MLSA: a static bugs analysis tool based on LLVM IR

Hongliang Liang[1], Lei Wang[1], Dongyang Wu[1], Jiuyun Xu[2]

[1]School of Computer Science
Beijing University of Posts and Telecommunications
Beijing, China
E-mail: hliang@bupt.edu.cn, wangcppclei@gmail.com, dongyangwu@bupt.edu.cn

[2]School of Computer & Communication Engineering
China University of Petroleum
Qingdao, China
E-mail: jyxu@upc.edu.cn

Abstract

Program bugs may result in unexpected software error, crash or serious security attack. Static program analysis is one of the most common methods to find program bugs. In this paper we present MLSA -- a static analysis tool based on LLVM Intermediate Representation (IR), which can analyze programs written in multiple programming languages. MLSA combines symbolic execution with Z3 SMT solver to find bugs. At present, MLSA can detect some kinds of bugs, such as divide zero error, pointer overflow and dead code. Moreover, as a framework, MLSA follows the scalability and extensibility principles, which can help detect other types of bugs. Experiments show that MLSA is effective in finding bugs in real world software.

Key words: symbolic execution; bug detection; static program analysis; LLVM IR; SMT solver

1. Introduction

Program analysis technology has been proposed to detect bugs in software. Based on whether the target program will be running, program analysis can be divided into dynamic program analysis and static analysis. Dynamic analysis can detect bugs through instrumenting analysis instructions into target program or simulating the running status of the target programs in virtual machine environment. Contrary to the dynamic analysis, static program analysis technique mainly focus on the source code of target program or intermediate language compiled by the compiler frontend, which needs no running status. Nowadays, main static program analysis methods consist of control flow analysis, data flow analysis, model checking, taint analysis, symbolic execution, etc.

Programs are composed of instructions, and an instruction is composed of operator and operand. 1) Control flow analysis is based on the operator of program instructions. According to the operator in the instruction, all programs can be represented by three structures, including sequential structure, branch structure and cycle structure of basic blocks, and each basic block represents as a unit which includes instructions, single entry and single exit. 2) Data flow analysis concerns the operands more instead of the operators, which analyzes the data flow of the program execution path and the possible values of each variable. It can be divided into flow insensitive analysis, flow sensitive analysis and path sensitive analysis according to analysis precision. 3) Model checking will firstly construct the bugs' automaton from the program bugs patterns, and then record the behaviors of the program using the language of automation which are used to feed the automaton. At last, the automaton will decide whether the behavior of the program is a defect. 4) Taint analysis is a form of data flow analysis. It concerns whether the taint information in the program will spread to the key function and operation. 5) Symbolic Execution is an advanced analysis technique, which can be used in dynamic analysis and static analysis. Combined with control flow analysis and data flow analysis, symbolic execution records the variable value as a symbol, and decide whether a execute path can be run according to the variable value and control flow graph. Hence, time efficiency and space efficiency will greatly reduce. Symbol execution technique is path sensitive.

MLSA uses the symbolic execution technique for its various advantages [1], such as higher accuracy. Symbolic execution is often combined with SMT (Satisfiability Modulo Theories), which is used to calculate the satisfiability under given conditions. We use the Z3 SMT solver [2] for its open source and efficient features.

LLVM is a compiler framework with a set of reusable libraries and well-defined interface [3]. Now it is official compiler of Apple Inc. Clang is its main frontend for C/C++/Objective C languages. LLVM also supports many other program languages and backend platform, resulted from its Intermediate Representation (IR). LLVM IR is generated by compiler frontend from different languages, and can be used to analyze, optimize and transform to executable file. Therefore, MLSA is designed and built on it to analyze the programs written with multiple programming languages.

Program bugs may cause program crash, runtime error or even incur seriously security accident. Since there are too many different kinds of bugs to detect, many existing research works either focus on one type of bugs and do deep research, or try to detect more kinds of bugs to their best.

