Static Analysis using Abstract Interpretation

Maxime Arthaud

NASA Ames Research Center, California
Syllabus

1. Introduction
   - Software development
   - Safety properties
   - Abstract Interpretation

2. IKOS

3. Analyses

4. Miscellaneous

5. Conclusion
Software development

- Software represent more than half of the development cost of an aircraft
- Regulated by international standards (DO-178 rev. B/C)
Software development

- Software represent more than half of the development cost of an aircraft
- Regulated by international standards (DO-178 rev. B/C)
- Tests
  - Expensive because run on a special hardware
  - Can miss bugs
  - Slow
Software development

- Software represent more than half of the development cost of an aircraft
- Regulated by international standards (DO-178 rev. B/C)
- Tests
  - Expensive because run on a special hardware
  - Can miss bugs
  - Slow
- Solution: use static analysis
- NASA V&V program
Main objectives: no runtime errors
- buffer overflow
- null dereference
- division by zero
- integer overflow

Harder objectives:
- assertions (pre/post invariants)
- termination

Certified $\Rightarrow$ soundness is required

Abstract interpretation is a good candidate

Runtime errors can be security vulnerabilities!
Abstract Interpretation

- based on the concrete semantics of your program
- automatic formal proof
- sound approximation of reachable states
Abstract Interpretation

\[ x(t) \]

semantics(\(P\))

Possible trajectories
Forbiden zone

\textit{specification}(P)
Abstract Interpretation

Forbidden zone

\[ x(t) \]

\[ \text{Forbidden zone} \]

\[ \text{Possible trajectories} \]

\[ t \]

\[ \text{semantics}(P) \subseteq \text{specification}(P) \]
Tests

Using testing

Forbidden zone

Test of a few trajectories

Possible trajectories

Error !!!

Using testing
Abstract Interpretation

Abstraction of the trajectories

\[ \text{abstraction}(P) \]
Abstract Interpretation

Forbidden zone

Abstraction of the trajectories

\[ \text{abstraction}(P) \subseteq \text{specification}(P) \]
Abstract Interpretation

\[ \text{semantics}(P) \subseteq \text{abstraction}(P) \subseteq \text{specification}(P) \]
Thank you Pierre Loïc Garoche
The IKOS project

- Inference Kernel for Open Static Analyzers
- C++ library for abstract interpretation
- C/C++ static analyzer
- Target embedded systems
- Analyses:
  - Buffer overflow
  - Division by zero
  - Null dereference
  - Uninitialized variables
  - Prover
- https://ti.arc.nasa.gov/opensource/ikos/
**Toolchain**

- C/C++ code
- clang
- LLVM IR
- ikos-pp
- Optimized LLVM IR
- LLVM opt command + AR pass (-arbos)
- AR in s-expr

**IKOS**

- Abstract Domains
  - Interval
  - Constants
  - Discrete
  - Congruence
  - Interval + Congruence
  - Octagons
  - Difference Bounds Matrix
  - Pointer Analysis

- ikos-pp
  - Ikos-pp is an executable that embeds the LLVM opt command. It applies several LLVM built-in optimizations + our own optimization passes to produce an intermediate optimized LLVM IR. Using the optimized LLVM IR, we run LLVM opt command with -arbos option to translate the optimized LLVM IR to AR
  - ikos-pp does at least the following optimizations before translating to AR: -mem2reg, -loweratomic, -lowerswitch, and -instnamer

**AR Plugin Analyzers**

- BOA - buffer overflow analysis
- DBZ - Intra-procedural integer division-by-zero analysis
- UVA - Inter-procedural uninitialized variable + array analysis
- NullPtr - Inter-procedural null dereference pointer analysis

**ARBOS**

{AR parser, analysis plugin framework}

- Outputs reports to console
- IKOSView: desktop GUI that queries results stored in SQLite3 database
- Integrated into web services (such as continuous build + bug tracking systems)
  - SonarQube – using sonar_runner
  - CodeDX – import results in cppcheck XML format
  - SWAMP – used in cybersecurity

**Analysis results**
- Low Level Virtual Machine
- Compiler Infrastructure
- Generic assembly language
- Allow language independent optimization
• Low Level Virtual Machine
• Compiler Infrastructure
• Generic assembly language
• Allow language independent optimization

\[
\begin{array}{c}
\text{C} \\
\text{C++} \\
\text{Fortran} \\
\text{Ada} \\
\text{x86} \\
\text{PowerPC} \\
\text{ARM} \\
\text{AR}
\end{array}
\]

\text{llvm bitcode}
$ cat test.c

#include <stdio.h>

int main(int argc, char** argv) {
    int a[10];
    int i;
    for (i = 0; i < 10; i++) {
        a[i] = i;
    }
    printf("%d\n", a[i - 1]);
    printf("%d\n", a[0]);
    return 0;
}
$ clang -c -emit-llvm -O1 -o test.bc test.c
$ opt -S test.bc

define i32 @main(i32, i8** nocapture readnone) local_unnamed_addr #0 {
  %3 = alloca [10 x i32], align 16
  %4 = bitcast [10 x i32]* %3 to i8*
  call void @llvm.lifetime.start(i64 40, i8* %4) #3
  br label %5

; <label>:5: ; preds = %5, %2
  %6 = phi i64 [ 0, %2 ], [ %9, %5 ]
  %7 = getelementptr inbounds [10 x i32], [10 x i32]* %3, i64 0, i64 %6
  %8 = trunc i64 %6 to i32
  store i32 %8, i32* %7, align 4
  %9 = add nuw nsw i64 %6, 1
  %10 = icmp eq i64 %9, 10
  br i1 %10, label %11, label %5

