Static Memory Access Checking using Symbolic Constraint Generation

Edvin Török

advisor: Marius Minea

Institute e-Austria Timişoara

Computer Science Department "Politehnica" University of Timişoara

Introduction

- Source of memory errors in programs
  - null dereference
  - **bounds violations**
  - use after free
  - use of uninitialized values
- We need to find such memory errors in programs using tools
  - runtime checkers
  - **static analyzers** (Coverity, PolySpace, ...)
  - model checking

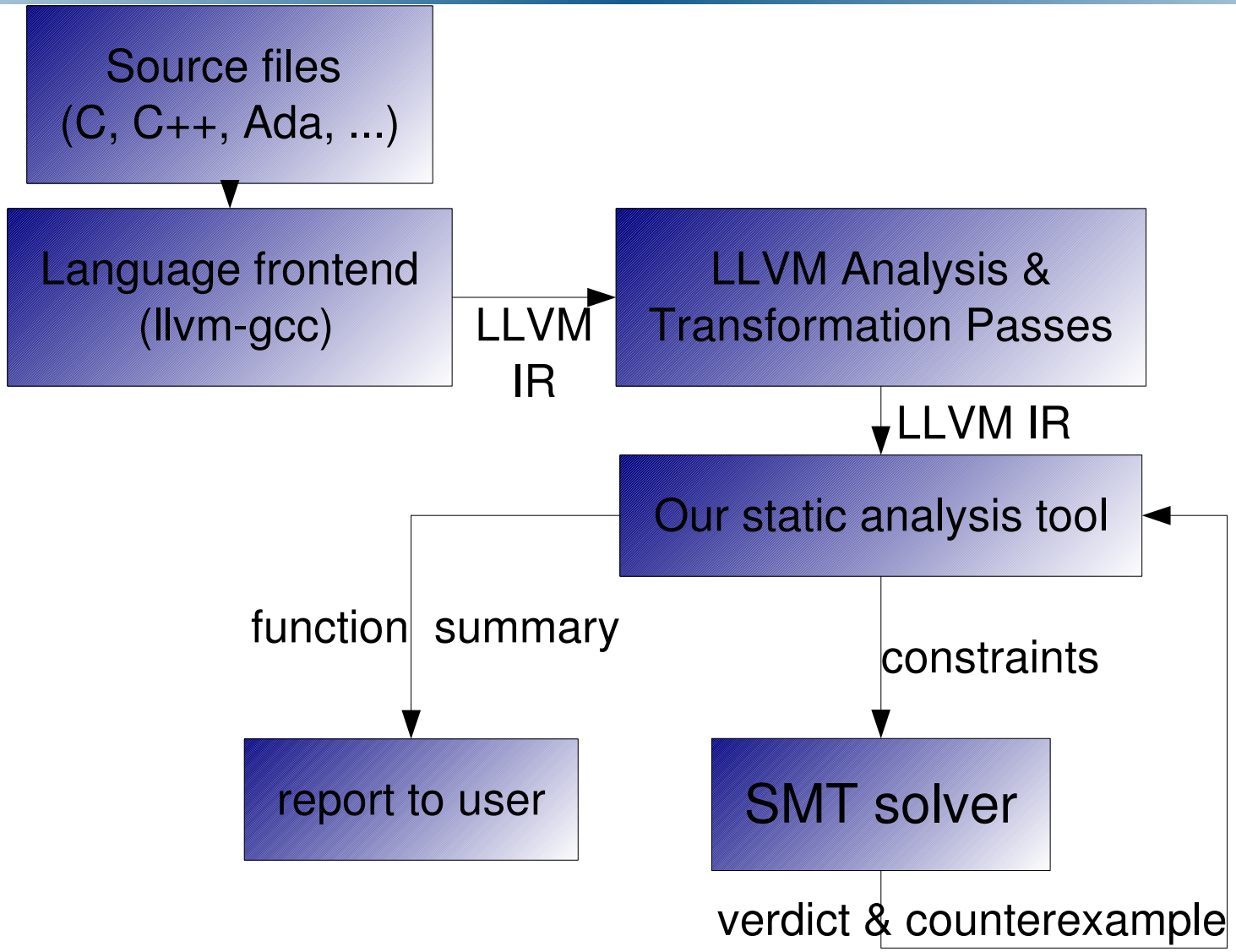
Overview

- A static analysis tool for detecting memory errors in programs (work in progress)
  - built using the LLVM compiler infrastructure
  - analyzes intermediate representation of source-code
  - generates path-sensitive constraints needed for absence of memory errors
  - constraints are generated in the smt-lib standard format facilitating a choice of solvers
  - uses an SMT solver to solve the constraints
  - reports issues to user

Kinds of bugs we are looking for

- Bounds violations
  - stack-allocated arrays
  - heap-allocated memory
- Overflows
  - integer overflow
  - logical errors: overflow of a structure field
- Soundness
  - Not yet, we need interprocedural analysis
  - Finding bugs, not building a full model

Block diagram



Why choose LLVM as a framework?