In this paper we present MLSA, which uses symbolic execution technique based on LLVM IR to find bugs. The contributions in this paper are as follows:

1) We present a static analysis tool MLSA, which is built on LLVM IR to support multiple programming languages and can now detect three types kinds of program bugs, such as, division zero error, pointer overflow, dead code. MLSA uses symbolic execution technique to get more accuracy, and is optimized to mitigate path explosion.

2) MLSA is evaluated with several real world software to examine its abilities in finding program bugs. We test MLSA on GNU Coreutils suite and find a pointer overflow bug. Moreover, the experiments also show MLSA's analysis abilities on other program languages such as Fortran.

In the next section we describe the overview of MLSA. Then we introduce the design and implement of the system in section 3. Section 4 shows the experiments. We describe and compare with related work in section 5. In section 6 we conclude the paper.

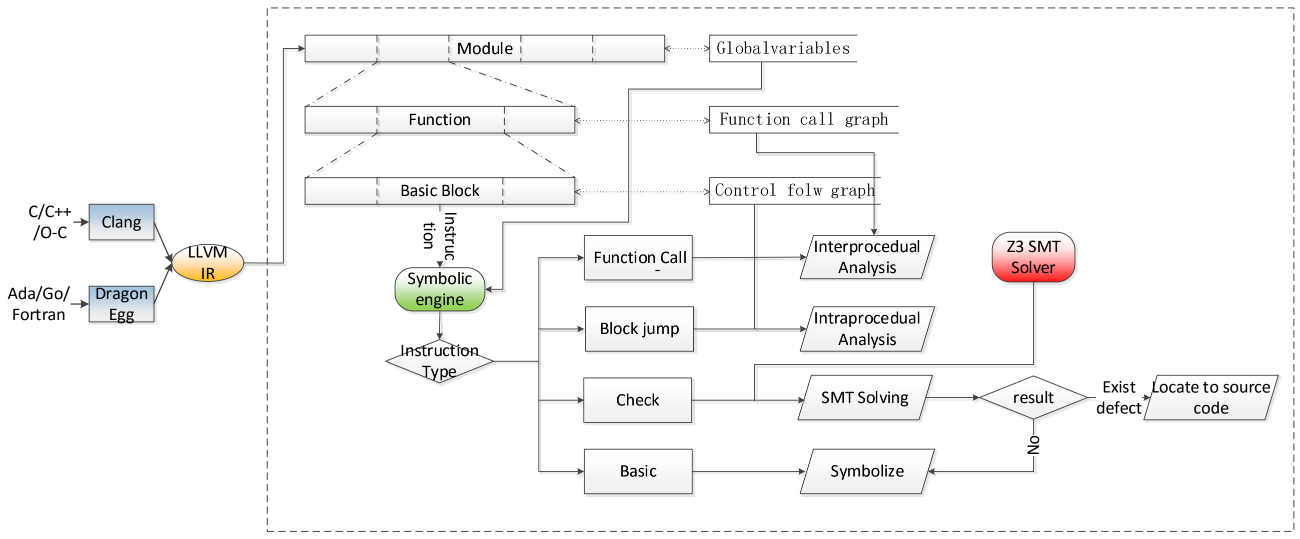

Figure 1 Architecture of MLSA

2. Overview

In this chapter we describe the overview of MLSA system, the architecture of MLSA is shown in Figure 1. The input of MLSA is LLVM IR file, our framework firstly use Clang, a famous compiler of LLVM for C/C++/Object-C program language, to compile source codes into LLVM IR. And in order to support other program language, such as Go, Fortran, Ada, our framework also contains DragonEgg compiler. MLSA can automatically choose the suitable compiler according to the program language of the target program.

After that we get a module which is the intermediate representation of source code. Figure 2 shows the structure of LLVM IR module. A module includes global variables which are shared with all functions, and many functions. A function includes some basic blocks which are single entry and single exit, and a basic block includes some instructions. There are two kinds of forms exist with LLVM IR, one is LLVM compiler can read from storage to memory and the other is human readable, they can transform with each other. We give a human readable LLVM IR example in Figure 3, there are a global variable @a, @ indicate variable a is global, and a function @_Z4mainv(), in this function we define some local variables %b, %c, %d and do some calculations such as addition, division and assignment between them.

