Development and Evaluation of LAV: An SMT-Based Error Finding Platform

Milena Vujošević-Janičić and Viktor Kunčak

Verified Software, Theories, Tools and Experiments
Philadelphia, USA
January 28-29, 2012.

Agenda

- Motivation and short overview of LAV
- Modeling of programs
- Implementation and preliminary evaluation
- Conclusions

Agenda

- Motivation and a short overview of the system
- Modeling of programs
- Implementation and preliminary evaluation
- Conclusions

Motivation and a Short Overview of the System

- LAV* is a bug-finding tool, it works on the LLVM low-level intermediate representation

- LAV combines symbolic execution, SAT encoding of program's behavior and bounded model checking

- LAV generates correctness conditions that are passed to a suitable SMT solver

*LLVM Automated Verifier

Motivation and a Short Overview of the System — Ex.

```
0:  int main()
1:  {
2:  int a0, a1, k, div = 1;
3:  if(a0>0)
4:         a0 = 1;
5:    else a0 = -1;
6:  if(a1>0)
7:         a1 = 1;
8:    else a1 = -1;
9:  div = a0+a1+2;
10: k = 1/div;
11: }
```

```
line 10: UNSAFE

function: main
error: division_by_zero
3: a0 == 0, a1 == 0, div == 1
5: a0 == -1, a1 == 0, div == 1
6: a0 == -1, a1 == 0, div == 1
8: a0 == -1, a1 == -1,  div == 1
10: a0 == -1, a1 == -1, div == 0
```

C code example (left) and LAV output (right)

Motivation and a Short Overview of the System — Ex.

# ifs & # vars	# paths	LAV bug in the first path	 bug in the last path	 no bug	KLEE bug in the first path	 bug in the last path	 no bug
2	4	0.07	0.07	0.07	< 1	0.05	0.05
5	32	0.18	0.19	0.18	< 1	0.55	0.55
10	1024	0.41	0.46	0.38	< 1	45.00	45.00
11	2048	0.42	0.54	0.43	< 1	107.00	107.00
12	4096	0.50	0.67	0.50	< 1	268.00	268.00
20	1'048'576	0.73	1.82	0.72	< 1	TO	TO
60	2^{60}	25.00	39.00	4.18	≈ 1	TO	TO
100	2^{100}	153.00	111.00	15.00	≈ 1	TO	TO

Path Explosion Example

Agenda

- Motivation and a short overview of the system
- Modeling of programs
- Implementation and preliminary evaluation
- Conclusions

Instructions, Variables, and Data types

- In LLVM:
  - Each program function consists of blocks of instructions, with no branching and no loops
  - Each block can be entered only at its entry point, and left only through its last command

- In LAV: Block summary, $Transformation(b)$, is constructed by symbolic execution; it describes the way in which a block b transforms the store of the program

Instructions, Variables, and Data types: Instructions

- Each instruction is symbolically executed, it transforms the store of a program and may add some constraints

code	store		additional constraints
	a	b	
`int a, b;`	a0	b0	*empty*
`b += a;`		b0+a0	*empty*
`a = 3;`	3		*empty*

- Values of the variables at the exit point of the block are given in terms of the values of the variables at the entry point

Instructions, Variables, and Data types: Pointers

- Buffers — sequences of memory allocated statically or dynamically; accessible by a pointer and an offset

- For a pointer p, $left(p)$ and $right(p)$ keep track of numbers of bytes reserved for the pointer p on its left and its right

code	store		additional constraints
	`b`	`p`	
`int b[10], *p;`	`b0`	`p0`	$left(b_0) = 0 \wedge right(b_0) = 40$ $\wedge left(p_0) = 0 \wedge right(p_0) = 0$
`p = b+3;`		`b0+12`	*empty*

Instructions, Variables and Data types: Memory

- For accessing memory via pointers: theory of arrays
- Flat memory model is used
- The value of the array mem is also kept in the store

code	store				additional constraints
	`mem`	`b`	`p`	`i`	
`int b[10], *p, i;`	`m0`	`b0`	`p0`	`i0`	$left(b_0) = 0 \wedge right(b_0) = 40$ $\wedge left(p_0) = 0 \wedge right(p_0) = 0$
`p = b+3;`			`b0+12`		
`*(p+i) = 5 ;` buffer overflow?	`store(m0, b0+12+` `+i0*4, 5)`*				

*assuming $sizeof(int) = 4$

Instructions, Variables and Data types: Example

- `*(p+i)` introduces a buffer overflow iff $left(p) \leq i \cdot sizeof(*p) < right(p)$ is false

instruction	safety condition for the instruction	available constraints
`*(p+i) = 5;` buffer overflow?	$left(b_0 + 12) \leq i_0 \cdot 4$ $\wedge$ $i_0 \cdot 4 < right(b_0 + 12)$	$left(b_0) = 0$ $\wedge right(b_0) = 40$ $\wedge left(p_0) = 0$ $\wedge right(p_0) = 0$