- Low Level Virtual Machine (llvm.org)
- Compiler Framework
  - fully functional compiler, able to compile millions of lines of source code fast
  - translates to a (partially) target-independent Intermediate Representation (bitcode)
  - all optimizations and analyses work on the IR
  - representation is low level, allowing us to track modular arithmetic accurately
  - not only for static analysis: native code generation, JIT, ...

What LLVM offers

- Alias Analysis
  - Basic, Andersen's, Steensgaard's, Data Structure Analysis
- Loop Analyses
  - Scalar Evolution
  - Loop Invariant Code Motion
- Single Static Assignment (SSA) form
- Control Flow Graphs (CFG)
- Data flow (via Use/Def chains)
- useful optimizations
  - Global Value Numbering
  - Memory to Register

How it works

```c
#include <stdlib.h>
int foo(unsigned n, unsigned m)
{
        char *x = malloc(n);
        if(x && m < n) {
                x[m] = 0;
        }
        return 0;
}
```

```
define i32 @foo(i32 %n, i32 %m) nounwind  {
entry:
        %tmp3 = malloc i8, i32 %n
        %tmp5 = icmp ne i8* %tmp3, null
        %tmp9 = icmp ult i32 %m, %n
        %tmp13 = and i1 %tmp5, %tmp9
        br i1 %tmp13, label %bb, label %UnifiedReturnBlock
bb:             ; preds = %entry
        %tmp1718 = zext i32 %m to i64
        %tmp19 = getelementptr i8* %tmp3, i64 %tmp1718
        store i8 0, i8* %tmp19, align 1
        ret i32 0
UnifiedReturnBlock:                               ; preds = %entry
        ret i32 0
}
```

- Analysis is demand-driven
  - Trigerred by a load or a store

Finding bounds

```
define i32 @foo(i32 %n, i32 %m) nounwind  {
entry:
        %tmp3 = malloc i8, i32 %n
        %tmp5 = icmp ne i8* %tmp3, null
        %tmp9 = icmp ult i32 %m, %n
        %tmp13 = and i1 %tmp5, %tmp9
        br i1 %tmp13, label %bb, label %UnifiedReturnBlock
bb:                ; preds = %entry
        %tmp1718 = zext i32 %m to i64
        %tmp19 = getelementptr i8* %tmp3, i64 %tmp1718
        store i8 0, i8* %tmp19, align 1
        ret i32 0
UnifiedReturnBlock:                ; preds = %entry
        ret i32 0
}
```

- using the reverse dataflow graph (use/def chains), we find the origin of values (allocation / constant)

Example: Generating symbolic constraints

```
#include <stdlib.h>
#include <stdint.h>
char *foo(uint32_t asize, uint32_t dsize)
{
        if(asize < 256) {
                char* x = malloc(asize + dsize);
                if(x) {
                        x[asize] = '\0';
                }
                return x;
        }
        return NULL;
}
```

Generating symbolic constraints

```
%tmp6 = add i32 %dsize, %asize
%tmp8 = malloc i8, i32 %tmp6
....
%tmp1516 = zext i32 %asize to i64
%tmp17 = getelementptr i8* %tmp8, i64 %tmp1516
store i8 0, i8* %tmp17, align 1
```

```
:assumption (= tmp6 (bvadd dsize asize ))
:formula (not (and (bvsge tmp6 bv0[32])(bvslt asize tmp6)))
```

- we build symbolic constraints / expressions using the dataflow graph
- we build a constraint that ensures the access is valid

Path Sensitivity

```
        %tmp2 = icmp ult i32 %asize, 256
        br i1 %tmp2, label %bb, label %UnifiedReturnBlock1
bb:                ; preds = %entry
        %tmp6 = add i32 %dsize, %asize
        %tmp8 = malloc i8, i32 %tmp6
```

```
:assumption (iff (= tmp2 bv1[1]) (bvult asize bv256[32]))
:assumption (= tmp2 bv1[1])
```

- we build constraints for the reachability of an instruction, by collecting condition variables that define the control flow
- filter out branch conditions that do not affect values used in the formula

SMT-LIB input to solver

```
(benchmark llvm2smt.smt
:source { Benchmarks generated by ... }
:status unknown
:difficulty{ unknown }
:category { unknown }
:logic QF_BV
:extrafuns ((asize BitVec[32]))
:extrafuns ((dsize BitVec[32]))
:extrafuns ((tmp2 BitVec[1]))
:extrafuns ((tmp6 BitVec[32]))
:extrafuns ((tmp8 BitVec[32]))
:extrafuns ((tmp10 BitVec[1]))
:assumption (= tmp6 (bvadd dsize asize ))
:assumption (iff (= tmp10 bv1[1]) (= tmp8 bv0[32]))
:assumption (= tmp10 bv1[1])
:assumption (iff (= tmp2 bv1[1]) (bvult asize bv256[32]))
:assumption (= tmp2 bv1[1])
:assumption (bvsge tmp6 bv0[32])
:formula (not (and (bvsge tmp6 bv0[32])(bvslt asize tmp6)))
)
```

- Uses standard SMT-LIB format as input to solvers
- Facilitates choice of solvers (yices, cvc3, ...)

