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

System Description

Milena Vujošević-Janičić[1] and Viktor Kuncak[2]

[1] Faculty of Mathematics, Studentski trg 16, 11000 Belgrade, Serbia
`milena@matf.bg.ac.rs`

[2] School of Computer and Communication Sciences, EPFL, Station 14, CH-1015 Lausanne, Switzerland `viktor.kuncak@epfl.ch`

Abstract. We present design and evaluation of LAV, a new open-source tool for statically checking program assertions and errors. LAV integrates into the popular LLVM infrastructure for compilation and analysis. LAV uses symbolic execution to construct a first-order logic formula that models the behavior of each basic blocks. It models the relationships between basic blocks using propositional formulas. By combining these two kinds of formulas LAV generates polynomial-sized verification conditions for loop-free code. It uses underapproximating or overapproximating unrolling to handle loops. LAV can pass generated verification conditions to one of the several SMT solvers: Boolector, MathSAT, Yices, and Z3. Our experiments with small 200 benchmarks suggest that LAV is competitive with related tools, so it can be used as an effective alternative for certain verification tasks. The experience also shows that LAV provides significant help in analyzing student programs and providing feedback to students in everyday university practice.

1 Introduction

In this paper we present LAV, a tool for finding bugs (such as buffer overflows and division by zero) and for checking functional correctness conditions given by assertions.[3] We evaluated our approach primarily on programs in the C programming language (where the opportunities for errors are abundant), but it can be also used for other languages. LAV works on the LLVM low-level intermediate representation, and applies to other similar representations. LLVM[4] is a compiler framework widely used for compilation tasks, but also for verification as in tools KLEE [10], Calysto [2], and LLBMC [20]. LLVM has front-ends for C, C++, Ada and Fortran, and there are further external projects for translating a number of other languages to LLVM.

* This work was partially supported by the Serbian Ministry of Science grant 174021 and by Swiss National Science Foundation grant SCOPES IZ73Z0_127979/1.

[3] LAV stands for *L*LVM *A*utomated *V*erifier. LAV also means *lion* in Serbian.

[4] `http://llvm.org/`

The approach we propose combines symbolic execution [17], SAT encoding of program's control-flow, and elements of bounded model checking [6]. LAV represents program meaning using first-order logic (FOL) formulas and generates final verification conditions as FOL formulas. Each block of code is represented by a FOL formula obtained through symbolic execution (blocks have no internal branchings and no loops). Symbolic execution, however, is not performed between different blocks. Instead, relationships between blocks are modeled by propositional variables encoding transitions between blocks. LAV constructs formulas that encode block semantics once for each block. It then combines these formulas with propositional formulas encoding the transitions between blocks. The resulting compound FOL formulas describe correctness and incorrectness of individual instructions. LAV checks them using one of the several SMT solvers. If a command cannot be proved to be safe, LAV translates a potential counterexample from the solver into a program trace that exhibits this error. It also extracts the values of relevant program variables along this trace. Our experiments with 200 benchmarks suggest that LAV is competitive with related tools. The experience also shows that LAV provides significant help in analyzing student programs, providing feedback to students in everyday university practice.

2 Motivating Example

Verification tools based on symbolic execution proved to be very efficient for many verification tasks. However, they also have weaknesses that make them, in some cases, less applicable than desired. As a simple example, consider the code in Figure 1. There are four paths to be explored to check whether the program has division by zero bug in line 10. If, in line 9, *div* is assigned $a0 + a1 + 2$, then the bug occurs in the first of these four paths (assuming that `else` branches are considered first). If, on the other hand, *div* was assigned $a0 + a1 - 2$, then the bug occurs in the last of these four paths. If *div* was assigned $a0 + a1 + 3$, then there is no division by zero bug in the line 10. In summary, if there is no bug, or the bug is found in the last path, then all paths need to be explored.

If we generalize the example from Figure 1 to have n instead of two variables, and to have n instead of two `if` commands, then there are 2^n paths to consider.[5] This is the well-known problem of path explosion. Instead of considering all paths separately, our approach models the control flow in a more compact way that uses symbolic execution only within fragments (blocks) without branching. The size of a formula in this approach is polynomial in the number of blocks. Consequently, the path explosion does not occur in the verification tool itself. The exploration of many possible paths is transferred to a reasoner (i.e. theorem prover) which receives case splits only implicitly in form of disjunctions within a formula representing verification condition. Thanks to the use of learning, the reasoner typically solves such formula much more efficiently than by considering

[5] Code with this control structure is not unrealistic. Many real-world functions, such as lexical analyzers and parsers, contain a large number of `if` commands [16].

