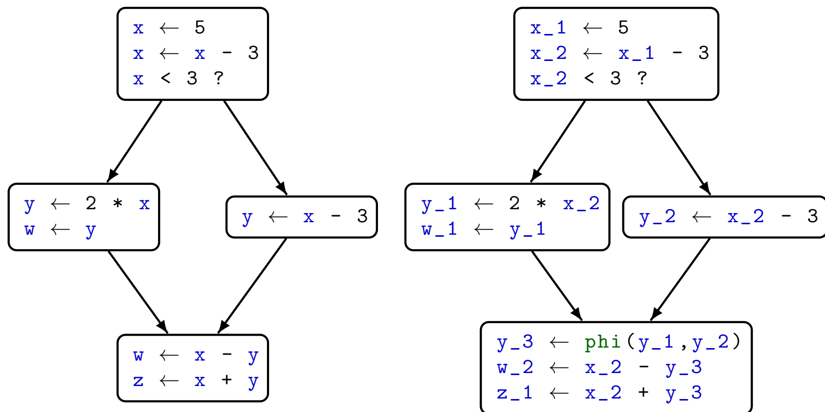
LLVM is a **compiler infrastructure** :

- Open source
- Various frontends (C, C++, Fortran)
- Various code generators (x86, ...)

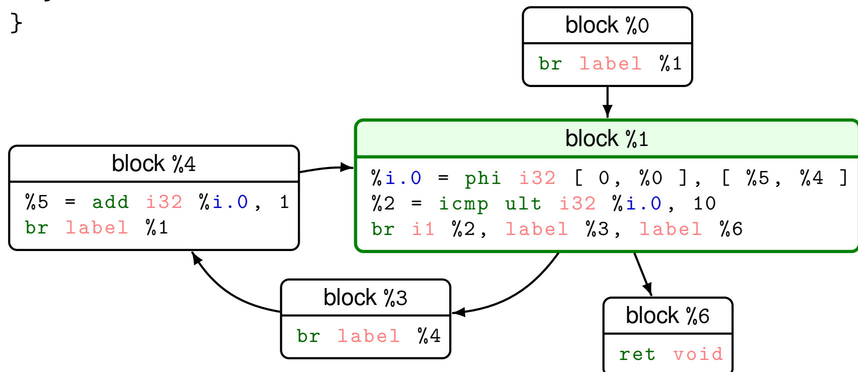
Writing optimisations is easier :

- A unique IR (**intermediate representation**)
- C++ iterators (functions, blocks, ...)

LLVM representation : SSA form



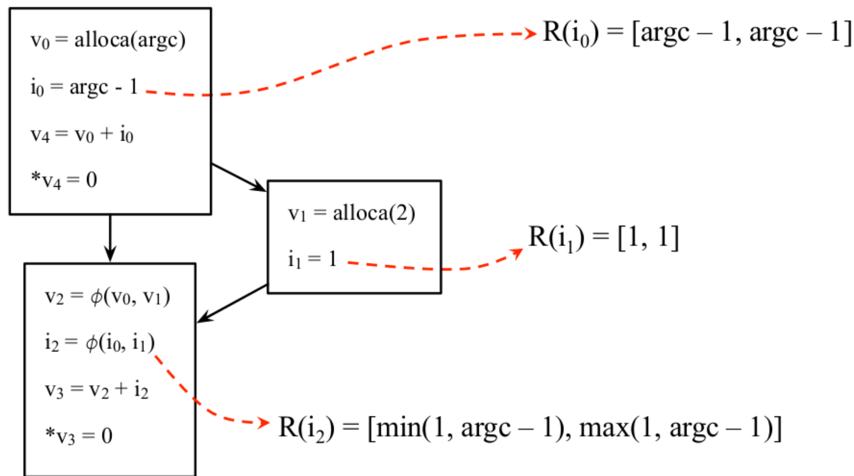
LLVM representation : a toy example



Credits G. Radanne

1. Motivation and big picture
2. Technical context LLVM
3. Symbolic Range Analysis
4. A bit on the other analyses
5. Experimental results
6. Conclusion

Symbolic Ranges (SRA) : Goal



SRA on SSA form : a sparse analysis

- An abtract interpretation-based technique.
- Very similar to classic range analysis.
- One abstract value (R) **per variable** : sparsity.

