PROGRAMMING LANGUAGES LABORATORY

Universidade Federal de Minas Gerais - Department of Computer Science

VALIDATION OF MEMORY ACCESSES THROUGH SYMBOLIC ANALYSES

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

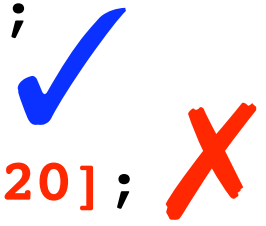
DCC
DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO

UFMG

The Goal of this Work

Our goal is to prove that some memory accesses are safe.

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



Keywords:

- Safety
- Performance

PPGCC
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO DCC-UFMG

The Contributions of This Work

- We have designed and implemented a technique to prove that some memory accesses are always safe.
  - Thus, they do not need to be guarded.
  - Abstract interpretation.

- Implemented in LLVM and tested in AddressSanitizer.
  - Eliminate 50% of the guards.
  - SPEC CPU 2006 17% faster.

- More effective than previous work.

Our Key Insight

- Symbolic analyses.
  - Symbolic range analysis (R).
  - Symbolic region analysis (W).

```
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) fits within W(p). Therefore, p[i] is always safe!

The Importance of Securing Memory

- "Unsafety" has one advantage: efficiency.
- But it makes room for bugs that are hard to find.
- Buffer overflow attacks are possible because arrays are not guarded against out-of-bounds accesses.

All these worms that plague the internet have much to thank that C/C++ allow for unsafe programs.

Sanitizing Memory Accesses in C

- There are tools that sanitize memory accesses in C
  - SAFE Code
  - Softbounds
  - AddressSanitizer
  - etc...
- These tools use different techniques
- But they all add an overhead on the programs that they secure.

Example: AddressSanitizer

- Shadow every memory allocated.
  - For each byte, we have a bit that says if that byte is allocated or not.
- Guard every array access.
  - For every array access, check if its shadow bit is valid.
- This approach slows down SPEC CPU 2006 by around 25%

Important: the techniques that we shall talk about work with tools other than AddressSanitizer, and languages other than C/C++

The GreenArrays Framework

- GreenArrays is a suite of static analyses that we use to prove that some memory accesses are safe:

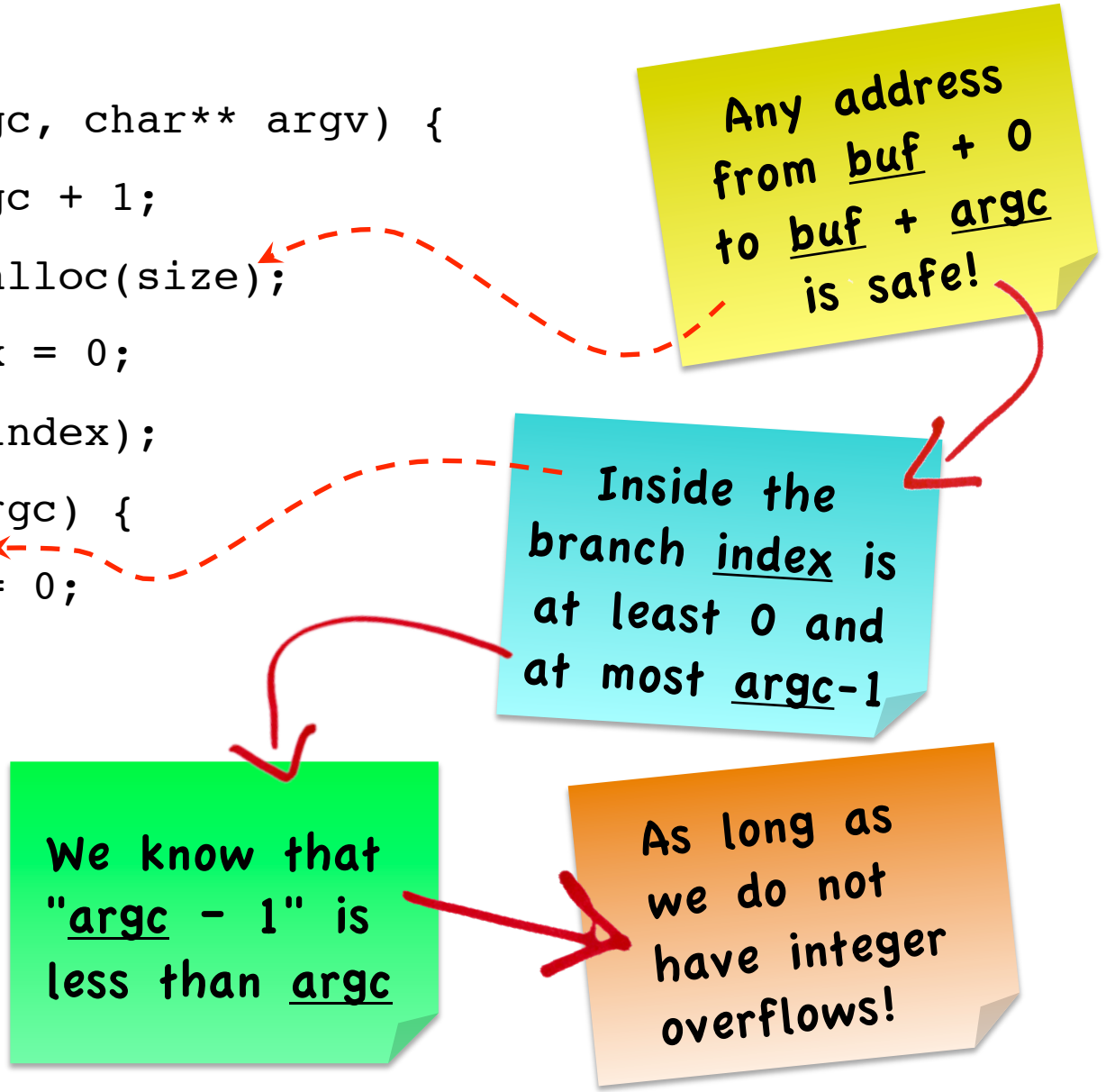
Symbolic Range Analysis:
finds the lower and upper values that variables can assume

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

Integer Overflow Analysis:
It lets us assume that "a < a + 1", for instance.

Overview