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

			LAV			KLEE	
# ifs & # vars	# paths	bug in the first path	bug in the last path	no bug	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	1'024	0.41	0.46	0.38	< 1	45.00	45.00
11	2'048	0.42	0.54	0.43	< 1	107.00	107.00
12	4'096	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

Fig. 1. Code example (left-hand side, up) and LAV output for $div = a0 + a1 + 2$ (right-hand side, up). The table shows the number of `if`-s and variables, the number of paths, the time spent by LAV and then by KLEE if a bug occurs in: the first path, the last path, and if there was no bug. Times are given in seconds. TO means timeout.

all cases. Using this approach, we avoid the path explosion problem using modern, powerful theorem provers, such as SMT solvers. This observations motivated our approach, and shows good results in our examples. In the above example, as n increases, the time spent by our tool increases polynomially, instead of the clearly exponential growth for the symbolic execution tool KLEE [10], as shown in the table in Figure 1. The table shows that the verification of a program (in the case where there are no bugs) with large number of paths is infeasible for KLEE. The table also shows that the time needed by KLEE to find a bug (if it exists) heavily depends on the path that leads to it. Neither of these holds for our tool: verification of a correct program or finding a bug both follow a construction of a single first order formula that is passed to an SMT solver. Certain differences in the solving times (for bugs occurring in different paths) are not consequences of the modeling process, but rather of the internal operation of the solver.

3 Modelling Variables, Data Types, and Blocks

Store and Blocks. Each program function consists of blocks, while each block is a sequence of commands. The execution can enter a block only at its entry point, and exit only through the last command of the block. A *store* of a program is a map from variables to values given by variable's type. Each instruction transforms the store and may add constraints over variables. In our approach, symbolic execution is used to compute a FOL formula $Transformation(b)$ that describes how the block b transforms the store of the program.

Denote by $s(b, v)$ the value of a variable v at the entry point of the block b and by $e(b, v)$ the value of v at the exit point. After the block is symbolically executed, the formula $Transformation(b)$ is constructed based on the values of the variables at the end of the block. In the general case it is given by $\bigwedge_{v \in V}(e(b, v) = e_v) \bigwedge AdditionalConstraints(b)$, where V is the set of variables and where e_v is the value of v at the end of the block expressed in terms of initial values $s(b, v)$. The formulas $AdditionalConstraints(b)$ are introduced for modeling certain operations (as described in the rest of this section). Another formula $Transformation(b, i)$ is defined analogously, but considering only the first i instructions of the block b.

Buffers, Structures and Unions. Buffers are sequences of memory allocated statically or dynamically and accessible by a pointer and an offset. While these pointers are treated in our tool as any other simple variables, they are also associated with sizes of corresponding buffers. To deal with buffer sizes, we introduce two functions: $left(p)$ and $right(p)$ for numbers of bytes reserved for the pointer p on its left and its right hand side. For example, the command `*(p+i)` introduces a buffer overflow iff $left(p) \leq i \cdot sizeof(int) < right(p)$ is false. The argument of the functions $left$ and $right$ can be a pointer or a sum of a pointer and an offset (which is of the integer type), in accordance with properties of pointers in C. Note that it always holds $left(p + n) = left(p) - n$ and $right(p + n) = right(p) - n$. These equalities can be considered as axioms about the functions $left$ and $right$, but, instead of introducing a universally quantified formula into the generated formula, we add only its relevant instances to the set of additional constraints attached to the block. More complex types, such as structures and unions, are also treated as sequences of individual bytes.

Memory Contents. For simplicity and precision, LAV uses a flat memory model: it treats the entire memory as an array *mem* of memory locations, that may get updated during the symbolic execution, just as any other variable. For modeling commands that access the memory via pointers we use the theory of arrays. The theory of arrays provides functions for storing a value at a certain index in the array (*store*) and for reading a value at a certain index in the array (*select*). Also, if there is a reference operator on a local variable within a function, then this variable is not tracked through its slot in the store, as other variables, but is tracked through the memory content. A run-time library guarantees that during the program execution all active variables and dynamic objects are assigned non-overlapping memory locations. Instead of adding conditions of the form $p \neq q$ for each pair (p, q) of addresses of variables or dynamic objects, a more efficient approach is used: each address p is assigned (within correctness conditions) a unique *fixed* number (or *magic number* [19]).

Global Variables. Global variables are accessible in all functions (and, hence, in all blocks), but instead of representing them individually within all functions, they are modeled by the variable modeling memory. The reasoning involving the theory of arrays can be expensive, but if there are not many updates of global variables, this model can still be more suitable. If there are many updates of global variables, then, in some cases, global variables are tracked through their slots in the store, just as local variables.

Function Calls. Function calls are modeled according to the available information about the function. If a *contract* (i.e., a *summary*) of the function is available, then the current store is updated and additional constraints are added according to this contract. If a contract of the function is not available, while the definition of function is, then an interprocedural analysis is required (performed as described in the next section). If neither a contract nor the definition of the function are available, then the memory contents (i.e. the current array *mem*) is set to a new (fresh) variable as an effect of the function call.

4 Modelling Control Flow

Intraprocedural Loop-free Control Flow. Whereas single blocks are represented by FOL formulas constructed using symbolic execution, LAV encodes relationships *between* blocks by propositional variables and SAT formulas.