; <label>:11: ; preds = %5
  %12 = getelementptr inbounds [10 x i32], [10 x i32]* %3, i64 0, i64 9
  %13 = load i32, i32* %12, align 4
  %14 = tail call i32 (i32, i8*, ...) @__printf_chk(i32 1,
    i8* getelementptr inbounds ([4 x i8], [4 x i8]* @.str, i64 0, i64 0), i32 %13) #3
  %15 = getelementptr inbounds [10 x i32], [10 x i32]* %3, i64 0, i64 0
  %16 = load i32, i32* %15, align 16
  %17 = tail call i32 (i32, i8*, ...) @__printf_chk(i32 1,
    i8* getelementptr inbounds ([4 x i8], [4 x i8]* @.str, i64 0, i64 0), i32 %16) #3
  call void @llvm.lifetime.end(i64 40, i8* nonnull %4) #3
  ret i32 0
}
%2:
%3 = alloca [10 x i32], align 16
%4 = bitcast [10 x i32]* %3 to i8*
call void @llvm.lifetime.start(i64 40, i8* %4) #3
br label %5

%5:
%6 = phi i64 [ 0, %2 ], [ %9, %5 ]
%7 = getelementptr inbounds [10 x i32], [10 x i32]* %3, i64 0, i64 %6
%8 = trunc i64 %6 to i32
store i32 %8, i32* %7, align 4, !tbaa !3
%9 = add nuw nsw i64 %6, 1
%10 = icmp eq i64 %9, 10
br i1 %10, label %11, label %5

%11:
%12 = getelementptr inbounds [10 x i32], [10 x i32]* %3, i64 0, i64 9
%13 = load i32, i32* %12, align 4, !tbaa !3
%14 = tail call i32 (i32, i8*, ...) @__printf_chk(i32 1, i8* getelementptr ...
... inbounds ([4 x i8], [4 x i8]* @.str, i64 0, i64 0), i32 %13) #3
%15 = getelementptr inbounds [10 x i32], [10 x i32]* %3, i64 0, i64 0
%16 = load i32, i32* %15, align 16, !tbaa !3
%17 = tail call i32 (i32, i8*, ...) @__printf_chk(i32 1, i8* getelementptr ...
... inbounds ([4 x i8], [4 x i8]* @.str, i64 0, i64 0), i32 %16) #3
call void @llvm.lifetime.end(i64 40, i8* nonnull %4) #3
ret i32 0
IKOS pre-processor

Run llvm optimization passes:
- `mem2reg`: SSA Form
- `globaldce`: Dead Code Elimination
- `globalopt`: Global Variable Optimizer
- `simplifycfg`: Control Flow Graph Optimizer
- `scalarrepl`: Scalar Replacement of Aggregates
- `sccp`: Sparse Conditional Constant Propagation
- `loop-simplify`: Canonical Form for Loops
- `lcssa`: Loop Closed SSA Form
- `loop-deletion`: Dead Loop Elimination
- `lowerinvoke`: Lower Invoke Instructions
- `lowerswitch`: Lower Switch Instructions

Run home made llvm passes:
- Lower Global Variable Initialization
- Lower Constant Expressions
- Lower Select Instructions
- Name Values
Abstract Representation

Major differences with llvm:
- Branching instructions are translated into assertions
- Memory instructions are byte oriented
- Some instructions are removed

Translation from llvm to AR using a llvm pass

Text representation using s-expressions
($function
($name ($main)) ($ty (!8))
($params ($p ($name ($main.arg_1)) ($ty (!9))) ($p ($name ($main.arg_2)) ($ty (!10))))
($local_vars ($local_var ($var ($name ($main._1)) ($ty (!11))))
($code
($entry ($bb_1)) ($exit ($bb_5)) ($unreachable) ($ehresume)
($basicblocks
($basicblock ($name ($bb_1))
($instructions
($allocate ($dest ($cst ($localvariableref ($name ($main._1)) ($ty (!11)))) ($alloca_ty (!12)) ($array_size ($cst ($constantint ($val (#1)) ($ty (!9)))) ($debug ($srcloc ($line (#-1)) ($col (#-1)) ($file (!2)))))
)
($basicblock ($name ($*in_bb_1_to_bb_2_phi))
($instructions
($assign ($lhs ($var ($name ($main.i.0)) ($ty (!9)))) ($rhs ($cst ($constantint ($val (#0)) ($ty (!9)))) ($debug ($srcloc ($line (#6)) ($col (#10)) ($file (!13)))))
)
)[...]
)
($trans
($edge ($bb_1) ($*in_bb_1_to_bb_2_phi))
($edge ($*in_bb_1_to_bb_2_phi) ($bb_2))
($edge ($bb_2) ($*out_bb_2_to_bb_3_icmp_true))
($edge ($bb_2) ($*out_bb_2_to_bb_5_icmp_false))
($edge ($*in_bb_4_to_bb_2_phi) ($bb_2))
)[...]
bb_4:
main._14 = add(main.i.0, 1)

*in_bb_4_to_bb_2_phi:
main.i.0 = main._14

bb_2:

*out_bb_2_to_bb_3_icmp_true:
main.i.0 slt 10
main.8 = -1

*out_bb_2_to_bb_5_icmp_false:
main.i.0 sge 10
main.8 = 0

bb_3:
main._10 = sext main.i.0
_v:7 = mul(4, main._10)
main._11 = ptr_shift(main._1, _v:7)
memory[main._11] = main.i.0

bb_5:
main._17 = sub(main.i.0, 1)
main._18 = sext main._17
_v:10 = mul(4, main._18)
main._19 = ptr_shift(main._1, _v:10)
main._20 = memory[main._19]
main._21 = ptr_shift(.str, 0)
main._22 = call printf(main._21, main._20)
main._24 = memory[main._1]
main._25 = ptr_shift(.str, 0)
main._26 = call printf(main._25, main._24)
return 0

bb_1:
main._1 = allocate(1, [10 x i32])