▶ Easy to implement (simple algorithm, simple data structure).

SRA on SSA form : constraint system

$$v = \bullet \quad \Rightarrow \quad R(v) = [v, v]$$

$$v = o \quad \Rightarrow \quad R(v) = R(o)$$

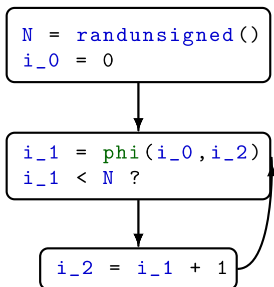
$$v = v_1 \oplus v_2 \quad \Rightarrow \quad R(v) = R(v_1) \oplus^I R(v_2)$$

$$v = \phi(v_1, v_2) \quad \Rightarrow \quad R(v) = R(v_1) \sqcup R(v_2)$$

$$\text{other instructions} \quad \Rightarrow \quad \emptyset$$

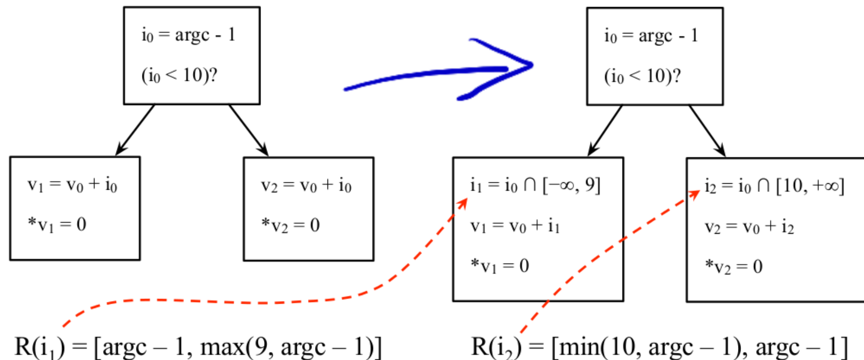
$\oplus^I$: abstract effect of the operation $\oplus$ on two intervals.
$\sqcup$: convex hull of two intervals. ▶ All these operation are performed symbolically thanks to **GiNaC**

SRA on SSA form : an example



- $R(i_0) = [0, 0]$
- $R(i_1) = [0, +\infty]$
- $R(i_2) = [1, +\infty]$

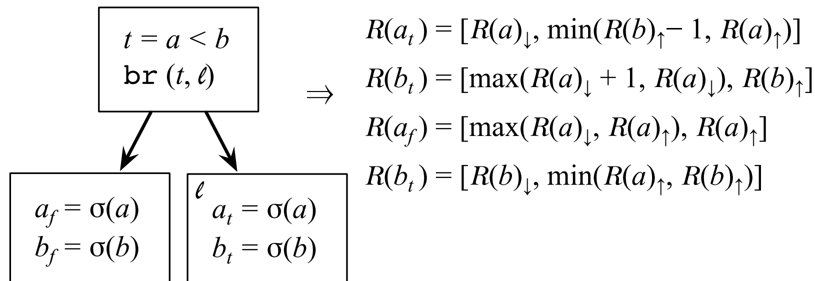
Improving precision of SRA : live-range splitting 1/2



- e-SSA form.

Improving precision of SRA : live-range splitting 2/2

Rule for live-range splitting :



- All simplications are done by GiNaC.