```
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. }
```

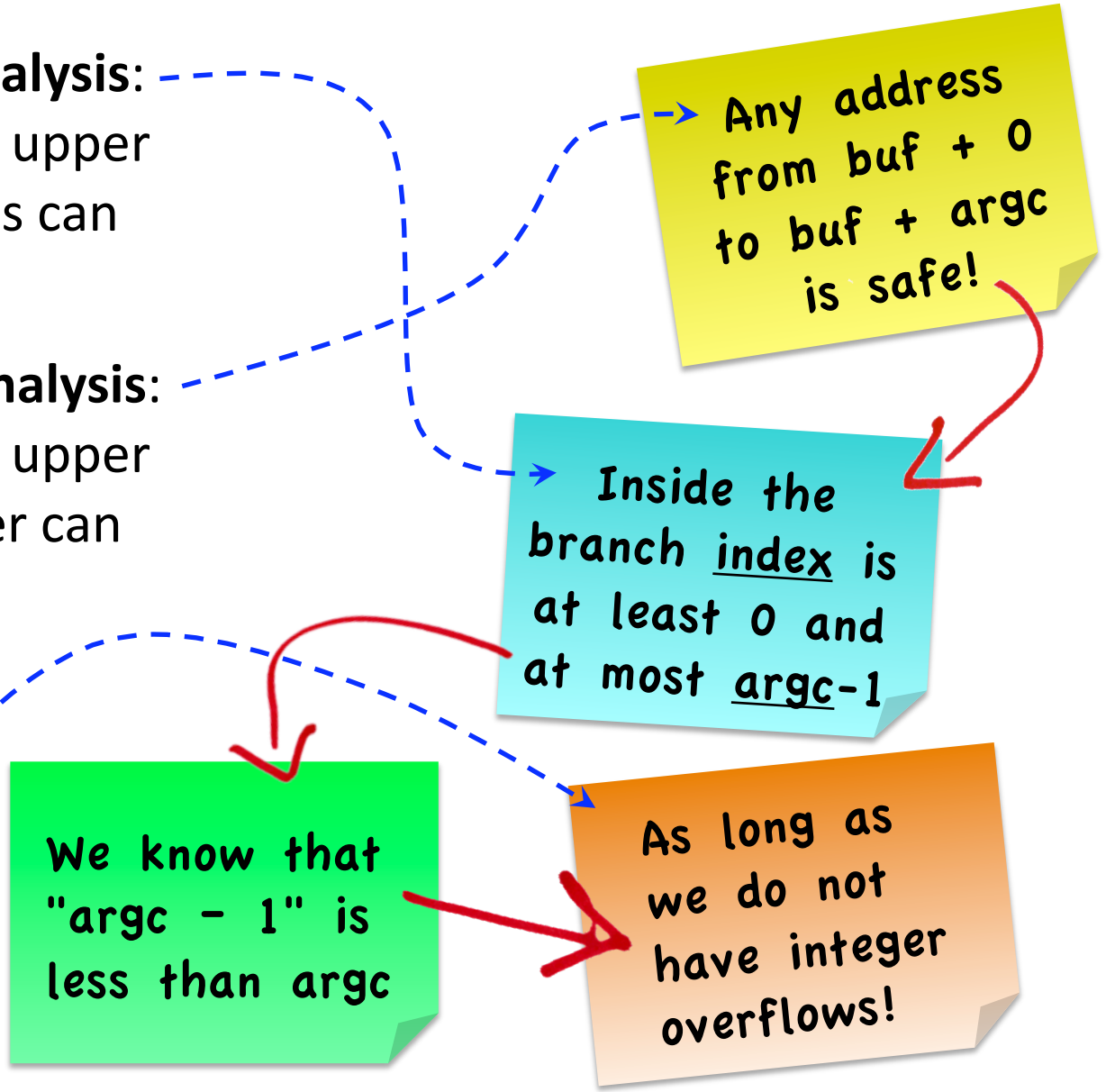


Overview

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

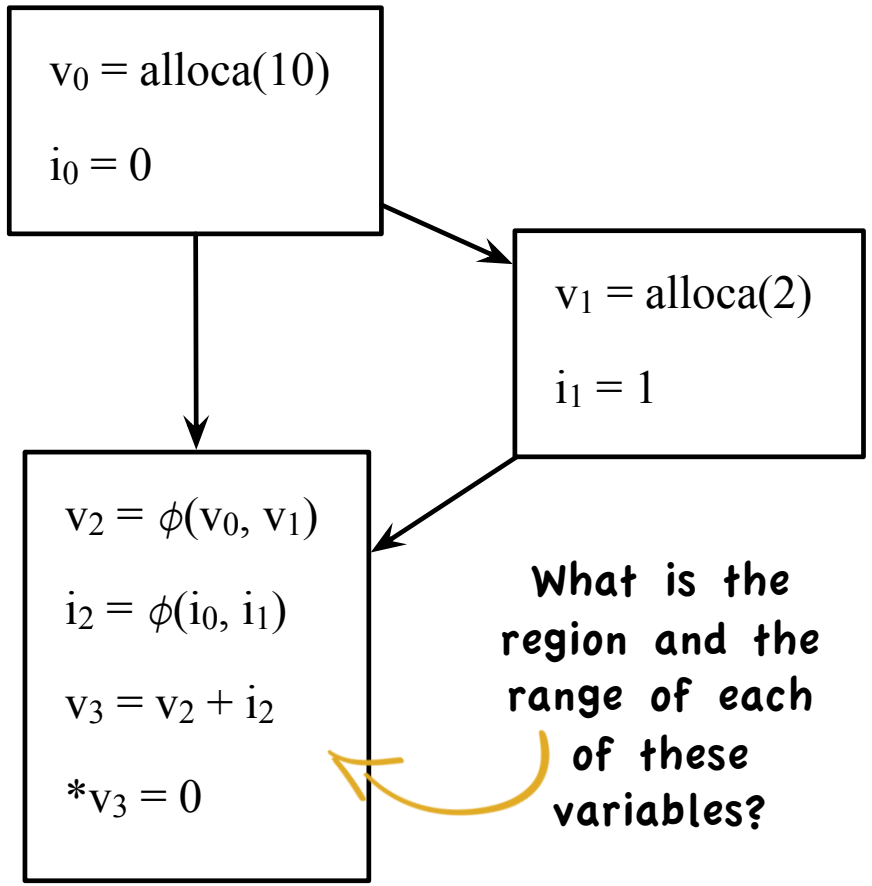
Symbolic Region Analysis: finds the lower and upper values that a pointer can address

Integer Overflow Analysis: Which arithmetic operations can overflow?



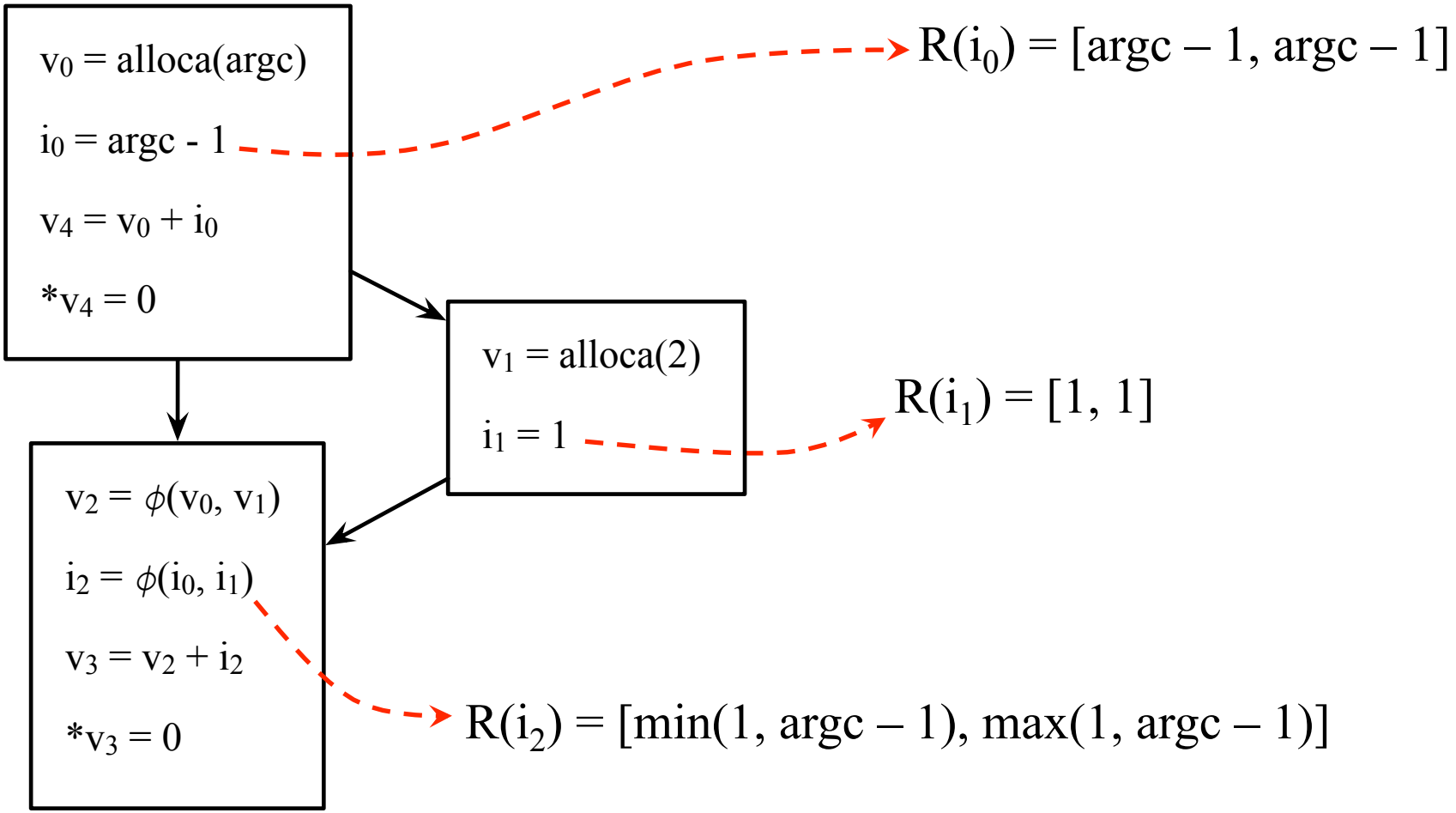
Abstract Interpretation of Programs

- We will be solving symbolic (range / region) analysis by abstract interpretation.
  - Assign an abstract state to each variable in the program.
  - Iterate the abstract interpreter until we reach a fixed point.
  - Apply widening to ensure termination.