Assume, for a moment, that the program has no loops in the control-flow graph. A path in this graph is then determined by the sequence of nodes (representing blocks) and edges (representing transitions from one block to another). For each block and for each transition we introduce a propositional variable that denotes whether the corresponding node or transition is in the path. Valid paths through the graph are encoded by entry conditions (represented by $EntryCond(b)$) and exit conditions ($ExitCond(b)$). Entry conditions are conditions that must hold at the entry point of the block b — b is in the path iff there is a transition to b from some of its predecessors; if the block b is reached from the block *pred*, then the initial values of the variables within the block b are equal to the values of the variables at the exit point of the block *pred*. Exit conditions are conditions that must hold at the exit point of the block b — each block must lead somewhere (either to some other block or to the exit of the function). If the block b was active and if the exit condition c_i of the block b was met, then the control is passed to the successor $succ_i$. The final formula *Description(b)* describing the block b is defined as $EntryCond(b) \wedge Transformation(b) \wedge ExitCond(b)$.

Loops. Loops are eliminated by unrolling. This way, the control flow graph of the function has no cycles and the above modeling mechanism can be applied. Our system supports two techniques for dealing with loops: underapproximation of loops and overapproximation of loops. In the former case, loops are unrolled a fixed number of times n, as in bounded model checking. If the unrolled code verifies successfully, it means that the original code has no bugs for n or less passes through the loop. In the latter case, the unrolled code simulates first n and last m entries to the loop, where n and m are configurable parameters. After the first n unrollings, we insert a block of code which simulates execution of an arbitrary loop iteration by resetting the values of each loop target (i.e. the value of each variable that is updated by any statement in any block in the loop), similarly as, for instance, in [4]. This resetting of the values may cause loss of precision and therefore may introduce false alarms. To overcome this problem, it is necessary to have loop invariants annotated in the program (or to use techniques that infer them automatically in some cases). If the overapproximated code is verified, then the original code has no bugs too.

Interprocedural Control Flow. Starting with block descriptions as building blocks, and given that there is a unique entry and a unique exit point for each function, [6] the description of a function is constructed as a conjunction of descriptions of the function blocks. Recursive function calls can be unrolled in a similar way as loops.

5 Correctness Conditions

To check whether a command leads to an error LAV builds two formulas, of the form $C \Rightarrow (\neg)safe(c)$. Here C is a formula describing a context: in the empty context (i.e., if the command is considered on its own), C equals $\top$, in the block context, C equals $Transformation(b,i)$ (if c is i-the instruction in b), and in the function context, C equals $\bigwedge Description(b')$ for all function's blocks b' that precede b. $(\neg)safe(c)$ is a formula describing (in)correctness condition of a command — it can be given by a bug definition (division by zero, buffer overflow, dereferencing null pointers) or it can be given by an annotation in the form of C logical expressions within assert commands.

If $safe(c)$ holds (under assumption C), then the command is *safe* and if $\neg safe(c)$ holds, the command c is *flawed*. If both $safe(c)$ and $\neg safe(c)$ hold for some context C (i.e. if C is inconsistent), then the command c is *unreachable*.[7] If neither $safe(c)$ nor $\neg safe(c)$ hold in a general case, then the command c is considered *unsafe*. The difference between a flawed and an unsafe command is that the flawed command always leads to an error in the program (if it is reachable), while unsafe command leads to an error only in some cases, depending on the context of the command, i.e., to the path condition leading to the command. Each command is first checked within the empty context. If it gets the unsafe status, the command is then checked within the block context. If it keeps the unsafe status within the block context, then it is checked within the function context. If a command is detected to be safe or flawed at some stage, then this status for the command is final and wider contexts are not considered. For each function call, correctness conditions for all unsafe commands from the called function are checked in the calling context.

Checking the status of the command c in the context C can always be done within one or two prover calls. After the context is added into the solver, the safety property of the command is first checked. If the solver proves it, then this means that it is either safe or unreachable. In both cases, it is not flawed or unsafe so there is no need for any further checks. If the solver cannot prove it,

[6] This can be ensured, as in the LLVM code.

[7] Note that, even if it was proved that the command c is safe, unsafe or flawed in some context, it still does not mean that it is reachable in some wider context.

then the negation of the safety property is checked and according to the answer of the prover it can be concluded if it is a flawed or an unsafe command. If one does not want to distinguish between flawed and unsafe commands (by selecting this tool's option, in case when trade-off of solving time and precision is needed), then this second call is omitted.

When proving different (in)correctness conditions in one function context, formulas corresponding to unnecessary but already considered blocks can be kept in the context (thanks to deductive monotonicity). This enables incremental approach, suitable for SMT solvers that can typically take advantage of the results learned from the previous proof attempts [5].

6 Transforming a Code Model into an SMT Goal

The quantifier-free formula that models program code typically involves arithmetic, logical, and relational operators, but also functions such as $left$ and $right$. We model integers by arbitrary-precision numbers (using linear arithmetic) or, if so selected by a command-line argument, by finite-precision numbers in bit-vector arithmetic. The functions $left$ and $right$ are considered to be *uninterpreted functions*, with their specific properties added to the correctness conditions. These functions are dealt by: (i) the theory of uninterpreted functions or by (ii) Ackermannizing the goal [1]. Each of these options can lead to more efficient reasoning in some cases [8]. Memory contents are represented by the theory of arrays, or can be just ignored (leading to a less precise reasoning) because of a high computational cost. Overall, the models of code typically require: bit-vector arithmetic (or linear arithmetic), theory of uninterpreted functions (or, alternatively, Ackermannization), and optionally theory or arrays. There are several SMT solvers that provide support for such combinations of theories.