SRA + live-range on an example

```
N = randunsigned()
i_0 = 0
```

```
i_1 = phi(i_0,i_2)
i_1 < N ?
```

```
i_t = sigma(i_1)
i_2 = i_t + 1
```

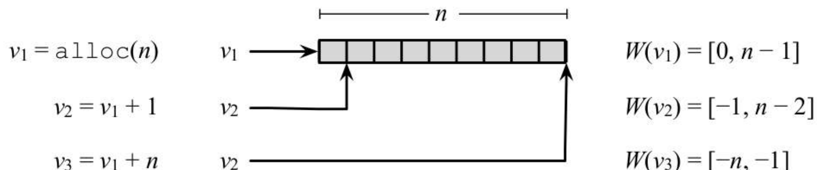
$$R(i_t) = [R(i_1)\downarrow, min(N-1, R(i_1)\uparrow)]$$

- $R(i_0) = [0, 0]$
- $R(i_1) = [0, min(N, 0)]$

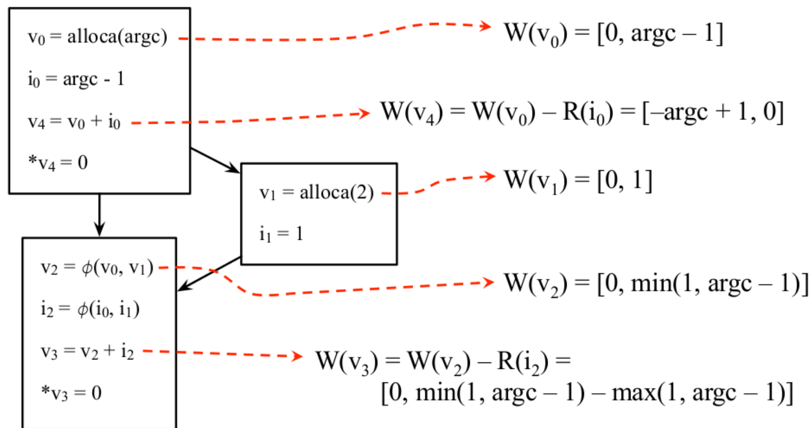
1. Motivation and big picture
2. Technical context LLVM
3. Symbolic Range Analysis
4. A bit on the other analyses
5. Experimental results
6. Conclusion

Symbolic regions 1/2 : Goal

Compute (an overapproximation of) the range of **valid accesses** from base pointers :

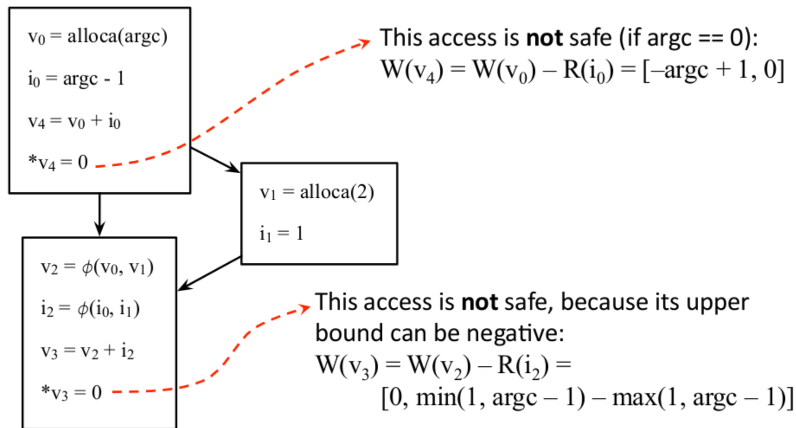


Symbolic regions 2/2 : An example



Safety : result

If $0 \in W(p)$, then $*p$ is **safe**, else **DK**



Overflows 1/2

```
int main(int argc, char** argv) {
  int index = argc + 1;
  int size = index * index;
  char* buf = malloc(size);
  return buf[index];
}
```

index * index may wrap around.

Do you know what malloc will return?