- Additional constraint (an instance of specific axioms): $left(b_0 + 12) = left(b_0) - 12 \wedge right(b_0 + 12) = right(b_0) - 12$

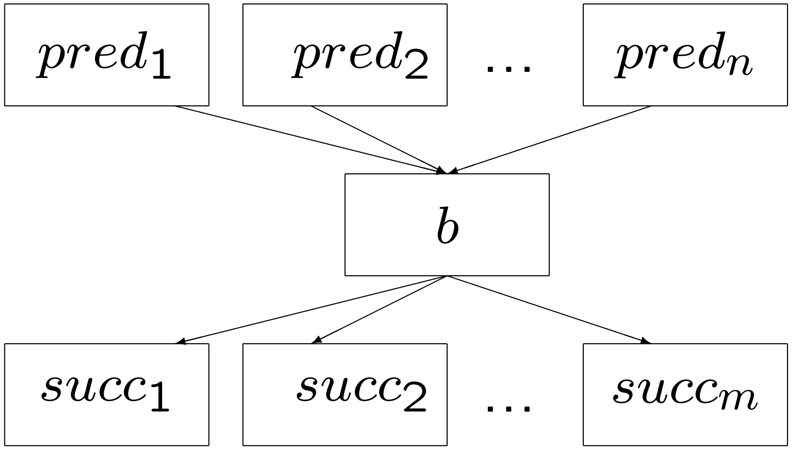
Instructions, Variables and Data types: Function Calls

- Function calls are modeled according to available information about the function, one of:
  - Contract available
  - Definition available
  - Nothing available

Modeling Control Flow and Interprocedural Analysis

- $Transformation(b) = StoreUpdate(b) \bigwedge AdditionalConstraints(b)$

- Links between blocks: propositional variables



- Postcondition of a block contains control flow information:

  $Postcondition(b) = EntryCond(b) \ \wedge \ Transformation(b) \ \wedge \ ExitCond(b)$

Modeling Control Flow and Interprocedural Analysis

- Two techniques for dealing with loops supported:
    - Underapproximation — loops are unrolled fixed number of times
    - Overapproximation — unrolled code simulates first m and last n entries to the loop
- Postcondition of a function — a conjunction of postconditions of its blocks
- Recursive functions are not supported yet

Constructing Correctness Conditions

- Correctness/incorrectness conditions are of the form:

  (CC) $Context \Rightarrow safe(c)$

  (IC) $Context \Rightarrow \neg safe(c)$

- $safe(c)$ — safety condition of an instruction (given by a bug definition or by an annotation within the code)

- $Context$ is a formula describing context, e.g.:

  empty context — `a/3;`

  block context — `b=3; b++; a/b;`

  function context — `b=3; if(c>d) b++; a/b;`

  wider context — `int f(int a, int b) {return a/b;}`
  `... f(a, 3) ...`

Constructing Correctness Conditions

- If $\neg CC$ is UNSAT: c is safe, and it is also safe in all wider contexts (if it is reachable)
- If $\neg IC$ is UNSAT: c is flawed, and it is also flawed in all wider contexts (if it is reachable)
- If both $\neg CC$ and $\neg IC$ are UNSAT: the context is inconsistent so c is unreachable, and it is unreachable in all wider contexts
- If both $\neg CC$ and $\neg IC$ are SAT for some context: c is unsafe; in some wider context c may have different status

Translating Correctness Conditions to SMT Formula

- Integers and operations over integers, one of:
  — arbitrary-precision numbers and linear arithmetic (`LA`)
  — finite-precision numbers and bit-vector arithmetic (`BVA`)

- Functions $left$ and $right$, one of:
  — theory of uninterpreted functions (EUF)
  — Ackermannization

- Functions $select$ and $store$:
  — theory of arrays (`ARRAYS`)

- Several SMT solvers provide support for combinations of the above theories

Agenda

- Motivation and a short overview of the system
- Modeling of programs
- Implementation and preliminary evaluation
- Conclusions

Implementation

- The tool LAV is implemented in C++ and open source: `http://argo.matf.bg.ac.rs/?content=lav`

- Supported solvers: Boolector (BVA and ARRAYS), Yices and MathSAT (LA, BVA, EUF) and Z3 (LA, BVA, EUF, ARRAYS)

- For unsafe and flawed commands, a counterexample which includes program trace and values of program variables along this trace is extracted from the model generated by a solver

Related Tools

Tool	LAV	CBMC	ESBMC	KLEE	LLBMC	CALYSTO	PEX
Frontend	LLVM	goto-cc	goto-cc	LLVM	LLVM	LLVM	.NET
Theories	- LA BV EUF ARR.	PL - - - -	- LA BV EUF ARR.	- - BV - ARR.	- - BV - ARR.	- - BV - -	- LA BV EUF ARR.
Solvers	MathSAT Boolector Z3 Yices	MiniSAT2 - - -	CVC Boolector Z3 -	STP - - -	- Boolector Z3 -	Spear - - -	- - Z3 -

- Other related tools: CORRAL,S2E,CPAChecker,ESC/JAVA, ...