*in_bb_1_to_bb_2_phi:
main.i.0 = 0

*in_bb_4_to_bb_2_phi:
main.i.0 = main._14
Load an Abstract Representation file (.ar) and apply passes

Similar to llvm opt command

IKOS passes:
- ps-opt: Optimize pointer shift statements
- branching-opt: Optimize the Control Flow Graph
- inline-init-gv: Inline initialization of global variables in main
- unify-exit-nodes: Unify exit nodes
- analyzer: Analyzer pass
bb 4:
main._14 = add(main.i.0, 1)
main.i.0 = main._14

*out_bb_2_to_bb_3_icmp_true:
main.i.0 slt 10
main._8 = -1

bb 1:
main._1 = allocate(1, [10 x i32])
main.i.0 = 0
main.i.0 slt 10
main._8 = -1

*out_bb_2_to_bb_5_icmp_false:
main.i.0 sge 10
main._8 = 0

bb 3:
main.10 = sext main.i.0
__v:7 = mul(4, main.10)
main.11 = ptr_shift(main.1, __v:7)
memory[main.11] = main.i.0

bb 5:
main._17 = sub(main.i.0, 1)
main._18 = sext main._17
__v:10 = mul(4, main._18)
main._19 = ptr_shift(main._1, __v:10)
main._20 = memory[main._19]
main._21 = ptr_shift(.str, 0)
main._22 = call printf(main._21, main._20)
main._24 = memory[main._1]
main._25 = ptr_shift(.str, 0)
main._26 = call printf(main._25, main._24)
return 0
Analyzer

- Liveness analysis
- Pointer analysis
- Memory analysis combining:
  - Numerical analysis
  - Pointer analysis
  - Uninitialized variable analysis
  - Null pointer analysis
- Checkers:
  - buffer overflow
  - division by zero
  - null dereference
  - uninitialized variables
  - assertion prover
- Store results in a SQLite database
- The toolchain is launched via a python script
- Generate reports in different formats:
  - Console (gcc style)
  - JSON
  - XML
  - etc.
- Output database reusable (using *ikos-render*)
Toolchain

**C/C++ code**

**clang**

**LLVM IR**

**ikos-pp**

**Optimized LLVM IR**

**LLVM opt command + AR pass (-arbos)**

**AR in s-expr**

**IKOS**

- Abstract Domains
  - Interval
  - Constants
  - Discrete
  - Congruence
  - Interval + Congruence
  - Octagons
  - Difference Bounds Matrix
  - Pointer Analysis

- ikos-pp
  - Ikos-pp is an executable that embeds the LLVM opt command. It applies several LLVM built-in optimizations + our own optimization passes to produce an intermediate optimized LLVM IR. Using the optimized LLVM IR, we run LLVM opt command with `-arbos` option to translate the optimized LLVM IR to AR.
  - ikos-pp does at least the following optimizations before translating to AR: `-mem2reg`, `-loweratomic`, `-lowerswitch`, and `-instnamer`

- AR Plugin Analyzers
  - BOA - buffer overflow analysis
  - DBZ - Intra-procedural integer division-by-zero analysis
  - UVA - Inter-procedural uninitialized variable + array analysis
  - NullPtr - Inter-procedural null dereference pointer analysis

**ARBOS**

({AR parser, analysis plugin framework})

**Analysis results**

- Outputs reports to console
- IKOSView: desktop GUI that queries results stored in SQLite3 database
- Integrated into web services (such as continuous build + bug tracking systems)
  - SonarQube – using sonar_runner
  - CodeDX – import results in cppcheck XML format
  - SWAMP – used in cybersecurity
Demo.

Demo.
Aeroquad - The Open Source Quadcopter

- **Code size:**
  - lines of code: 167k
  - bitcode instructions: 4634

- **Time stats:**
  - arbos: 1 min 51.888 sec
  - ikos-pp: 0.126 sec
  - llvm-to-ar: 0.898 sec

- **Summary:**
  - number of checks: 2908
  - number of unreachable checks: 46 (1.6%)
  - number of safe checks: 2688 (92.4%)
  - number of definite unsafe checks: 0
  - number of warnings: 174 (5.9%)
Aeroquad - The Open Source Quadcopter

- **Writes at specific addresses:**
  
  ```
  *(0x42) = x;
  ```

- **False positives on loops with casts:**
  
  ```
  for (byte axis = 0; axis < 3; axis++) {
      accelSample[axis] = 0;
  }
  ```

- **Tricky array indexing:**
  
  ```
  static byte receiverPin[6] =
      {2, 5, 6, 4, 7, 8};
  pinData[receiverPin[channel]].edge = FALLING_EDGE;
  ```
Paparazzi - Autopilot System for UAV

- **Code size**:
  - lines of code: 23k
  - bitcode instructions: 4436

- **Time stats**:
  - arbos: 1 min 2.930 sec
  - ikos-pp: 0.132 sec
  - llvm-to-ar: 1.111 sec

- **Summary**:
  - number of checks: 2372
  - number of unreachable checks: 352 (14.8%)
  - number of safe checks: 2020 (85.2%)
  - number of definite unsafe checks: 0
  - number of warnings: 0
Results

GEN2

- Code size:
  - lines of code: 13k
  - bitcode instructions: 5340

- Time stats:
  - arbos: 2 min 16.161 sec
  - ikos-pp: 0.199 sec
  - llvm-to-ar: 1.358 sec