7 Implementation

The approach described in previous sections is implemented in C++ as a tool LAV. The tool is publicly available and open-source.[8] The tool is built on top of LLVM that serves as a front-end to input programs. LLVM is developed primarily for the programming language C, but can be used for other languages as well. Thanks to this universality, LAV successfully handled several non trivial examples in Fortran. (Dealing with object oriented languages requires certain additions to our tool, which are planned for our future work.)

The LLVM programs are processed and the formulas representing correctness conditions are generated following the approach described in the sections 3 and 4. As the default parameters, loop unrolling simulates the first two and the last one passes through the loop, but this can be changed by the user. Correctness conditions, built as described in Section 5, can be translated to a number of theories and their combinations, as described in Section 6. Currently, there is

[8] `http://argo.matf.bg.ac.rs/?content=lav`

support for export to linear arithmetic, bit-vector arithmetic, the theory of uninterpreted functions (and Ackermannization, as its alternative), and the theory of arrays. Recursive function calls and support for floating point number arithmetic are not implemented yet. Automated inference of loop invariants is not part of LAV. There is currently no general notation for function contracts, but contracts of certain memory-safety-critical functions such as `malloc`, `calloc`, `realloc`, `free`, `strcpy` are encoded directly in C++, within LAV implementation. In addition, statements or assumptions (concerning loops or function calls) can be given in the form of C logical expression within *assume* function call.

The generated formulas are passed to SMT solvers, by using function calls from their APIs. Currently, supported solvers are Boolector [7] (for the theories BVA and ARRAYS), Yices [15] and MathSAT [9] (LA, BVA and EUF) and Z3 [14] (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 (if a corresponding option is used).

8 Evaluation and Comparison to Related Tools

Related tools. CBMC [12] and ESBMC [13] are bounded model checkers for ANSI C programs. As a front-end, ESBMC uses CBMC, which, in turn, uses `goto-cc`, a compiler from C and C++ into GOTO-programs. On the other hand, LAV uses LLVM, which is a multi language platform. CBMC and ESBMC unwind program loops, while LAV supports both underapproximation and overapproximation of loops. CBMC translates correctness conditions to propositional logic and instances of SAT. Like LAV, ESBMC converts verification conditions using different background theories and passes them directly to an SMT solver.

KLEE, Calysto, S2E, and LLBMC use the LLVM compiler infrastructure as a front-end to input programs. KLEE [10] is a symbolic execution tool, which employs a variety of constraint solving optimizations, represents program states compactly and uses search heuristics to improve code coverage. KLEE is used within other verification tools, such as the S2E platform [11] for developing analyzers. S2E introduces selective symbolic execution, relaxed execution consistency models, and supports analysis of binaries. Calysto [2] is a static checker for NULL pointer dereferencing and user-provided assertions. Calysto preserves the structure of the analyzed program in the phase of symbolic execution and uses it as an automatic abstraction/refinement framework for filtering verification conditions. It handles loops and pointers in an unsound manner; for example, loops are unrolled only once. The above tools are closely integrated with their theorem provers and with theories that these provers use. This is in contrast to LAV, which can chose amongst different SMT solvers and theories. LLBMC [20] is a tool for low-level bounded model checking of C programs, which was developed in parallel with our work. It focuses on covering memory consistency constraints; it models control flow of a program in a similar way as LAV and uses Boolector (the theory of bit-vectors and arrays) as the back-end solver.

Table 1 summarizes the front ends, supported theories, and solvers used by considered tools.

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

Table 1. Frontends, supported theories and solvers for considered tools

Experimental Comparison. We describe experimental comparison of LAV with KLEE, CBMC and ESBMC. At the time of writing, LLBMC and Calysto are not publicly available, so we were not able to include them in this evaluation. We also did not include the symbolic execution tool PEX [21], because it deals with C# and not with C.

The experimental comparison of LAV with the related tools was based on the NECLA static analysis benchmarks [19]. These benchmarks contain C programs that demonstrate common programming situations that arise in practice such as interprocedural data-flow, aliasing, array allocation modes, array size propagation, string library usage and so on. The ability of different techniques to prove them (in)correct is an indication of their areas of strengths and weaknesses. All ANSI C programs from the NECLA static analysis benchmarks are included in our evaluation except those that contain recursive function calls, string library usage and which depend on floating point number calculations (44 out of 57 benchmarks are included).