Interpreting output from solver

- Solver output is sat / unsat / unknown, followed by a counterexample (for sat)
- a counterexample for the previous code is:
  - asize = 254
  - dsize = 4294967043
- constraint violations that do not depend on external factors (function parameters, other function calls, ...) are considered bugs and reported

Logical errors: structure field overflow

```
struct test {
        char buffer[10];
        uint16_t important_data;
};

void init(struct test* x)
{
        int i;
        x->buffer[10] = 0x55;
        for(i=0;i<=10;i++) {
                /* overwrites important_data field */
                x->buffer[i] = 0x55;
        }
}
```

```
$ llvm-gcc -g gep-struct-field-overflow.c -c -o gep-struct-field-overflow.bc -emit-llvm

$ opt -mem2reg -indvars -gvn -raiseallocs -bounds -disable-output foo.bc
```

(in gep-struct-field-overflow.c:12): warning: [struct-field-overflow] array bounds exceeded involving variable **x.buffer[10]** (in gep-struct-field-overflow.c:8) [I#1]

LLVM provided analyses

- Scalar Evolution
  - Based on chains of recurrences theory
    - Differences of order k
    - Newton's interpolation formula
- Loop Info
  - Trip-count
  - Invariants
  - canonical induction variable
  - canonical loop form

```
j = 3
for (i = 1; i <= 123; i+=5)
  {
    j = i + 7 + j;
  }
```

→ j = {3, +, 8, +, 5}(x)

Example of error found by the tool in Xorg

```
struct detailed_monitor_section {
  union {
    ....
    struct whitePoints wp[2];           /* 32 */
    ....
  } section;                            /* max: 80 */
};

static void
get_whitepoint_section(Uchar *c, struct whitePoints *wp)
{
       ...
       wp[2].white_x = WHITEX2;
       ...
}

get_whitepoint_section(c,det_mon[i].section.wp);
```

[I#1]: array bounds exceeded at (in interpret_edid.bc:xf86InterpretEDID)
warning: [struct-field-overflow]
Statement: (in interpret_edid.bc:xf86InterpretEDID)
Offending instruction:
%tmp1346 = getelementptr [2 x %struct.whitePoints]* %tmp12871288, i32 0, i64 2, i32 1
While investigating value:
store float %tmp13441345, float* %tmp1346, align 4

Results

- Tested on LLVM's testsuite
- Tested on X.org (1.4.99.901)
  - 572496 lines of code
  - Time to process
    - System: AMD Athlon(tm) 64 Processor 3200+, 2G RAM, Debian Linux
    - compile+generate bitcode (gcc and llvm)
      - real 23m25.634s, user 16m38.665s, sys 04m47.239s
    - **analyze (our tool)**
      - real 0m51.679s, user 0m29.615s, sys 0m9.199s
  - Has found one minor bug
  - Reports 160 (high) / 48058 (total) issues, need interprocedural analysis for more accuracy

Reporting of issues

- immediately reportable issues
- issues depending on function parameters
- report source:line where issue occurs
- show example values leading to constraint violation
- purpose
  - usable, practical

Limitations of our tool

- not yet inteprocedural
- analyzes integer operations only
- no assembler
- doesn't treat exception handlers yet
- Source line reporting is based on DWARF debug info in bitcode
  - Debug info only produced at *-O0*
  - Optimizations work best in absence of debug info
  - But this is about to change

Future work

- Interprocedurality
  - Use collected function summaries
- use LLVM's Value Range Propagation solver
- analyze C++ exception handlers
- use-after-free
- soundness

Related work

- Astree [Cousot et al '04] – used for checking avionics code
  - sound, floating point
  - doesn't analyze dynamically allocated memory
- Archer [Engler et al '03] - symbolic, path-sensitive analysis to detect memory access errors
  - Interprocedural, prunes impossible paths
- SAFEcode / Data Structure Analysis [Adve et al '06]
  - Combined approach: static and inserts runtime checks
  - Works on llvm bitcode
- Calysto [Babic et al '07]
  - Interprocedural
  - null pointer dereference detection

Questions?