- Summary:
  - number of checks: 3121
  - number of unreachable checks: 0
  - number of safe checks: 3028 (97.1%)
  - number of definite unsafe checks: 0
  - number of warnings: 93 (2.9%)
MNAV

- Code size:
  - lines of code: 159k
  - bitcode instructions: 2145

- Time stats:
  - arbos: 12.950 sec
  - ikos-pp: 0.056 sec
  - llvm-to-ar: 0.468 sec

- Summary:
  - number of checks: 430
  - number of unreachable checks: 17 (3.9%)
  - number of safe checks: 330 (76.7%)
  - number of definite unsafe checks: 0
  - number of warnings: 83 (19.3%)
Results

CASS

- **Time stats:**
  - arbos: 1 day 2 hour 17.463 sec
  - ikos-pp: 13.234 sec
  - llvm-to-ar: 24.431 sec

- **Summary:**
  - number of checks: 254452
  - number of unreachable checks: 33300 (13.0%)
  - number of safe checks: 172521 (67.8%)
  - number of definite unsafe checks: 0
  - number of warnings: 48631 (19.1%)
Results

FLTz - flight simulator with OpenGL displays

- Code size:
  - lines of code: 91k
  - bitcode instructions: 14501

- Time stats:
  - arbos: 5 day 9 hour 27 min 41.459 sec
  - ikos-pp: 25.211 sec
  - llvm-to-ar: 1 min 2.661 sec

- Summary:
  - number of checks: 1302470
  - number of unreachable checks: 72409 (5.5%)
  - number of safe checks: 153312 (11.7%)
  - number of definite unsafe checks: 19 (0.001%)
  - number of warnings: 1076730 (82.6%)
Syllabus

1. Introduction

2. IKOS

3. Analyses
   - Liveness analysis
   - Pointer analysis
   - Memory analysis
   - Property checking

4. Miscellaneous

5. Conclusion
Liveness analysis

- Mark *live* and *dead* variables after each basic block
- Dataflow analysis
- Used to clean up variables in the abstract domain
- Problem for relationnal domains
Liveness analysis - Algorithm

- **Kill - Gen algorithm**
- $GEN[b]$: set of variables used in $b$ before any assignment
- $KILL[b]$: set of variables that are assigned in $b$
Liveness analysis - Algorithm

- Kill - Gen algorithm
- $GEN[b]$ : set of variables used in $b$ before any assignment
- $KILL[b]$ : set of variables that are assigned in $b$

- $GEN[stmt : y \leftarrow f(x_1, \cdots, x_n)] = \{x_1, \ldots, x_n\}$
- $KILL[stmt : y \leftarrow f(x_1, \cdots, x_n)] = \{y\}$
Liveness analysis - Algorithm

- **Kill - Gen algorithm**
  - \( GEN[b] \) : set of variables used in \( b \) before any assignment
  - \( KILL[b] \) : set of variables that are assigned in \( b \)

- \( GEN[stmt : y ← f(x_1, \cdots, x_n)] = \{x_1, \ldots, x_n\} \)
- \( KILL[stmt : y ← f(x_1, \cdots, x_n)] = \{y\} \)

- \( LIVE_{in}[b] = GEN[b] \cup (LIVE_{out}[b] − KILL[b]) \)
- \( LIVE_{out}[b] = \bigcup_{p \in succ[b]} LIVE_{in}[p] \)
- \( LIVE_{out}[final] = \emptyset \)
Liveness analysis - Example

bb 4:
main. 14 = add(main.i.0, 1)
main.i.0 = main. 14

*out_bb_2_to_bb_3_icmp_true:
main.i.0 slt 10
main._8 = -1

bb 3:
main. 10 = sext main.i.0
_v:7 = mul(4, main. 10)
main. 11 = ptr_shift(main. 1, _v:7)
memory[main. 11] = main.i.0

bb 5:
return 0

*out_bb_2_to_bb_5_icmp_false:
main.i.0 sge 10
main._8 = 0

bb 1:
main. 1 = allocate(1, [10 x i32])
main.i.0 = 0
main.i.0 slt 10
main._8 = -1

Maxime Arthaud
Pointer analysis: What memory locations can a pointer expression refer to?

Alias analysis: Are two pointers referring to the same locations?

Intraprocedural vs Interprocedural

Flow sensitive vs Flow insensitive

Context sensitive vs Context insensitive
How to model memory locations?

- Global variables: use symbolic names (e.g., `g`)
- Local variables: use symbolic names (e.g., `main.x`)
- Dynamically allocated memory: use symbolic names?
  - Problem: potentially unbounded locations (think about a loop)
  - Solution: use symbolic names with an instruction counter (e.g., `blk(l, λ)`)
Andersen’s pointer analysis

For each pointer \( p \), we call \( T_p \) the set of memory locations pointed by \( p \)

Goal : find \( T_p \) for each pointer \( p \)

Idea : view pointer assignments as subset constraints

Complexity : \( O(n^3) \), worst case \( O(n^4) \)
Andersen’s pointer analysis

For each pointer \( p \), we call \( T_p \) the set of memory locations pointed by \( p \)

Goal : find \( T_p \) for each pointer \( p \)

Idea : view pointer assignments as subset constraints

Complexity : \( O(n^3) \), worst case \( O(n^4) \)

\[
\begin{align*}
p = \& x & \Leftrightarrow & T_p \supseteq \{ x \} \\
p = q + o & \Leftrightarrow & T_p \supseteq T_q \\
p = *q & \Leftrightarrow & T_p \supseteq *T_q \Rightarrow \forall x \in T_q, T_p \supseteq O(x) \\
*p = q & \Leftrightarrow & *T_p \supseteq T_q \Rightarrow \forall x \in T_p, O(x) \supseteq T_q
\end{align*}
\]
- Andersen’s pointer analysis
- For each pointer $p$, we call $T_p$ the set of memory locations pointed by $p$
- Goal: find $T_p$ for each pointer $p$
- Idea: view pointer assignments as subset constraints
- Complexity: $O(n^3)$, worst case $O(n^4)$