MLSA processes the instructions in the module according to the control flow graph and the relationship between basic blocks. Many static analysis technologies can be used to analyze this module, and we use symbolic execution technique and Z3 SMT solver to assist with it in this paper.

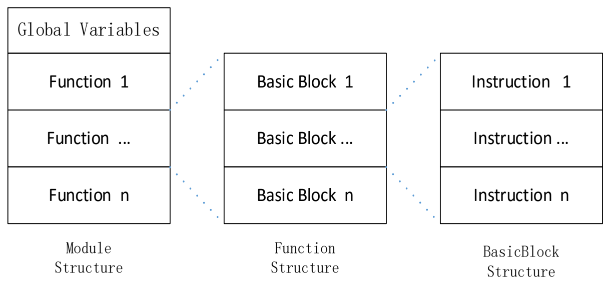

Figure 2 LLVM IR's structure

Symbolic execution is a common technique which can be used in both static and dynamic program analysis. It records variables with symbol value but not real value in static analysis. We introduce the fundamental in Figure 4 and Figure 5. Figure 4 is the pseudo code of a symbolic execution example and Figure 5 is the program flow graph. The symbolic execution works as Figure 5, according the path condition and current states to pend which path can be reached, for example in Figure 5, the initial value of variable a is unknown, there are two branches a>0 and a<=0 which may be true under certain conditions. When a>0, a>0 is the path condition of its sub-paths, then symbolic engine will check current states(a>-1) whether be true under the condition of path condition is a>0, the symbolic execution engine will know the path① will be executed, and path② will not be executed(a>0 && a<=-1). In the same way the path③ and path④ both can be executed in some case according to the symbolic engine.

```
void  SymbolExecute( int a) {
    if ( a > 0 ) {
        if ( a > -1 ) {
            ① ;
        }
        else {
            ② ;
        } }
    else {
        if ( a < -1 ) {
            ③;
        }
        else {
            ④;
        } } }
```

Figure 3 LLVM IR Example

```
@a = global i32 0, align 4
define i32 @_Z4mainv() #0 {
entry:
  %b = alloca i32, align 4
  %c = alloca i32, align 4
  %d = alloca i32, align 4
            store i32 10, i32* %a, align 4      # a=10
    store i32 20, i32* %b, align 4      # b=20
  %0 = load i32* %a, align 4
  %1 = load i32* %b, align 4
  %add = add nsw i32 %0, %1        # a+b
    store i32 %add, i32* %c, align 4   # c=a+b
  %2 = load i32* %b, align 4
  %3 = load i32* %a, align 4
  %div = sdiv i32 %2, %3           # b/a
  store i32 %div, i32* %d, align 4
  ret i32 0
}
```

Figure 4 Pseudo code of symbolic execution

Symbolic engine only records the symbol value of variables, SMT solver is used to decide the whether satisfied under current condition. SMT solver's inputs are all conditions and judgment included. SMT solver's outputs only can be 'sat' 'unsat' and 'unknown', symbol execution engine decides whether the condition is satisfied according to the result from SMT solver.

When finding a bug in the target program, MLSA will record the location information of the bug in the source codes so that analyzers can find the bug quickly. MLSA records these errors information in a log file.

3. Design & implementation

In this section we describe the design and implementation of our system.

MLSA takes LLVM IR files as input. Its architecture is shown in Figure 3. As shown in Figure 3, firstly, different compiler frontends are used to translate different program language codes to the common LLVM IR files. Secondly, MLSA processes these IR files according to the module structure, the module level, the function level, the basic block level and the instruction level. A module is an input file, which contains some global variables and some functions information. At this level we construct the function call graph. A function contains some basic blocks, we can construct the control flow graph (CFG) in every function, and we can know the jump relation between basic blocks based on the generated CFG. Since a basic block contains some instructions we deal with the instructions with the symbolic execution engine.