All the tools checked the benchmarks for pointer errors, buffer overflows, division by zero, and user-defined assertions. The tools terminated (with an appropriate report) when a first flawed command was found or when the code was verified. The experiments were performed with default parameters for each tool. We consider the results obtained with default parameters the most indicative since the user does not have to examine the code in order to determine unwinding and other parameters. If some tool, for its default parameters, produced an irregular output (such as an error message, a false alarm or time out), then it was invoked again with a loop unwinding parameter — with the upper bound of the loop, if it exists. If that call produced an irregular output or the upper bound of the loop does not exist, then a small loop bound was used. LAV and ESBMC were used with a solver for the theory of bit-vectors and arrays (because for KLEE this is the only option).

Experiments were performed on a system running Ubuntu, with Intel processor on 1.6GHz and with 1GB of RAM memory. The results are given in Table 2. The table contains a name of the benchmark (`bnc`), the number of code lines (`#L`), the number of loop unwindings (`#UNW`), whether or not the program contains some flawed commands (`F/V`), the name of the tool and the name of the solver used (for some tools, in some cases, the verification did not require invoking a solver). Abbreviations used are: `B` – Boolector, `NA` – not applicable, `FA` – false alarm, `UB` – missed (not discovered) bug, `U` – unreachable bug, `*` – SAT/SMT solver was not called, `ERR` – error, `TO` — time out, and `Z3` – solver Z3 was called instead of Boolector. The summary of the results is given in Table 3.

			F /	LAV		CBMC	ESBMC		KLEE
bnc.	`#L`	`#UNW`	V	B	Z3		B	Z3	
ex1	21	def	V	FA	FA	**5.02***	**5.02***	**5.02***	**0.19**
		513	V	TO	TO	**5.02***	**5.02***	**5.02***	NA
		3	V	0.90	0.35	**0.14***	**0.14***	**0.14***	NA
ex2	40	def	V	0.63	**0.54**	TO	TO	TO	ERR
		1024	V	TO	TO	ERR	Z3	**67.48**	NA
		3	V	1.03	0.47	ERR	Z3	**0.27**	NA
ex3	24	def	F	**0.04**	0.06	0.08	0.09	0.09	**0.04**
ex4	16	def	F	0.13	0.24	0.14	0.15	0.18	**0.02**
ex5	18	-	V	**0.02**	**0.02**	0.06*	0.06*	0.06*	**0.02**
ex6	21	-	V	0.07	0.11	0.07	Z3	0.07	**0.03**
ex7	28	def	V	**0.22**	**0.22**	TO	TO	TO	ERR
		3	V	0.21	**0.15**	ERR	Z3-FA	FA	NA
ex8	20	def	F	**0.13**	0.15	TO	TO	TO	ERR
		3	F	**0.14**	**0.14**	FA	ERR	ERR	NA
ex9	43	def	V	1.34	**0.85**	TO	TO	TO	ERR
		1024	V	TO	TO	ERR	Z3-TO	TO	NA
		3	V	2.93	**0.62**	ERR	Z3-FA	FA	NA
ex10	72	def	F	1.32	0.59	TO	TO	TO	**0.03**
		17	F	TO	10.47	**0.31**	UB	UB	NA
		3	F	4.02	1.14	**0.13**	UB	UB	NA
ex11	25	def	V	FA	FA	TO	TO	TO	TO
		3	V	**0.05**	0.08	0.06*	0.06*	0.06*	NA
ex12	24	def	F	0.12	0.16	0.12	0.10	0.10	**0.03**
ex13	10	-	F	**0.03**	0.44	0.07	0.06	0.13	TO
ex14	16	def	V	0.10	0.13	0.08*	0.08*	0.08*	**0.03**
ex15	35	-	V	0.56	0.34	FA	Z3	0.09	**0.03**
ex16	35	def	U	**0.09F**	0.10F	TO	TO	TO	TO
		2	U	**0.08F**	0.09F	**0.08*V**	**0.08*V**	**0.08*V**	NA
ex17	45	def	V	1.56	0.68	0.34*	0.24*	0.24*	**0.02**
ex18	30	def	V	FA	FA	TO	TO	TO	ERR
		100	V	TO	TO	ERR	ERR	ERR	NA
		10	V	**1.30**	3.0	ERR	ERR	ERR	NA
ex19	29	def	V	FA	FA	TO	TO	TO	TO
		3	V	0.14	**0.08**	0.10	**0.08**	0.09	NA
ex20	33	def	F	FA	FA	TO	TO	TO	**0.12**
		1024	F	455	TO	40.98	**40.0**	206	NA
		3	F	0.21	0.32	0.25	**0.09**	0.11	NA
ex21	26	def	V	0.45	0.36	FA	Z3	1.68	**0.02**
ex22	39	def	V	12.22	4.1	0.64	Z3	0.81	**0.06**
ex23	26	def	V	FA	FA	16.49	**0.16**	0.18	0.69
		36	V	25.14	6.46	16.49	**0.16**	0.18	NA
ex25	27	def	V	**0.26**	0.27	TO	TO	TO	TO
		3	V	0.21	0.20	0.10*	**0.08***	**0.08***	NA
ex26	30	def	F	1.88	**0.62**	6.42	Z3	4.79	UB
ex27	40	def	F	25.34	5.28	3.40	Z3	3.24	**0.09**
ex30	44	def	F	**0.15**	0.24	TO	TO	TO	ERR