- $p = \&x \iff T_p \supseteq \{x\}$
- $p = q + o \iff T_p \supseteq T_q$
- $p = *q \iff T_p \supseteq *T_q \iff \forall x \in T_q, T_p \supseteq O(x)$
- $*p = q \iff *T_p \supseteq T_q \iff \forall x \in T_p, O(x) \supseteq T_q$

- How to solve the constraints system? A fix point, of course!
Example:

- \( p = &a \)
- \( q = &b \)
- \( *p = q \)
- \( r = &c \)
- \( s = p \)
- \( t = *p \)
- \( *s = r \)
Example:

- $p = &a \iff T_p \supseteq \{a\}$
- $q = &b \iff T_q \supseteq \{b\}$
- $*p = q \iff *T_p \supseteq T_q$
- $r = &c \iff T_r \supseteq \{c\}$
- $s = p \iff T_s \supseteq T_p$
- $t = *p \iff T_t \supseteq *T_p$
- $*s = r \iff *T_s \supseteq T_r$

Exercice : solve it!
Solution:

- $T_p = \{a\}$
- $T_q = \{b\}$
- $T_r = \{c\}$
- $T_s = \{a\}$
- $T_t = \{b, c\}$
- $O(a) = \{b, c\}$
- $O(b) = \emptyset$
- $O(c) = \emptyset$
Steensgaard’s pointer analysis

Idea: view pointer assignments as equality constraints

Question: Is it more or less precise? Why?

Question: Complexity?
Steensgaard’s pointer analysis

Idea: view pointer assignments as equality constraints

- $p = \& x \iff T_p \supseteq \{x\}$
- $p = q + o \iff T_p = T_q$
- $p = *q \iff T_p = * T_q \iff \forall x \in T_q, T_p = O(x)$
- $*p = q \iff * T_p = T_q \iff \forall x \in T_p, O(x) = T_q$
Steensgaard’s pointer analysis

Idea: view pointer assignments as equality constraints

- \( p = &x \iff T_p \supseteq \{x\} \)
- \( p = q + o \iff T_p = T_q \)
- \( p = *q \iff T_p = *T_q \iff \forall x \in T_q, T_p = O(x) \)
- \( *p = q \iff *T_p = T_q \iff \forall x \in T_p, O(x) = T_q \)

Question: Is it more or less precise? Why?

Question: Complexity?
Steensgaard is less precise than Andersen’s algorithm
Each equality constraint is equivalent to 2 inclusion constraints
Steensgaard’s constraints system include Andersen’s constraints
Think fix point: once you reached Andersen’s system fix point solution, you will keep growing to satisfy equality constraints
Complexity: $O(n \log(n))$ (process each constraint once using union-find)
Solution:

- $T_p = T_s = \{a\}$
- $T_q = T_t = T_r = O(a) = \{b, c\}$
- $O(b) = \emptyset$
- $O(c) = \emptyset$
IKOS uses Andersen’s approach

Based on Arnaud Venet’s paper: « A Scalable Nonuniform Pointer Analysis for Embedded Programs », SAS 2004

Compute points-to set (Andersen) and offset (Intervals) for each pointer

\[ D^# = \mathbb{P} \rightarrow (\mathbb{A} \cup \{\top\}) \times \mathbb{I} \]

Interprocedural

Flow insensitive

Context insensitive
Memory analysis (also called Value analysis) based on a reduced domain product of:
- Numerical domain for integers (by default, intervals)
- Pointer domain
- Null pointer domain
- Uninitialized variable domain
- Floating points are currently ignored

Based on Antoine Mine’s paper: « Field-Sensitive Value Analysis of Embedded C Programs with Union Types and Pointer Arithmetics », LCTES’06

- Interprocedural
- Context sensitive
Memory analysis - Pointer domain

- Pointer abstract domain
  \[ D_p^\# = \mathbb{V} \rightarrow (\mathbb{A} \cup \{\top\}) \times \mathbb{I} \]
- Pointwise order \( \sqsubseteq_p^\# \), Pointwise union \( \sqcup_p^\# \)
- \( (D_p^\#, \sqsubseteq_p^\#, \sqcup_p^\#) \) is a lattice
- Galois connection \((\alpha_p, \gamma_p)\) with the concrete semantics
- Reduction with the previous flow-insensitive pointer analysis
Abstract operations:

- $\llbracket p = \&x \rrbracket (\rho) = \rho [p \rightarrow (\{x\}, [0, 0])]$
- $\llbracket p = q + o \rrbracket (\rho) = \rho [p \rightarrow (\text{addresses}(\rho(q)), \text{offsets}(\rho(q)) + o)]$
- $\llbracket \ast p = q \rrbracket (\rho) = \rho$
- $\llbracket p = \ast q \rrbracket (\rho) = \rho [p \rightarrow (\top, ]-\infty, +\infty[)]$
Memory analysis - Pointer domain

- Abstract operations:
  - $\llbracket p = x \rrbracket^\#(\rho) = \rho [p \rightarrow (\{x\}, [0, 0])]$
  - $\llbracket p = q + o \rrbracket^\#(\rho) = \rho [p \rightarrow (\text{addresses}(\rho(q)), \text{offsets}(\rho(q)) + o)]$
  - $\llbracket *p = q \rrbracket^\#(\rho) = \rho$
  - $\llbracket p = *q \rrbracket^\#(\rho) = \rho [p \rightarrow (\top, ]-\infty, +\infty[)]$