  - Use narrowing to recover precision.



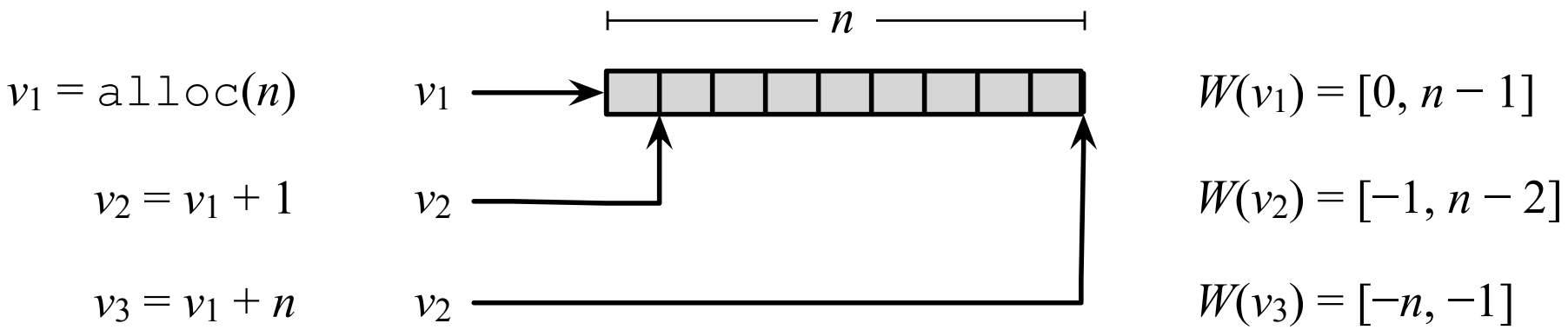
Symbolic Range Analysis

- Symbolic range analysis associates a symbolic range, e.g., $[\ell, \underline{u}]$ with each integer variable in a program.



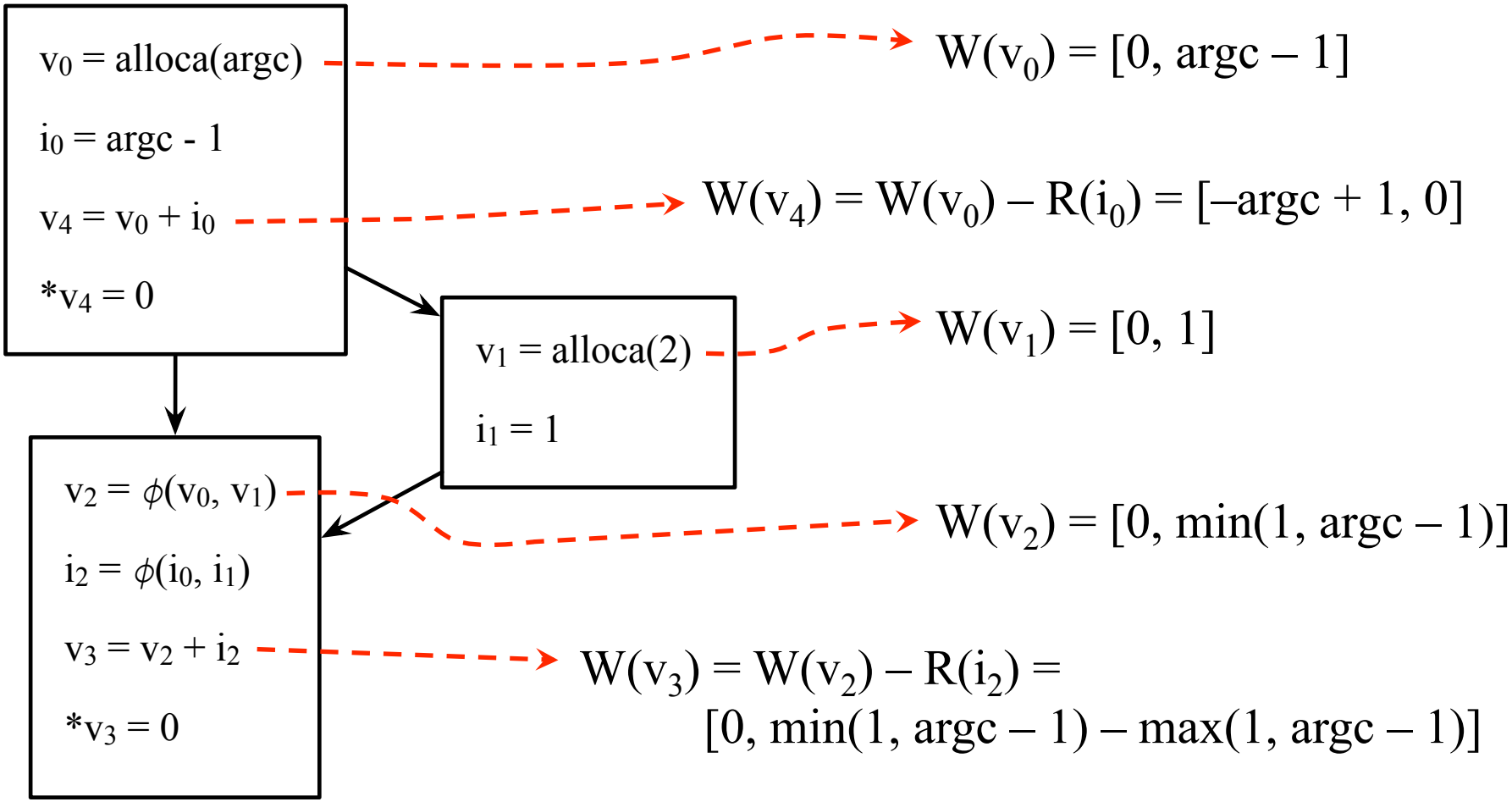
Symbolic Region Analysis

- Symbolic Region Analysis associates each pointer p with the range of valid offsets that can use that p as base.



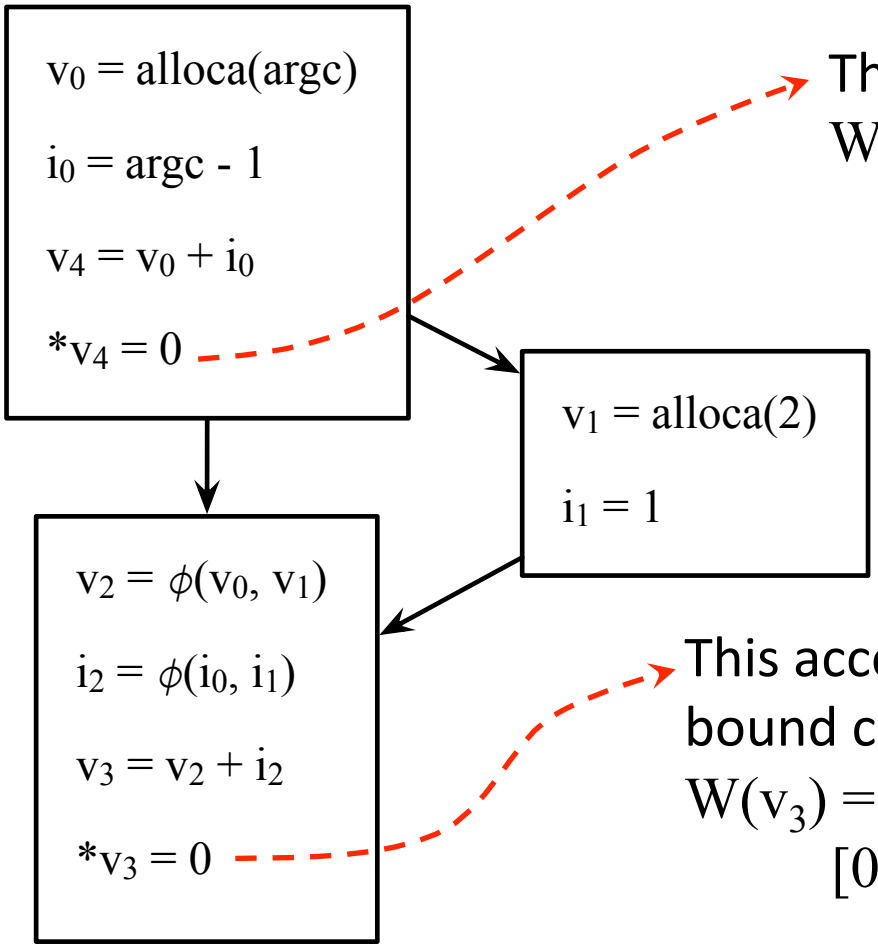
Symbolic Region Analysis

- Symbolic Region Analysis associates each pointer p with the range of valid offsets that can use that p as base.



Proving Safety

- **Theorem**: If $0 \in W(p)$, then *p is a safe memory access.

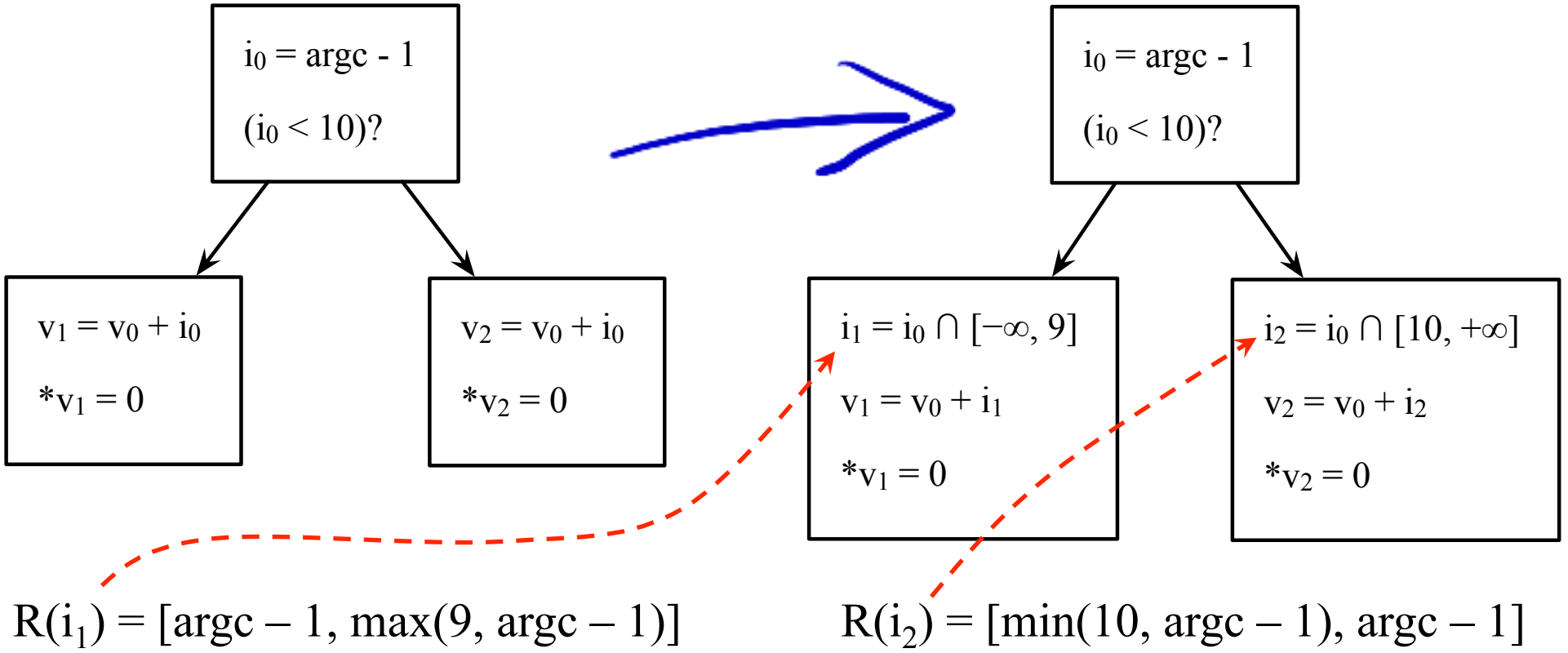


