VALIDATION OF MEMORY ACCESES THROUGH SYMBOLIC ANALYSES

Henrique Nazare
Izabela Maffra
Willer Santos
Leonardo Oliveira
Fernando Quintão
Laure Gonnord
The Goal of this Work

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

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

Keywords:
- Safety
- Performance
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).

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

\[ W(p) = [0, \text{argc} - 1]. \]

\[ R(i) = [0, \text{argc} - 1] \]

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.

We want to reduce this overhead!
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.

What are the necessary assumptions to ensure that p[i] is a safe memory access?
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. }

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!
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?

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

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

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

As long as we do not have integer overflows!
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.

- What is the region and the range of each of these variables?

```
v_0 = alloca(10)
i_0 = 0

v_1 = alloca(2)
i_1 = 1

v_2 = \phi(v_0, v_1)
i_2 = \phi(i_0, i_1)
v_3 = v_2 + i_2
*v_3 = 0
```
Symbolic Range Analysis

- Symbolic range analysis associates a symbolic range, e.g., \([l, u]\) with each integer variable in a program.

\[
\begin{align*}
R(i_0) &= [\text{argc} - 1, \text{argc} - 1] \\
R(i_1) &= [1, 1] \\
R(i_2) &= [\min(1, \text{argc} - 1), \max(1, \text{argc} - 1)]
\end{align*}
\]
Symbolic Region Analysis

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

$v_1 = \text{alloc}(n) \quad v_2 = v_1 + 1 \quad v_3 = v_1 + n

W(v_1) = [0, n - 1] \quad W(v_2) = [-1, n - 2] \quad W(v_3) = [-n, -1]$
Symbolic Region Analysis

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

\[
W(v_0) = [0, argc - 1] \\
W(v_1) = [0, 1] \\
W(v_2) = [0, \min(1, argc - 1)] \\
W(v_3) = W(v_2) - R(i_2) = [0, \min(1, argc - 1) - \max(1, argc - 1)]
\]
Proving Safety

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

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

This access is **not** safe, because its upper bound can be negative:
$W(v_3) = W(v_2) - R(i_2) = [0, \min(1, \text{argc} - 1) - \max(1, \text{argc} - 1)]$
"Hay que mejorar la \textit{precisión}, pero sin perder la \textit{speed} jamás"

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

\begin{align*}
i_0 &= \text{argc} - 1 \\
&\quad (i_0 < 10)\

v_1 &= v_0 + i_0 \\
*_{v_1} &= 0
\end{align*}

\begin{align*}
i_0 &= \text{argc} - 1 \\
&\quad (i_0 < 10)\

v_2 &= v_0 + i_0 \\
*_{v_2} &= 0
\end{align*}

\begin{align*}
i_1 &= i_0 \cap [-\infty, 9] \\
v_1 &= v_0 + i_1 \\
*_{v_1} &= 0
\end{align*}

\begin{align*}
i_2 &= i_0 \cap [10, +\infty] \\
v_2 &= v_0 + i_2 \\
*_{v_2} &= 0
\end{align*}

\begin{align*}
R(i_1) &= [\text{argc} - 1, \text{max}(9, \text{argc} - 1)] \\
R(i_2) &= [\text{min}(10, \text{argc} - 1), \text{argc} - 1]
\end{align*}
Wrapping Arithmetics

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

Is this program always ok?
The Problem of Integer Overflows

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

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.

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

• We instrument the slice of instructions that influence memory to detect the occurrence of overflows.
Putting it all Together: GreenArrays

Original program
Efficient, but unsafe
\[ \text{int } x = v[i] \]

LLVM
+ AddressSanitizer

Instrumented program
Safe, but inefficient
\[
\begin{align*}
\text{if } (\text{inBounds}(i, v)) \\
\text{int } x = v[i]; \\
\text{else error();}
\end{align*}
\]
If \( l \geq 0 \), and \( u < n \), then remove guard

Optimized program
Safe and efficient
\[ \text{int } x = v[i] \]

Integer Overflow Analysis
Instrument the program to prevent integer overflows from compromising the correctness of our analyses

Tainted Flow Analysis
Check if the index can be controlled by an adversary. If this is not possible, avoid inserting the guard

Region Analysis
Find \( A(v) = n \), the lower bound of the possible sizes of the memory region allocated to array \( v \).

Symbolic Range Analysis
Find \( R(i) = [l, u] \), the lower and the upper symbolic bounds of array index \( i \). Symbols are given in terms of unknown inputs.

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

**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.
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".†

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

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

Percentage of Bound Checks Removed

The higher, the better.
Pentagons: 27%.
GreenArrays: 43%
The lower the bar, the faster. Time is normalized to AddressSanitizer without bound-check elimination. Average speedup: Pentagons = 9%. GreenArrays = 16%.
Asymptotic Complexity

Runtimes:
GreenArrays: 262.6 sec
Range Analysis: 66.9 sec
Pentagons: 91.9 sec

gcc, 1,046,180 instructions
enc-3des, 3,999 instructions

GreenArrays  Range Analysis  Pentagons
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