- Question: $\llbracket p == q \rrbracket^\#(\rho) = ?$
- Question: $\llbracket p \neq q \rrbracket^\#(\rho) = ?$
Null pointer abstract domain

\[ D_n = \{ \bot, \text{Null}, \text{NonNull}, \top \} \]

\[ \mathbb{D}^\#_n = \mathbb{V} \rightarrow D_n \]

\[ \bot \sqsubseteq^\#_n \text{Null}, \bot \sqsubseteq^\#_n \text{NonNull}, \text{Null} \sqsubseteq^\#_n \top, \text{NonNull} \sqsubseteq^\#_n \top \]

\[ \text{Null} \sqcup^\#_n \text{NonNull} = \top \]

\[ (\mathbb{D}^\#, \sqsubseteq^\#, \sqcup^\#_n) \text{ is a lattice} \]

Galois connection \((\alpha_n, \gamma_n)\) with the concrete semantics
Uninitialized variable abstract domain

\[ D_u = \{ \bot, \text{Init}, \text{Uninit}, \top \} \]

\[ \mathbb{D}_u^\# = \forall \rightarrow D_u \]

\[ \bot \subseteq_u ^\# \text{Init}, \bot \subseteq_u ^\# \text{Uninit}, \text{Init} \subseteq_u ^\# \top, \text{Uninit} \subseteq_u ^\# \top \]

\[ \text{Init} \uplus_u ^\# \text{Uninit} = \top \]

\[ (\mathbb{D}_u^\#, \subseteq_u ^\#, \uplus_u ^\#) \text{ is a lattice} \]

Galois connection \((\alpha_u, \gamma_u)\) with the concrete semantics
Question: how to model the memory?

LLVM is low level, a byte representation is necessary

The C language is not type safe and is very permissive on casts
Question : how to model the memory?
LLVM is low level, a byte representation is necessary
The C language is not type safe and is very permissive on casts

We need to model correctly the following code :

```c
uint64_t x = 1;
uint32_t* p = (uint32_t*)&x;
p += 1;
uint32_t y = *p;
```

By the way, what is \( y \)'s value?
Memory model from « Formalizing the LLVM Intermediate Representation for Verified Program Transformations », POPL 2012

Memory cell mc = mb(size, byte) | mptr(blk, offset, index) | muinit

Memory state = (N, B, C)
- **N**: next block id
- **B**: \( \mathbb{Z}^+ \to \mathbb{Z}^+ \): block id to block size (bytes)
- **C**: \( \mathbb{Z}^+ \times \mathbb{Z}^+ \to MC \): (block id, offset in bytes) to memory cell
Example:

```c
int* p = (int*) malloc(sizeof(int) + sizeof(int*));
*p = 0x01020304;
int** q = (int**)(p + 1);
*q = p + 2;
```
Example:

```c
int* p = (int*) malloc(sizeof(int) + sizeof(int*));
*p = 0x01020304;
int** q = (int**)(p + 1);
*q = p + 2;
```

<table>
<thead>
<tr>
<th>blk id</th>
<th>offset</th>
<th>memory cell</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>mb(32, 4)</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>mb(32, 3)</td>
</tr>
<tr>
<td>0</td>
<td>2</td>
<td>mb(32, 2)</td>
</tr>
<tr>
<td>0</td>
<td>3</td>
<td>mb(32, 1)</td>
</tr>
<tr>
<td>0</td>
<td>4</td>
<td>mptr(l, 8, 0)</td>
</tr>
<tr>
<td>0</td>
<td>5</td>
<td>mptr(l, 8, 1)</td>
</tr>
<tr>
<td>0</td>
<td>6</td>
<td>mptr(l, 8, 2)</td>
</tr>
<tr>
<td>0</td>
<td>7</td>
<td>mptr(l, 8, 3)</td>
</tr>
</tbody>
</table>

By the way, what architecture could it be?
Memory analysis - Memory abstract domain

- Memory abstract domain
- Based on Antoine Mine’s paper: « Field-Sensitive Value Analysis of Embedded C Programs with Union Types and Pointer Arithmetics », LCTES’06
- Idea: abstract memory using cells: \( C(\text{address}, \text{offset}, \text{size}) \)
- Each cell is considered as a variable in the underlying abstract domain
- Cells may overlap
- \( C = A \rightarrow Z^+ \times Z^+ \)
- \( D_{\text{mem}} = C \times D_{\text{underlying}}^{\#} \)
- In IKOS, \( D_{\text{underlying}}^{\#} = D_{\text{num}}^{\#} \times D_{\text{ptr}}^{\#} \times D_{\text{null}}^{\#} \times D_{\text{unini}}^{\#} \)
- Pointwise partial order, Pointwise union
Abstract operations: forward to $\mathbb{D}^\#_{\text{underlying}}$, except memory read and write.