This access is **not** safe (if argc == 0):
$W(v_4) = W(v_0) - R(i_0) = [-argc + 1, 0]$

This access is **not** safe, because its upper bound can be negative:
$W(v_3) = W(v_2) - R(i_2) =$
$[0, \min(1, argc - 1) - \max(1, argc - 1)]$

"Hay que mejorar la *precisión*, pero sin perder la *speed* jamás"

- **Sparse analysis**: each variable name carries only one piece of abstract information. Fast implementation!
- To improve precision, we do **live range splitting**.



Wrapping Arithmetics

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

The Problem of Integer Overflows

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

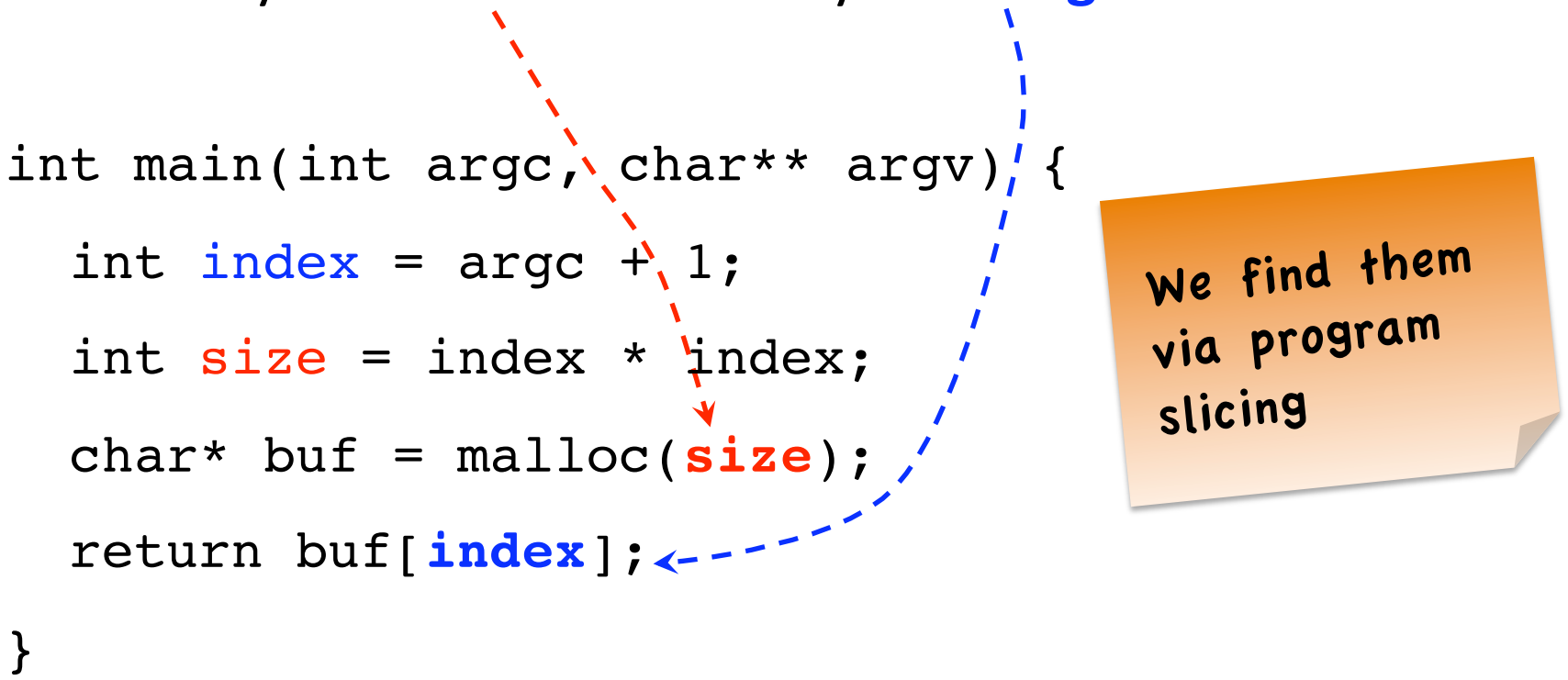
index * index may wrap around.