Experimental Comparison

- Limited experimental comparison with KLEE, CBMC and ESBMC.
- Based on the NECLA static analysis benchmarks (44 out of 57 benchmarks)
- All the tools checked the benchmarks for pointer errors, buffer overflows, division by zero, and user-defined assertions

Experimental Comparison: Results

Tool	LAV	CBMC	ESBMC	KLEE
Best times, default params.	45%	2%	0%	**47%**
Best times, upp. bound	0%	22%	**56%**	NA
Best times, unw. bound	**66%**	17%	44%	NA
Confirmed missed bugs	**0%**	1%	7%	2%
False alarms	9%	11%	8%	**0%**
Tool failure	**0%**	11%	4%	23%
Timeouts	**11%**	26%	26%	13%

LAV's performance is comparable to other tools

Application in Education: Experiments

- A tool that could help students and teachers to detect bugs would be very benefitial

- 157 programs written by students at exams during an introductory course in programming analyzed

Problem	# Solutions	Avg. Lines	Avg. Reported Bugs	Avg. False Alarms
calculations	60	30	0.82	0.05
arrays and matrices	71	46	4.20	0
strings and structures	26	60	2.92	1.11
Summary	157	42	2.69	0.20

Application in Education: Analysis of Results

	calculations & arrays and matrices	strings and structures
Most frequent bug	buffer overflow	null pointer dereferencing
# programs with the above bug	81	15
# bugs	225	46
Second most frequent bug	devision by zero	buffer overflow
# programs with the abouve bug	22	15
# bugs	22	30

- The vast majority of bugs due to wrong expectations e.g., that input parameters of programs will meet certain constraints

- This explains the large number of bugs in the corpus — adding only one check in a program would typically eliminate several bugs

Application in Education: One Simplified Student's Code

```
1:  #include<stdio.h>
2:  #include<stdlib.h>
3:  int power(int n)
4:  {
5:  int i, pow;
6:  for(i=0, pow=1; i<n; i++, pow*=10);
7:  return pow;
8:  }
9:
10: int get_digit(int n, int d)
11: {
12: return (n/power(d))%10;
13: }
14:
15: int main(int argc, char** argv)
16: {
17: int n, d;
18: n = atoi(argv[1]);
19: d = atoi(argv[2]);
20: printf("%d\n", get_digit(n, d));
21: }
```

```
line 12: UNSAFE
line 18: UNSAFE
line 19: UNSAFE
line 20: 12: UNSAFE

function: get_digit
error: division_by_zero
line 12: d == 1073741824,

function: main
error: buffer_overflow
line 18: argc == 1, argv == 1

function: main
error: buffer_overflow
line 19: argc == 2, argv == 1

function: main
error: division_by_zero
line 20: 12: argc == 512,
             argv == 1,
             d == 1073741824, n == 0
```

Agenda

- Motivation and a short overview of the system
- Modeling of programs
- Implementation and preliminary evaluation
- Conclusions

Conclusions

- LAV combines symbolic execution, SAT encoding of programs behavior and bounded model checking
- LAV's performance is comparable to other tools (based on a limited benchmarks)
- Promissing directions for applications in education

Thank you

Future Work

- Take advantage of LLVM code optimizations

- Further improvement of modeling power and efficiency

- LAVedu — for real world applications in education

Modeling Control Flow

$$\begin{aligned}
Postcondition(b) &= EntryCond(b) \;\wedge\; Transformation(b) \;\wedge\; ExitCond(b) \\
EntryCond(b) &= activating(b) \wedge initialize(b) \\
Transformation(b) &= \bigwedge_{v \in V} (e(b,v) = e_v) \bigwedge AdditionalConstraints(b) \\
ExitCond(b) &= jump(b) \wedge leaving(b) \\
activating(b) &= \left(\bigvee_{pred \in Predcesors} transition(pred, b) \right) \Leftrightarrow active(b) \\
initialize(b) &= \bigwedge_{pred \in Predcesors} \left(transition(pred, b) \Rightarrow \bigwedge_{v \in V_f} e(pred, v) = s(b, v) \right) \\
jump(b) &= \bigwedge_{succ_i \in Successors} ((active(b) \wedge e(b, c_i)) \Leftrightarrow transition(b, succ_i)) \\
leaving(b) &= active(b) \Leftrightarrow \left(\bigvee_{succ \in Successors} transition(b, succ) \right)
\end{aligned}$$