Memory write:
- set to $\bot$ if $p$ is null or uninitialized
- $(\text{points\_to, offset}) = \rho(p)$
- $\text{cells} = \text{realize\_write}(\text{points\_to, offset})$
- $\forall c \in \text{cells}, \text{strong\_update}(c, \text{rhs})$ or $\text{weak\_update}(c, \text{rhs})$

Memory read:
- set to $\bot$ if $p$ is null or uninitialized
- $(\text{points\_to, offset}) = \rho(p)$
- $\text{cells} = \text{realize\_read}(\text{points\_to, offset})$
- $\forall c \in \text{cells}, \text{strong\_update}(\text{lhs}, c)$ or $\text{weak\_update}(\text{lhs}, c)$
Example:

```c
int* p = (int*) malloc(sizeof(int) + sizeof(int*));
*p = 0x01020304;
int** q = (int**)(p + 1);
*q = p + 2;
```
Memory analysis - Memory abstract domain

Example:

```c
int* p = (int*) malloc(sizeof(int) + sizeof(int*));
*p = 0x01020304;
int** q = (int**)(p + 1);
*q = p + 2;
```

Abstract value at the end:

```
(malloc → {{0, 4}, {4, 4}})
(C(malloc, 0, 4) → [0x01020304, 0x01020304])
(C(malloc, 4, 4) → (malloc, [8, 8]),
   p → (malloc, [0, 0]),
   q → (malloc, [4, 4]))
(C(malloc, 4, 4) →NonNull, p →NonNull, q →NonNull)
(C(malloc, 0, 4) →Init, C(malloc, 4, 4) →Init, p →Init, q →Init))
```
static union {
    struct { uint8 al, ah, bl, bh, ... } b;
    struct { uint16 ax, bx, ... } w;
} regs;
regs.w.ax = X; // (1)
if (!regs.b.ah) { // (2)
    regs.b.bl = regs.b.al; // (3)
} else { // (4)
    regs.b.bh = regs.b.al; // (5)
}
// (6)
regs.b.al = X; // (7)
Memory analysis - Memory abstract domain

(1)

(2)

(3)

(4)

(5)

(6)

(7)
• Last step: check for properties at each statement location
• Checkers:
  • buffer overflow: $0 \leq offset$ and $offset + read\_size \leq buffer\_size$
  • division by zero: $divisor \neq 0$
  • null dereference: $p \neq Null$
  • uninitialized variable: $v \neq Uninit$
  • prover: $v \neq 0$
Syllabus

1. Introduction

2. IKOS

3. Analyses

4. Miscellaneous
   - Abstract domains implementation
   - Analyzing C++
   - Exception handling
   - Relational abstract domains
   - Function summarization
   - Integer overflow

5. Conclusion
Abstract domains implementation

- Separate domain ($\mathbb{V} \rightarrow \mathbb{D}$) are implemented with patricia trees
- Insertion and removal in $O(\log(n))$
- Merge in $O(n)$
- Transformation in $O(n)$
- Very cheap union!
Analyzing C++

Analyzing C++ is very tricky:

- Heavy chains of function calls because of templates
- The libc++ needs to be modeled
- Need to be precise on pointers for virtual method calls
- Handle exceptions
Analyzing C++ is very tricky:

- Heavy chains of function calls because of templates
- The libc++ needs to be modeled
- Need to be precise on pointers for virtual method calls
- Handle exceptions

Work in progress!
Exception handling

bb_1:
memory[x] = 9
_Z1fv._2 = call __Z14__ikos_unknownv()

*out_bb_1_to_bb_2_icmp_true:
_Z1fv._2 ne 0
_Z1fv._3 = -1

*out_bb_1_to_bb_3_icmp_false:
_Z1fv._2 eq 0
_Z1fv._3 = 0

bb_2:
_Z1fv._5 = call __cxa_allocate_exception(8)
_Z1fv._6 = bitcast _Z1fv._5
memory[_Z1fv._6] = $null
_Z1fv._8 = bitcast _ZTIDn
__v:6 = call __cxa_throw(_Z1fv._5, _Z1fv._8, $null)
unreachable

bb_3:
memory[x] = 0
return

_unified_exit:
Exception handling

**bb_8:**
main._17 = memory[y]

***_bb_8_split_icmp_false:***
main._17 ne 0
main._18 = 0

***_bb_8_split_icmp_true:***
main._17 eq 0
main._18 = -1

***out bb 8 merge icmp:***
___v:17 = invoke_Z13__ikos_assertb(main._18)

**bb_10:**
landingpad(main._21)
___v:18 = 0
___v:18 = add(___v:18, 0)
main._22 = extract_elem(main._21, ___v:18)
___v:19 = 0
___v:19 = add(___v:19, 8)
main._23 = extract_elem(main._21, ___v:19)

**bb_11:**
main._25 = bitcast_ZTIPv
main._26 = call llvm.eh.typeid.for(main._25)

**bb_9:**
\[ \mathcal{D}_{\text{exc}}^\# = \mathcal{D}^\# \times \mathcal{D}^\# \]

\[ \llbracket \text{throw}(e) \rrbracket^\#(N, E) = (\bot, N \cup E) \]

\[ \llbracket \text{landingpad}(e) \rrbracket^\#(N, E) = (E, \bot) \]

\[ \llbracket v = x \rrbracket^\#(N, E) = (\llbracket v = x \rrbracket^\#(N), E) \]

\[ (N_1, E_1) \sqcup^\# (N_2, E_2) = (N_1 \cup N_2, E_1 \cup E_2) \]
Intervals are very imprecise for loops with a non-deterministic bound

Solution: use a weakly-relational domain, such as the DBM domain

Difference-Bound Matrices

- Weakly-relational abstract domain

\[
\begin{bmatrix}
0 & m_{0,1} & m_{0,2} & \ldots & m_{0,n} \\
m_{1,0} & 0 & m_{1,2} & \ldots & m_{1,n} \\
m_{2,0} & m_{2,1} & 0 & \ldots & m_{2,n} \\
\vdots & \vdots & \vdots & \ddots & \vdots \\
m_{n,0} & m_{n,1} & m_{n,2} & \ldots & 0
\end{bmatrix}
\]

- \( m_{i,j} \in \mathbb{Z} \cup \{+\infty\} \)
- \( v_i - v_j \leq m_{j,i} \)
- \( v_0 = 0 \), thus \( v_i \in [-m_{i,0}, m_{0,i}] \)

Abstract operations require normalization

normalization:
- \( v_i - v_k \leq m_{k,i} \) and
- \( v_k - v_j \leq m_{j,k} \) \( \Rightarrow \)
- \( v_i - v_j \leq m_{k,i} + m_{j,k} \)

- cost \( O(n^3) \), \( n \) number of variables
Idea: keep a list of DBMs, where each DBM contains variables that are related to each other.