Do you know what malloc will return?

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

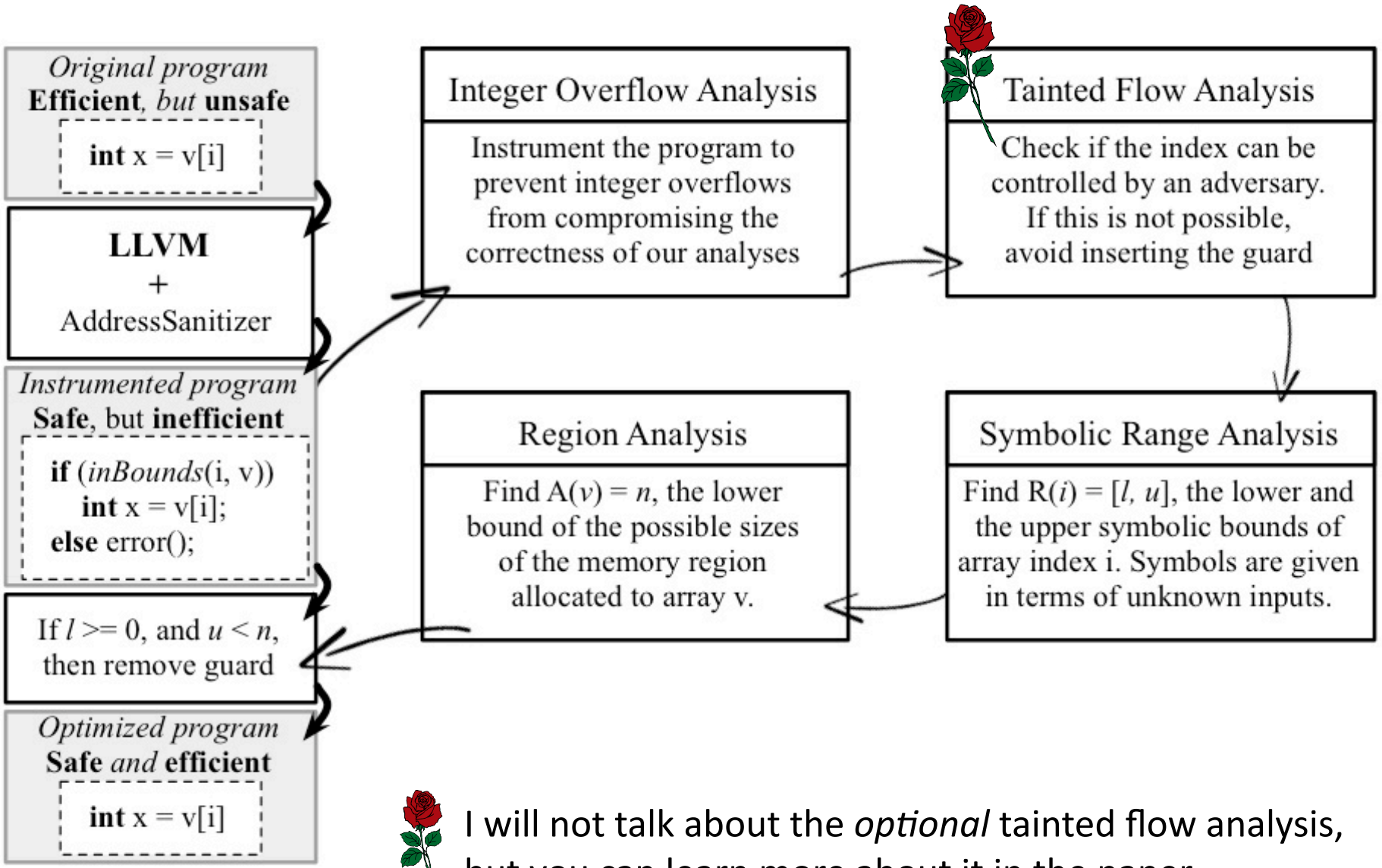
Guarding Against Integer Overflows

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



- We instrument the slice of instructions that influence memory to detect the occurrence of overflows.

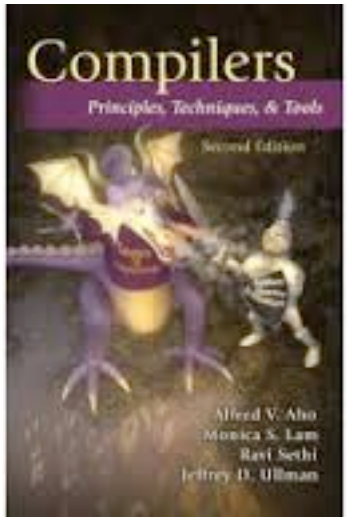
Putting it all Together: GreenArrays



I will not talk about the *optional* tainted flow analysis, but you can learn more about it in the paper.

A Bit of Perspective

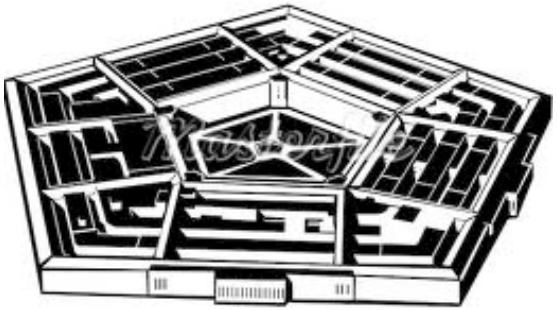
Chapter 9



Precision

- Non-Relational Analyses:
  - They associate variables with information that do not depend on other variables. **Examples**: the usual data flow analyses.