Continued on next page

Table 2 – continued from previous page

bnc.	#L	#UNW	F / V	LAV B	LAV Z3	CBMC	ESBMC B	ESBMC Z3	KLEE
		100	F	TO	TO	ERR	Z3-UB	UB	NA
ex31	14	def	V	FA	FA	TO	0.08*	0.08*	**0.02**
		7	V	1.38	5.62	0.57	**0.08***	**0.08***	NA
ex32	27	def	V	0.78	0.5	2.30*	Z3	4.11	**0.18**
ex34	25	-	V	**0.08**	0.24	0.10	0.12	0.14	0.16
ex37	30	-	F	**0.16**	0.20	FA	Z3-UB	UB	ERR
ex39	27	def	F	**0.06**	0.08	TO	TO	TO	ERR
		3	F	**0.07**	**0.07**	UB	0.09	0.09	NA
ex40	20	def	V	0.09	0.12	TO	TO	TO	**0.02**
		3	V	**0.08**	0.10	0.12	**0.08**	**0.08**	NA
ex41	23	def	F	0.25	**0.25**	TO	TO	TO	17
		3	F	0.49	0.44	0.25	**0.07**	0.10	NA
ex43	113	def	F	28.56	17.91	FA	Z3	25.15	**0.06**
ex46	38	def	F	6.57	**5.75**	TO	TO	TO	ERR
		2	F	**37.43**	TO	FA	Z3-FA	FA	NA
ex47	35	def	F	3.71	**2.32**	TO	TO	TO	ERR
		2	F	4.40	**1.38**	FA	Z3-FA	FA	NA
ex49	16	def	F	0.18	**0.11**	TO	TO	TO	TO
		3	V	**0.06**	0.08	0.08	0.07	0.08	NA
inf1	36	-	F	**0.12**	0.22	0.18	UB	0.15	0.13
inf2	63	-	F	4.84	**1.25**	FA	Z3-FA	FA	UB
inf4	62	-	F	0.23	0.38	**0.11**	0.12	0.19	0.40
inf5	62	-	F	0.09	0.15	0.11	0.12	0.15	**0.06**
inf6	43	-	V	0.29	0.12	0.41	Z3	0.40	**0.06**
inf8	44	-	V	0.16	0.19	0.11	FA	0.12	**0.06**

Table 2: Experimental results.

Analysis of Results. False alarms and bugs undiscovered by CBMC can be explained by the way it models memory and control-flow of the programs. For example, CBMC assumes that each dynamic memory allocation will succeed, although this is not valid assumption. Also, CBMC does not precisely model the memory assigned to global arrays and pointers to pointers so this explains some false alarms. Concerning control-flow, imprecisions may arise after loop unwinding. If CBMC cannot prove that the unwinding is performed for the upper loop bound, it dismisses all current information about memory allocations for arrays, no matter if these allocations were static or dynamic. Since CBMC reports only flawed commands (and not commands that cannot prove to be safe), this may lead to undiscovered bugs. CBMC does not check/report unreachable bugs. Therefore, it can miss a bug if it is unreachable for all CBMC-feasible unwinding parameters.[9]

ESBMC inherits program modeling of CBMC, but also introduces some improvements. ESBMC models memory more precisely and it exhibits less false alarms than CBMC. It models that dynamically allocation may not succeed, but it still may miss some NULL-dereferencing bugs. Concerning solvers, it seems that ESBMC does not have support for Ackermannization and it calls the Z3

[9] For instance, in example 39.c, CBMC encounters time out for its default parameters, and for a small number of loop unwindings it does not discover the bug. The bug in this example, which is reachable as a consequence of a possible integer overflow, is discovered by LAV even for a small number of loop unwindings because LAV finds that the command itself is flawed so its reachability is not further checked.

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

Table 3. Summary

solver instead of Boolector whenever the theory of uninterpreted functions is involved (Boolector does not support the theory of uninterpreted functions). Also, it is likely that there are some errors in solver interfaces since ESBMC can give different results when different solvers for the same theory are used for the same problem. ESBMC exhibits the largest number of timeouts and the largest number of undiscovered bugs. ESBMC with its default parameters was not best for any benchmark time, but it has the largest number of best times when the upper loop bound was specified.

The usage of KLEE is somewhat different than the usage of LAV, CBMC and ESBMC. Unlike these tools, for KLEE it is necessary to annotate programs with claims which variables should be treated as symbolic. It is not possible to have dynamic memory allocation with a size represented by a symbolic value, so KLEE reports error messages for some benchmarks. Also, it is not possible to simulate nondeterministic choice as a loop entry parameter. KLEE does not terminate when it finds a first error (as other tools do) and there is no option to do so. However, this behavior does not affect the best times reported in the table. As a symbolic execution tool, the number of loop unwindings cannot be specified to KLEE, but the number of states considered can. Since this is not comparable to number of loop unwindings, we compare KLEE to other tools only with its default parameters. KLEE had six time outs, ten errors, two undiscovered bugs and no false alarms. It has the largest number of best times. KLEE was the most efficient on examples where there is only one possible path through the program, because it efficiently finds it and the symbolic execution in these cases take almost the same as a real execution. Other tools, because of the different nature of modeling, do not take the advantage of having just one path through the code. However, in practical applications, this is rarely the case.