Union-Find structure to dynamically infer relations among variables

Normalization cost $O(n)$, $n$ number of DBMs
Variable packing

- Idea: keep a list of DBMs, where each DBM contains variables that are related to each other.
- Union-Find structure to dynamically infer relations among variables
- Normalization cost $O(n)$, $n$ number of DBMs

![Diagram showing DBMs and their variables]
Pointer analysis using function summarization.

<table>
<thead>
<tr>
<th>File</th>
<th>DBMs</th>
<th>Size</th>
<th>Var Packing</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>astree-ex</td>
<td>1.01s</td>
<td>36</td>
<td>0.13s</td>
<td>7</td>
</tr>
<tr>
<td>test-1</td>
<td>0.13s</td>
<td>27</td>
<td>0.03s</td>
<td>4</td>
</tr>
<tr>
<td>test-1-unsafe</td>
<td>0.13s</td>
<td>27</td>
<td>0.02s</td>
<td>4</td>
</tr>
<tr>
<td>test-10</td>
<td>0.03s</td>
<td>10</td>
<td>0.02s</td>
<td>4</td>
</tr>
<tr>
<td>test-10-unsafe</td>
<td>0.03s</td>
<td>11</td>
<td>0.02s</td>
<td>4</td>
</tr>
<tr>
<td>paparazzi-microjet</td>
<td>3241.14s</td>
<td>611</td>
<td>158.50s</td>
<td>88</td>
</tr>
<tr>
<td>gen2</td>
<td>&gt; 5h</td>
<td>?</td>
<td>7817.42s</td>
<td>367</td>
</tr>
<tr>
<td>aeroquad-servo</td>
<td>78.12s</td>
<td>71</td>
<td>1.33s</td>
<td>14</td>
</tr>
<tr>
<td>aeroquad-new</td>
<td>86.18s</td>
<td>65</td>
<td>0.76s</td>
<td>5</td>
</tr>
<tr>
<td>cornell</td>
<td>447.06s</td>
<td>226</td>
<td>2.64s</td>
<td>6</td>
</tr>
<tr>
<td>sporesate2-spore-pl</td>
<td>895.45s</td>
<td>?</td>
<td>10.29s</td>
<td>19</td>
</tr>
</tbody>
</table>
Other ideas

- Group variables depending on heuristics
- Use the gauge domain
Other ideas

- Group variables depending on heuristics
- Use the gauge domain

Work in progress!
IKOS uses dynamic inlining

Idea: analyse each function only once to build a summary
Problem: call graph cycle

```
main
  
  f
  ↓
  g

h
  ↓
  w

```
Problem: call graph cycle

- Strongly connected component analysis
- Topological order
- Bottom-up analysis (from the leaves to the root)
- Top-down analysis (from the root to the leaves)
Need a way to express the effect of a function call on the memory
More particularly on global variables and pointer parameters
Relation between the input memory state and the output memory state
Idea : Introduce *input cells* and *output cells*

\[ x = x + 1 \iff Cell\{x, 0, 4, Out\} = Cell\{x, 0, 4, In\} + 1 \]
Buffer overflow analysis using function summarization

<table>
<thead>
<tr>
<th>File</th>
<th>Inlining</th>
<th>Summaries</th>
<th>Warnings</th>
<th>Errors</th>
<th>Lines</th>
</tr>
</thead>
<tbody>
<tr>
<td>astree-ex</td>
<td>0.36s</td>
<td>0.57s</td>
<td>2/2</td>
<td>0/0</td>
<td>22 (1)</td>
</tr>
<tr>
<td>test-1</td>
<td>0.14s</td>
<td>0.16s</td>
<td>0/0</td>
<td>0/0</td>
<td>22 (1)</td>
</tr>
<tr>
<td>test-1-unsafe</td>
<td>0.13s</td>
<td>0.18s</td>
<td>0/0</td>
<td>2/2</td>
<td>22 (1)</td>
</tr>
<tr>
<td>test-10</td>
<td>0.10s</td>
<td>0.13s</td>
<td>0/2</td>
<td>0/0</td>
<td>20 (3)</td>
</tr>
<tr>
<td>paparazzi</td>
<td>154.03s</td>
<td>110.09s</td>
<td>0/0</td>
<td>0/0</td>
<td>24650 (199)</td>
</tr>
<tr>
<td>gen2</td>
<td>307.66s</td>
<td>&gt; 3h</td>
<td>195/?</td>
<td>0/?</td>
<td>22030 (82)</td>
</tr>
</tbody>
</table>
• Problem: LLVM integer types are signedness agnostic

• Because most instructions are signedness agnostic: add, sub, mul, etc.

• How to be be sound and precise?
  • Intervals with infinite precision: imprecise or unsound
  • Suppose integers are unsigned: imprecise
  • Suppose integers are signed: imprecise
  • Wrapped intervals: Jorge Navas’s paper « Signedness-Agnostic Program Analysis: Precise Integer Bounds for Low-Level Code »
  • Domain product: unsigned and signed
Syllabus

1. Introduction
2. IKOS
3. Analyses
4. Miscellaneous
5. Conclusion
Thank you. Questions?