- Semi-Relational analyses:
  - They associate variables with information that may contain other variable names. **Examples**: pentagons, **symbolic range analysis, symbolic region analysis**.
- Relational analyses:
  - They associate sets of variables with information. **Examples**: octagons and polyhedrons.

Speed

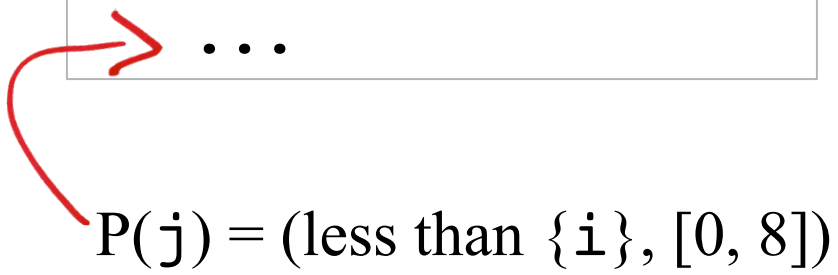


EXPERIMENTS

The Setup

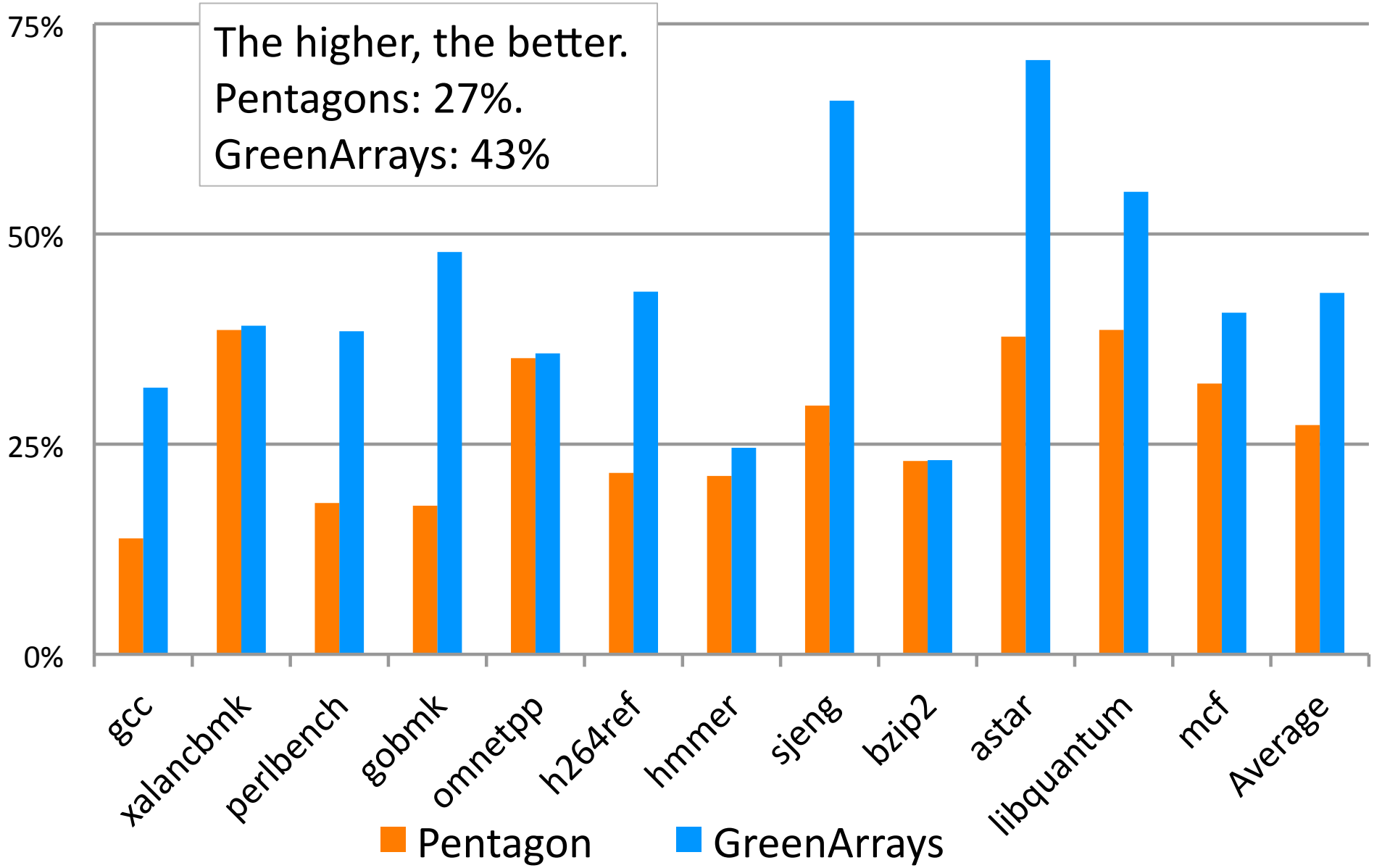
- **Implementation**: LLVM + AddressSanitizer
- **Benchmarks**: SPEC CPU 2006 + LLVM test suite
- **Machine**: Intel(R) Xeon(R) 2.00GHz,with 15,360KB of cache and 16GB or RAM
- **Baseline**: Pentagons
  - Abstract interpretation that combines "less-than" and "integer ranges".†