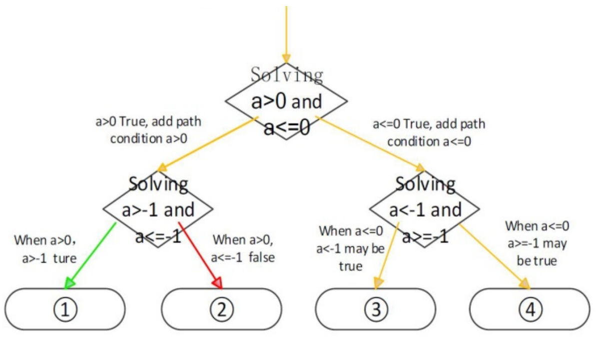
Solving
a>0 and
a<=0
a>0 True, add path condition a>0
a<=0 True, add path condition a<=0
Solving
a>-1 and
a<=-1
Solving
a<-1 and
a>=-1
When a>0, a>-1 ture
When a>0, a<=-1 false
When a<=0 a<-1 may be true
When a<=0 a>=-1 may be true
①
②
③
④

Figure 5 Symbolic Execution example

We divide the design and implementation of MLSA as five steps: 1) the preprocess, compile source code to LLVM IR; 2) the process of global variables, the construction of function call graph and control flow graph; 3) the symbolic execution, including symbolic of the basic variables, the intraprocedural analysis in a function, and the interprocedural analysis between functions; 4) the SMT solver process, including the transform from the symbol value to SMT input format and push constraint condition to solver, and yield result from solver; 5) checking whether bugs exist according to the return value from SMT solver, if so it will locate the bug in source codes.

3.1 Preprocess. This step is to compile source codes to LLVM IR, as shown in Figure 3, there are two ways to accomplish this, one is to use clang, and the other is to use DragonEgg [4] which is a GCC plugin from LLVM project.

3.2 Constructing function graphs and symbolizing global variables. After being compiled, there is a module file, as the input of MLSA, which includes some global variables and functions. Because MLSA strictly follows the function call relationship and CFG, and global variables are shared with all functions, firstly what we need to do is symbolize the global variables and store them into first block ('entry' block mostly) of entry function ('main' function mostly).

Function call relationship is the basis of interprocedural analysis, which can improve the analysis precision. We design two containers to store the information about function-call relationship, both of them are key-value pairs. One stores <Function Name, Function Info.>, the other stores <Caller, Callee>. All function-call relationships will store in this way.

Intraprocedural analysis is a program analysis technique, which only concerns variables and relation in one function. Usually, one function contains some basic blocks, and a basic block is a sing entry sing exit instructions sequence. The CFG plays an important role in intraprocedural analysis which records the jump between basic blocks. Similar with the way to store function-call graph, we construct CFG by storing the basic block in the <Basic block Name, Basic block Info.> key-value pair, and storing the basic block jump in <Source block, Destination block> key-value pair. All these <Source block, Destination block> key-value pairs composed the CFG.

3.3 Symbolic executing. After constructing function-call relationship graph and control flow graph, MLSA processes the instructions in basic blocks. According to role of different instructions, we divide all instructions into four parts: basic instructions, checking instructions, function-call instructions, and jump instructions between blocks.

Most instructions are basic instructions, including alloca, store, add, etc. MLSA need symbolize the variables of these instructions, for example, '%temp = alloca i32, align 4' will allocate 32-bit variable, named 'temp', 'store i32 1, i32* @m, align 4' will store value 1 to variable m, '%add = add nsw i32 %0, %1' will add variable 0 and variable 1, and assign the result to variable add. For general variables, which has constant value during the compilation phase, MLSA stores the variable with the value, otherwise stores the variable with symbol. For pointers, MLSA need store more information about the left and right boundary of these pointers, besides the value or symbol of the variable, and the boundary is stored in similar way with variable. Table 1 shows the detail of how to deal with variables and pointers.