LAV has no timeouts for its default parameters. This comes with a price of the biggest number of false alarms for default parameters. These false alarms are due to the policy of LAV that reports all commands that are potentially unsafe (that could not be proved to be safe). So, all the false alarms come with a message that the command is potentially unsafe, and never with a message that the command is flawed, which makes the difference to tools that have no ranking of potential bugs. If we change this policy, and if LAV reports only commands which are proved to be flawed, then LAV would not have so many false alarms but would have undiscovered bugs. In more than half cases when LAV reported false alarm, the other tools had timeouts, therefore, neither tool exhibited desired behavior. Concerning timeouts, most of the timeouts that LAV exhibited were due to the high loop unwinding parameter. In most cases, the default parameters already gave good results so there was no need for the unwinding with the upper loop bound. LAV has no error messages, undiscovered bugs and has no false alarms for a fixed number of unwindings. If we compare Boolector and Z3, we can see that efficiency of LAV with Boolector is very similar to the efficiency obtained with Z3, except that there are two cases when Z3 encountered timeout when Boolector did not and only one case when Boolector encountered timeout and Z3 did not.

We believe that these results present a useful experimental comparison of existing tools. They also suggest that LAV is an interesting point in the design space of verification and bug finding tools.

9 Application in Education

Software verification tools have different areas of application. One typical area of application are safety-critical computer programs. On the other hand, verification tools can be very beneficial in checking computer programs that are far from being safety-critical but are massive in number and have other nature of importance. In this section we consider one such application: computer programs developed by students within programming courses in high schools and universities. A tool that could help students and teachers to notice errors in programs would have multiple benefits. For students, such tool would be helpful when there is no teacher to check the solution (which is, most of the time, the case). For teachers, such tool would be helpful in marking exams, at least for pointing to standard errors. For both, such tool would be illustrative and would demonstrate power of verification tools, to which students should get accustomed and ready to adopt in their professional work.

With this motivation in mind, we performed another set of experiments with our tool—we analyzed programs written by students that took an introductory C programming course at the University of Belgrade. Our corpus consists of 157 programs which were written by the students at test exams along the course.[10] We divided the corpus into three groups. The first group consists of solutions of problems that involve numerical calculations and manipulation of command line arguments. The second group consists of solutions of problems that involve manipulation with arrays and matrices. The third group consists of solutions of problems that involve manipulation of strings and data structures. LAV was set to use its default parameters and the time that LAV spent in analyzing the programs was typically negligible. Some of the programs from the corpus did not meet the given specification, but we considered only bugs and not functional correctness. LAV discovered 423 genuine bugs in 121 programs and had 32 false alarms in 8 programs.

[10] The corpus can be found at the LAV web page.

The results of our experiments are summarized in Table 4. In the first two groups, the largest number of bugs were possible buffer overflows (225 bugs were discovered in 81 programs), while the next most frequent bug was division by zero (22 bugs in 22 programs). In the third group, the largest number of bugs were possible null pointer dereferencing (46 bugs in 15 programs), while the next most frequent bug was buffer overflow (30 bugs in 15 programs).

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

Table 4. Application in education: the table contains the number of solutions considered for the given problem, the average number of lines per solution, the average number of reported bugs per solution, and the average number of false alarms per solution.

The vast majority of bugs that students produced follow wrong expectations — for instance, expectations that input parameters of their programs will meet certain constraints and that memory allocation will always succeed. In many cases, omission of a necessary check (e.g. whether an input parameter meets the constraints) produces several bugs in the rest of the program. This explains the large number of bugs in the corpus — adding only one check in a program would typically eliminate several bugs. Apart from these sources of bugs, there were just a few bugs with other origin (such as uninitialized variables or accessing memory that was not allocated). Concerning false alarms, all false alarms were consequences of overapproximations of loops or of the absence of support within LAV for some functions from the standard library. It was beyond the scope of this experiment to manually annotate all existing bugs so we cannot report on the number of bugs that LAV missed to report.

A simplified example of a program from the corpus is given in Figure 9. This example illustrates several typical students' bugs: two possible buffer overflows (lines 18 and 19) and one division by zero (line 12). Models generated by LAV (given at the right-hand side of the figure) can help in correcting these bugs. Although these are only preliminary experiments, the experience suggest that LAV can be useful in everyday practice of an introductory programming course.