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

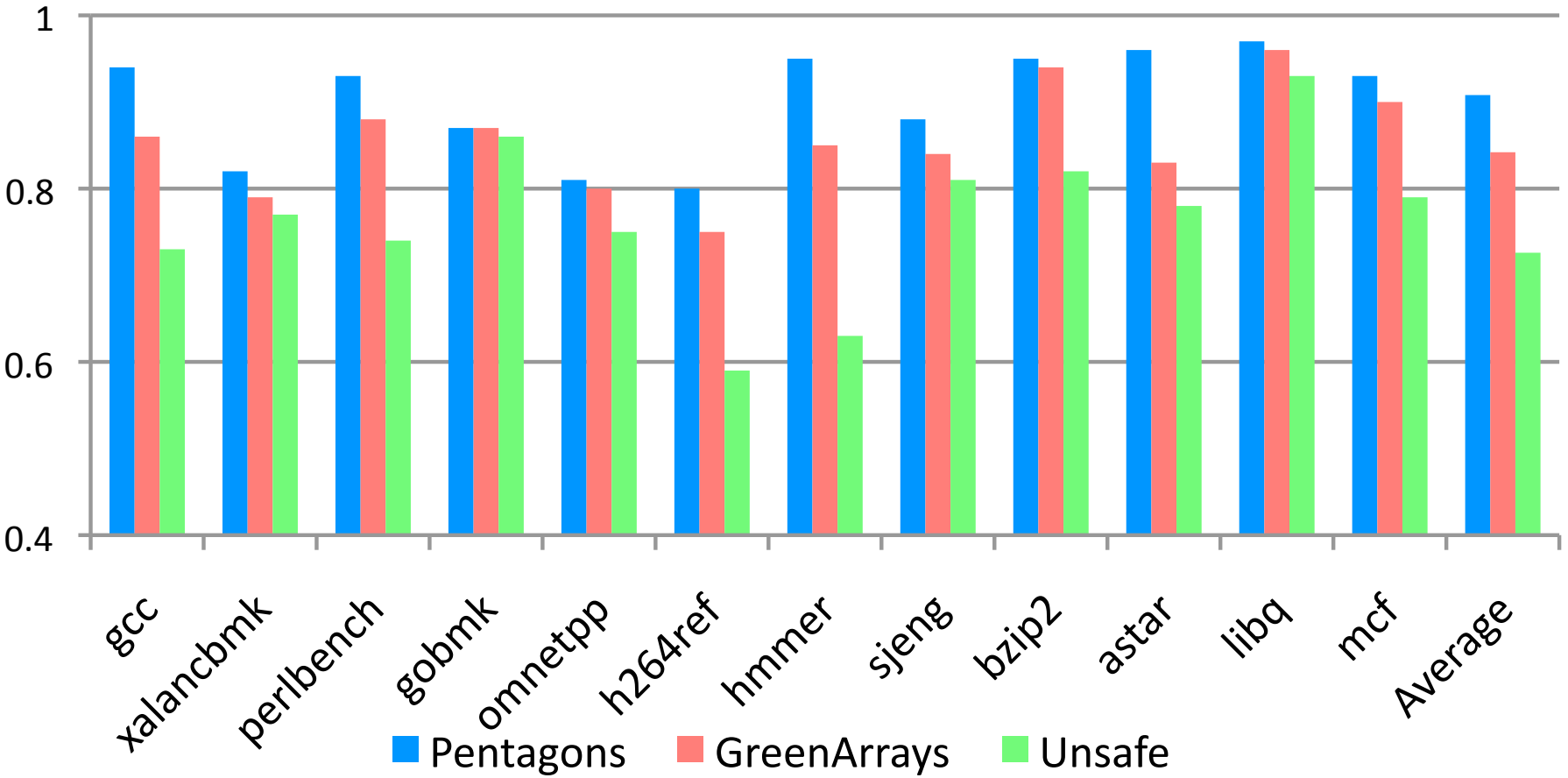


†: Pentagons: A weakly relational abstract domain for the efficient validation of array accesses, 2010, Science of Computer Programming

Percentage of Bound Checks Removed

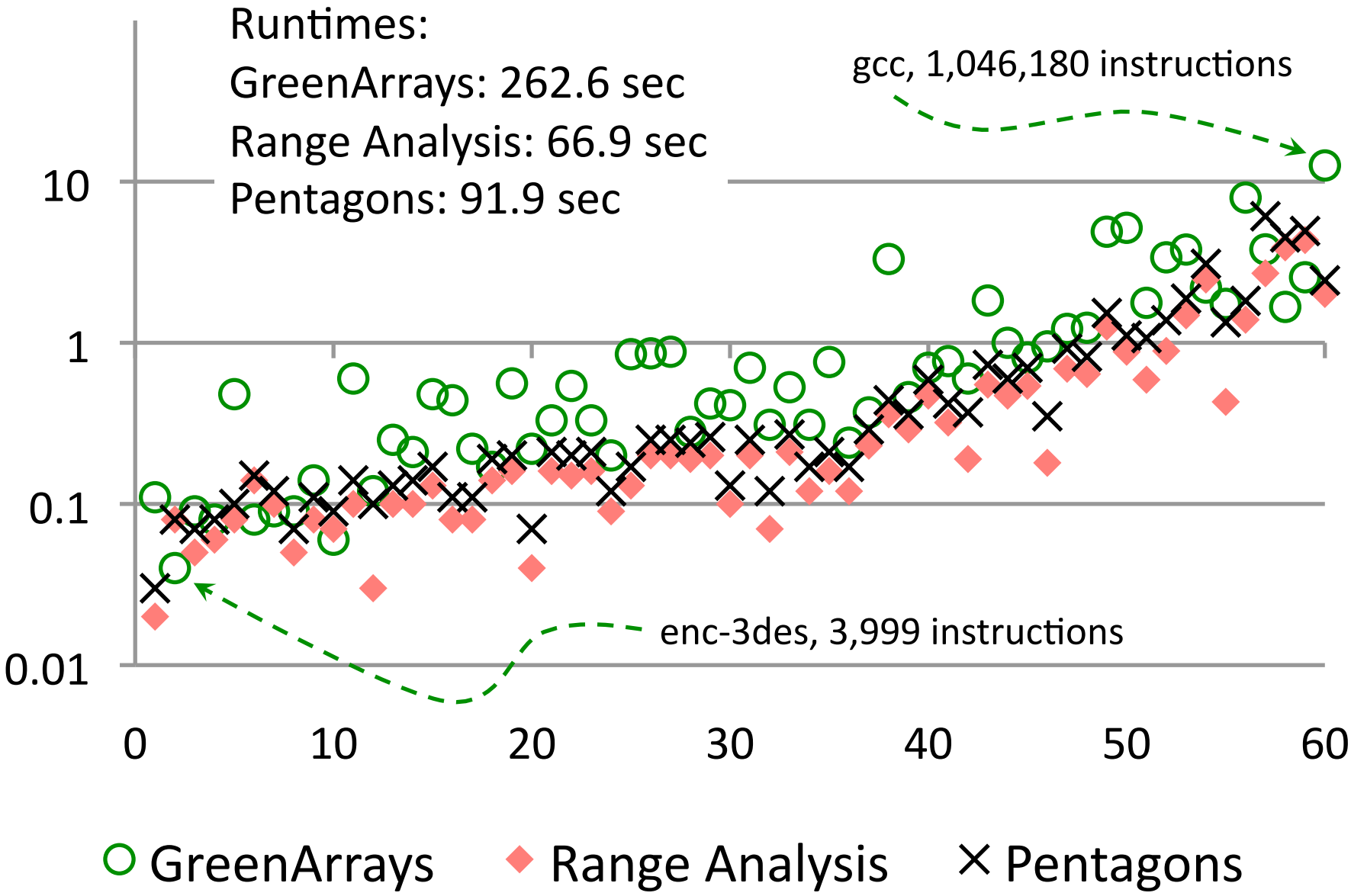


Runtime Improvement



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

Asymptotic Complexity



Final Thoughts

- We have presented two new static analyses that we can use to prove that memory accesses are safe.
- And they have many uses! Take a look:

All the code is publicly available at:
https://code.google.com/p/ecosoc/
Get in touch: fernando@dcc.ufmg.br