Table 1. Symbolizing variables and pointers

Source Code	**Symbol**		**Relation**	**Detecting Condition**
	a	**b**		
Int a,b;	a0	b0		
a = 3;	3	b0	a==3	
b += a;	3	3+b0	b==a+b	
c = a/b;				b == 0
Int a[10],* b; int i	a0	b0	left(a)=0∧ right(a)=10	
b = a+3;	a0	a0+3	left(b)=3∧ right(b)=7	left(b-3)>=0∧ right(b-3)<10
*（b+i）= 10;				left(b+i)>=0∧ right(b+i)<10

We use two containers to store the information of variables, one (Symbol in Table 1) of which stores every variable's information, the other (Relation in Table 1) stores the relationship of variables (equality, add, etc.).In Table 1 the Store is represents the symbol of variables, the Relationship represents the relation between variables, and the Detecting Condition represent the conditions MLSA need to check. We first define two variables a and b, their value is unknown, so we store them with the symbol a0 and b0, then for 'b += a', we store the relation to the Relation, and update the Store of b with 'a0+b0'. Continue to assign variable 'a' with a value, the Store of b is stored with a value. In the division, it need to detect b whether can be zero. For pointer, we add the left boundary and right boundary, when access these pointers it must check whether the access addresses are in the scope of these pointers.

As described above, some instructions are checking instructions, which are chose depending on what bugs we need check. For example, MLSA check the divide zero error, the instruction 'sdiv' or 'udiv' will regard as check instruction, for the pointer overflow, the instruction 'getelementptr' will be regard as checking instruction. MLSA can extend the kinds of bugs by adding some new check instructions.

Interprocedural analysis is relied on the jump instructions at end of every basic block. It's of great significance to process the jump instruction which concerns the analytical precision. MLSA uses symbolic execution to decide whether the next path can be reached, if that MLSA will copy the containers of current block to the next block so that the symbol information delivers as the CFG. There is a serious problem in interprocedural analysis that the CFG may have loops which can result in the CPU and memory source will be exhausted if MLSA don't process them. MLSA counts the run time of every basic block, when one's runs times exceed the time limit which is setting by system default, MLSA will force to skip this block. That's because when a block run enough times there is no need to redo it. Sometimes there are many branches in the jump instruction, MLSA employ the pattern of depth first travel.

Intraprocedural analysis is also relied on the function-call instructions (such as 'call', 'invoke' in LLVM IR). We adopt a similar approach with interprocedural analysis, when a call instruction running, MLSA copies the container of current block to the entry block of callee function, after the callee function runs over, updating the current block's container of caller function with the last block's container of callee function. While different variables from different functions may have the same name, MLSA names every variable prefixing with the name of relative function.

3.4 Constraints solving with SMT solver. Checking instructions symbolize the value of the variable in it, add appropriate assert to the Relation container. MLSA must transform the relations in the Relation container to the Z3 input format before submitting to the Z3 solver. We construct an expression queue, and then push the assert expression into the queue, recursively push the expression according to the relation in the container from bottom to top while popping the expression and transforming this expression to context of solver. By this way we can add all constraints to Z3 SMT solver.

MLSA is designed to check three kinds of bugs: for division zero error, MLSA check two asserts ‘x==0’ and ‘x!=0’, if ‘x==0’ and ‘x!=0’ both are true, it dedicate this may be a division zero error, if ‘x==0’ is true, and ‘x!=0 is false, it dedicate that this is a division zero error actually, in other case there is no division error. For pointer overflow and dead code MLSA using the same approach above, the different compared with division zero error is that MLSA checks the boundary of pointer and path reachability of dead code.

3.5 Recording the location of bugs. When finding a program bug, MLSA will record the location of the bug in source codes. By compiling the source codes with ‘-g’, compiler frontend will add debug information to LLVM IR. According to the debug information, MLSA can extract the file name and line number of certain a bug.

We implemented MLSA with the form of LLVM pass on the platform of Unbuntu14.04 and LLVM 3.6.2, MLSA contains over 5000 lines of C++ codes. Now it is designed to check three kinds of bugs, every of which represents a class of bugs and can be extended easily.