10 Conclusions and Further Work

We presented a new software verification approach and a corresponding tool, LAV, for bug finding and for checking correctness conditions. LAV uses the compiler intermediate language, LLVM, code. Therefore, LAV need not deal

```
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
```

Fig. 2. A simplified version of a program from the corpus (shown on the left) and the LAV output (shown on the right). The shown output for this program is generated by invoking LAV with default parameters, so the loop in line 6 is over-approximated. The output states that lines 12, 18 and 19 are unsafe in general, and that the line 12 is unsafe when the function `get_digit` is called from the line 20. LAV also shows the nature of a possible error and the values of relevant variables.

with the specifics of C, and can be used for analysis of programs in several programming languages. In addition, the approach can be used with any similar low-level code representation. The approach combines symbolic execution, SAT encoding of program's behavior and bounded model checking. Individual blocks of the code are modeled by first-order logic formulas constructed by symbolic execution, and relationships between blocks are modeled by propositional formulas. Formulas that describe blocks' behavior are combined with correctness conditions for individual commands to produce correctness conditions of the program to be verified. These conditions are passed on to an SMT solver covering a suitable combination of theories. The proposed approach is implemented as an open-source tool LAV. Currently, several SMT solvers (Boolector, MathSAT, Yices, and Z3) are supported. Our experiments suggest that the presented approach is competitive with related tools. We believe that our approach can be a useful component of tools that combine multiple analysis and model checking approaches (as suggested by recent tools [3]).

In the future we plan to further improve the modeling power and efficiency of the tool. We plan to modify the implementation to use multi-core processor design. We also plan to improve our interprocedural analysis and the robustness of the tool. We are interested in further experiments with other benchmark suites (such as, for example, [22] and [18]). We plan to make a number of extensions, such as an improved user interface using a web client, more descriptive bug explanations, and automated test-case generation. We expect that these features will make LAV even more applicable in education and practice.

References

1. W. Ackermann. *Solvable cases of the decision problem.* North Holland, 1954.
2. D. Babić and A. J. Hu. Calysto: Scalable and Precise Extended Static Checking. In *ICSE'08*, pages 211–220. ACM, May 2008.
3. G. Balakrishnan, M. K. Ganai, A. Gupta, F. Ivancic, V. Kahlon, W. Li, N. Maeda, N. Papakonstantinou, S. Sankaranarayanan, N. Sinha, and C. Wang. Scalable and precise program analysis at nec. In *FMCAD*, 2010.
4. M. Barnett and K. R. M. Leino. Weakest-precondition of unstructured programs. *ACM Sigsoft Software Engineering Notes*, 31:82–87, 2006.
5. C. Barrett, R. Sebastiani, S. A. Seshia, and C. Tinelli. Satisfiability modulo theories. In *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, pages 825–885. IOS Press, 2009.
6. A. Biere, A. Cimatti, E. Clarke, O. Strichman, and Y. Zhu. Bounded model checking. *Advances in Computers*, 58, 2003.
7. R. Brummayer and A. Biere. Boolector: An efficient SMT solver for bit-vectors and arrays. In *TACAS*, volume 5505 of *LNCS*. Springer, 2009.
8. R. Bruttomesso, A. Cimatti, A. Franzén, A. Griggio, A. Santuari, and R. Sebastiani. To ackermannize or not to ackermannize? In *LPAR 2006*, volume 4246 of *LNCS*, pages 557–571. Springer, 2006.
9. R. Bruttomesso, A. Cimatti, A. Franzén, A. Griggio, and R. Sebastiani. The MathSAT 4 SMT solver. In *CAV*, volume 5123 of *LNCS*, pages 299–303. 2008.
10. C. Cadar, D. Dunbar, and D. Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, 2008.
11. V. Chipounov, V. Kuznetsov, and G. Candea. S2e: a platform for in-vivo multi-path analysis of software systems. *SIGARCH Comput. Archit. News*, 39, 2011.
12. E. Clarke, D. Kroening, and F. Lerda. A tool for checking ansi-c programs. In *In TACAS*, pages 168–176. Springer, 2004.
13. L. Cordeiro, B. Fischer, and J. Marques-Silva. SMT-based bounded model checking for embedded ansi-c software. *In ASE*, 0:137–148, 2009.
14. L. De Moura and N. Bjorner. Z3: An Efficient SMT Solver. In *TACAS*, pages 337–340, 2008.
15. B. Dutertre and L. de Moura. The Yices SMT solver. Tool paper at http://-yices.csl.sri.com/tool-paper.pdf, August 2006.
16. C. Flanagan, J.B. Saxe. Avoiding exponential explosion: generating compact verification conditions. In *Proc. ACM SIGPLAN POPL*, January 2001.
17. J. C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.
18. K. Ku, T. E. Hart, M. Chechik, and D. Lie. A buffer overflow benchmark for software model checkers. In *Proceedings of ASE '07*. ACM, 2007.
19. S. Sankaranarrayanan. Necla static analysis benchmarks, http://www.nec-labs.com/research/system, 2009.
20. C. Sinz, S. Falke, and F. Merz. The low-level bounded model checker llbmc: A precise memory model for llbmc. In *SSV*, 2010.
21. N. Tillmann and J. Halleux. Pex white box test generation for .net. In *Proc. of TAP 2008*, volume 4966 of *LNCS*, pages 134–153. Springer, 2008.
22. M. Zitser, R. Lippmann, and T. Leek. Testing static analysis tools using exploitable buffer overflows from open source code. *SIGSOFT Softw. Eng. Notes*, 29, 2004.