```
int main(int argc, char **argv) {
  int *a = malloc(16 * sizeof(int));
  for (int i = 0; i < 16; ++i)
    a[i] = i;
  for (int i = 0; i < 17; ++i)
    a[i] = i;

  int *b = malloc(argc * sizeof(int));
  for (int i = 0; i < argc; ++i)
    b[i] = i;
  for (int i = 0; i < argc + 1; ++i)
    b[i] = i;

  return 0;
}
```

```
int main(int argc, char **argv) {
  int *a = malloc(16 * sizeof(int));
  for (int i = 0; i < 16; ++i)
    a[i] = i;
  for (int i = 0; i < 17; ++i)
    /* bytes(&a[i]) = 0 * sizeof(int)   */
    /* index(&a[i]) = 0                 */
    /* WARNING: Possibly unsafe access. */
    /*          Off-by-one error.       */
    a[i] = i;

  int *b = malloc(argc * sizeof(int));
  for (int i = 0; i < argc; ++i)
    b[i] = i;
  for (int i = 0; i < argc + 1; ++i)
    /* bytes(&b[i]) = 0 * sizeof(int)   */
    /* index(&b[i]) = 0                 */
    /* WARNING: Possibly unsafe access. */
    /*          Off-by-one error.       */
    b[i] = i;

  return 0;
}
```