4. Evaluation

In this section, we test MLSA on several programs of C and Fortran language, then we check GNU Coreutils [5] with MLSA, at last we compare MLSA with other related tool using NECLA static analysis benchmark [6].

At first we test MLSA with simple test cases to show the ability of MLSA, there are three kinds of bugs we need to check, we give the pointer overflow only here. Figure 6 is a C program example of pointer overflow and MLSA’s check result followed. At line 8 the program dynamically allocates a memory of 40 bytes, then assign to the array of different index at line 9 and 10.The error messages illustrate the reason of this error, said the boundary of pointer is [0,10), but the current index is 10, exceed the boundary of pointer. For other two kinds of bugs, we also test them using MLSA, the result show that MLSA has the ability to check these bugs. Due to space limitations we are not show the details here.

```
1  voidsetvalue(int p[], intpos, intval)
2  {
3          p[pos] = value;
4  }
5
6  int main()
7  {
8          int *p = new int[10];
9          setvalue(p, 1, 10);
10         setvalue(p, 10, 20)
11           return 0;
12 }

Error：
Process GetElementPtr...
    FileName:testdivzero.cpp Line:3  Maybe an overflow error! The left_boundary is 0 and the right_boundary is 10. The position is 10.
```

Figure 6 Pointer overflow example

```
 1 program main
  2  integer i
  3  integer m
  4  integer n
  5  i = 1
  6  m = 2
  7  n = m / (i-1)
  8  print *,i,m,n
  9 endprogram
Error：
Process SDiv...
FileName: Line:0  Maybe a divide zero error!
```

Figure 7 Fortran division zero example

```
1832    staticboolparse_obsolete_option(intargc, char * const *argv, uintmax_t * n_units)
1833  {
1834        const char * p;
                        …
1845        if(!(argc  ==  2)  ||(argc==3)&&(argv[2][0]=='-'&&argv[2][1]) …)
1848                  return false;
1851        p = argv[1];
1853        switch(*p++) {…}
1879        while ( ISDIGIT(*p) )
1880        p++;
}
Error：
FileName:/coreutils/coreutils-8.20/src:tailc.c  Line:1853  Maybe an overflow error.
```

Figure 8 A bug in tail.c found by MLSA

Figure 7 is a program written by Fortran language. Before MLSA begins to analyze, it should be compiled as the way mentioned in Section 3. There is a division zero error in the program, and MLSA detected the bug. It shows MLSA has the ability to analyze different source language. MLSA currently can't locate this error to source code because DragonEgg can't completely support Fortran language well. To support more program language (such as ada, go) may be another work we can do in the future.

To evaluate its ability to analyze the real world software, MLSA is used to analyze the Coreutils package which consists of common used programs in Linux/Unix such as ls, tail, etc. We downloaded a stable version of GNU Coreutils (v8.20), and compiled 96 tools successfully. Most of these tools consist of 300~2500 lines of codes. Experiments were performed on a system running Ubuntu 14.04, with Intel I5 processor and 4G memory. It took five minutes to complete the analysis.

MLSA found a bug in tail program; we report it to the official who confirmed that it is a bug though there is currently any serious problem due to it. Figure 8 shows the detail of the defect. The parse_obsolete_option function has three parameters, argc and argv are both from main function directly, line 1834 define a pointer, line 1845 filter the argument of argc and argv, we only concern the condition when argc == 2, line 1851 assign argv[1] to pointer p, line 1853 pointer p add self and get the value of p point, line 1879 will get the value of point p's next address point, when argc==2 and the size of arg[1] is 1,try to access argv[1][1] is a pointer overflow error.

Table 2. Comparison of MLSA with LAV and KLEE

	MLSA	LAV	KLEE
Accuracy	63.0%	74.1%	55.6%
False negatives	14.8%	25.9%	0%
False positives	7.4%	0%	3.7%
Error rate	14.8%	0%	22.2%
Time out	0%	0%	18.5%

Besides the pointer overflow analysis, MLSA also do some statistics on the coverage of path. According to the analysis log 65 of 97 (67.0%) can fully cover, 6 of 97(6.2%) can't fully cover, and 26 of 97(26.1%) occur error. Due to diversity of CoreUtils, MLSA can reach 67.0% path coverage which is an average level of related work.