Because we manipulate symbols, "argc + 1 < (argc + 1) * (argc + 1)" only in the absence of integer overflows

Overflows 2/2

- We find every arithmetic operation that may influence memory **allocation** or memory **indexing**.

```
int main(int argc, char** argv) {
  int index = argc + 1;
  int size = index * index;
  char* buf = malloc(size);
  return buf[index];
}
```

We find them via program slicing

▶ We **instrument the code** to detect overflows.

1. Motivation and big picture
2. Technical context LLVM
3. Symbolic Range Analysis
4. A bit on the other analyses
5. **Experimental results**
6. Conclusion

Experimental setup

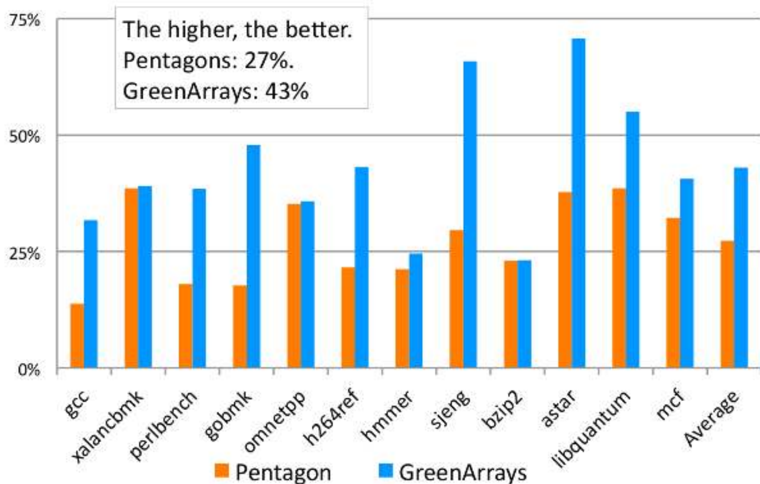
- **Implementation**: LLVM + AddressSanitizer
- **Benchmarks**: SPEC CPU 2006 + LLVM test suite
- **Machine**: Intel(R) Xeon(R) 2.00GHz,with 15,360KB of cache and 16GB or RAM
- **Baseline**: Pentagons
  - Abstract interpretation that combines "less-than" and "integer ranges".†

```
int i = 0;
unsigned j = read();
if (...)
    i = 9;
if (j < i)
    ...
```

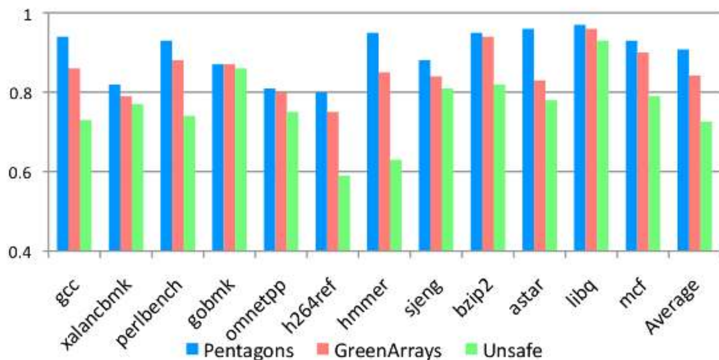
P(j) = (less than {i}, [0, 8])

†: Pentagons: A weakly relational abstract domain for the efficient validation of array accesses, 2010, Science of Computer Programming

Percentage of bound checks removed



Runtime improvement



The lower the bar, the faster. Time is normalized to AddressSanitizer without bound-check elimination. Average speedup: Pentagons = 9%. GreenArrays = 16%.

1. Motivation and big picture
2. Technical context LLVM
3. Symbolic Range Analysis
4. A bit on the other analyses
5. Experimental results
6. Conclusion

In the paper

A complete formalisation of all the analyses :

- Concrete and abstract semantics.
- Safety is proved.
- Interprocedural analysis.

Conclusion

Static analyses for memory accesses :

- New toolchain.
- Reusable analyses.

▶ `https://code.google.com/p/ecosoc/`

Work in progress : improving precision of the symbolic range analysis (with D. Monniaux).