We also compared MLSA with some related tools, including LAV and Klee, using the NECLA [6] static analysis benchmark. We remove 30 of 57 of these benchmarks because LAV/Klee do not support them and a bug in LLVM compiler [7]. The compare result statistics are shown in Table 2. It seems that LAV has better experimental result, because NECLA has many complex calculations and LAV using first-order logic formula and propositional formulas can do better, while MLSA and Klee do not work well in this field. MLSA is better than Klee on the analysis time because MLSA is optimized to deal with loop processing.

MLSA is better than LAV in that MLSA can analyze real world software such as Coreutils suite. For comparison, we test LAV on the Coreutils following the LAV's documents, results show that LAV can only process four IR files of Coreutils (Coreutils 8.20 and compiled with LLVM 3.3) in our experiment. Furthermore, the LAV developers don't carry on the experiments on some real world big programs.

5. Related work

There are several research work based on LLVM IR, as well as a few work use symbolic execution technique so far. Klee [8] is a famous tool, which aims to get high-coverage test cases, used symbolic execution technique and STP solver. LLMBC [9] is a tool which is used the bounded model checking of C and C++ program language and based on the LLVM IR. LLMBC encodes LLVM IR to its own intermediate language (ILR) and adds many checks into it, then passes them to SMT solver to pend whether program exists bug or not. LAV [10] uses a more formalized method; it transforms LLVM IR into first-order logic formulas and propositional formulas, then passes these formulas to SMT solver. Others tools like KLOVER [11], CALYSTO [12] are static analysis tools related.

6. Conclusion

MLSA, a static analysis tool based on symbolic execution technique and LLVM IR, is presented. MLSA can now detect three kinds of program bugs, division zero error, pointer overflow and dead code, each of them represent one aspect, arithmetic operation, pointer analysis, symbolic execution. MLSA is evaluated not only on simple programs of C and Fortran program language, but also on many real world software such as those included in GNU Coreutils package. Compared with Klee and LAV, MLSA is extensible to analyze more kinds of bugs and is optimized to mitigate path explosion problem. In the future we will extend MLSA to analyze more kinds of bugs and evaluate MLSA on larger projects that is used in industry.

Acknowledgement

This work is partially supported by the National Science Foundation of China (NSFC) under grant No. 91418206.

References

[1] C Cadar, K Sen. Symbolic execution for software testing: three decades later. Communications of the ACM, 2013

[2] L De Moura, N Bjørner. Z3: An efficient SMT solver. Tools and Algorithms for the Construction and Analysis of Systems. 2008

[3] C Lattner, V Adve. LLVM: A compilation framework for lifelong program analysis & transformation. Code Generation and Optimization, 2004

[4] DragonEgg. http://dragonegg.llvm.org/

[5] GNU CoreUtils. http://www.gnu.org/software/coreutils/coreutils.html

[6] NECLA static analysis benchmark.http://www.nec-labs.com/

[7] LLVM Bug 21852. https://llvm.org/bugs/show_bug.cgi?id=21852

[8] C Cadar, D Dunbar, DR Engler. KLEE:Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. OSDI, 2008.

[9] C Sinz, F Merz, S Falke. LLBMC: A bounded model checker for LLVM’s intermediate representation. Tools and Algorithms for Construction and Analysis of Systems. 2012.

[10] M Vujošević-Janičić, V Kuncak. Development and evaluation of LAV: an SMT-based error finding platform. Verified Software: Theories, Tools, Experiments. 2012.

[11] G Li, I Ghosh, SP Rajan. KLOVER: A symbolic execution and automatic test generation tool for C++ programs. Computer Aided Verification, 2011.

[12] D Babic, AJ Hu. Calysto. Software Engineering, 2008. ICSE'08.