SESL: A Memory Safety Checker based on Separation Logic
(Competition Contribution)

Xie Li[1,2], Yutian Zhu[1,2], Zongxin Liu[1,2], Wanyun Su[1], Zhilin Wu[1], and Lijun Zhang[1,3]

[1] Institute of Software, China Academy of Sciences
[2] University of China Academy of Sciences
[3] Institute of Intelligent Software, Guangzhou

Abstract. SESL is a memory safety analyzer for C programs. It uses SMACK as the front end. Its back end relies on a symbolic execution engine which encodes path constraints as array separation logic (ASL) formulas. The ASL formulas are then solved by a separation logic solver called CompSpen. The salient feature of SESL is that it utilizes array separation logic to support byte-precise formal reasoning of memory safety issues.

Keywords: Memory safety · Symbolic execution · Separation logic.

1 Overview

SESL is a static memory safety analyzer for C programs. The basic methodology are inherited from the symbolic execution based on separation logic (SL)[1] and we use ASL to symbolically represent the a set of states. Compared to the original SL [3] [LX: add later variants :XL], ASL compacts a sequence of uninitialized points-to predicates into a single predicate `blk(x,y)`, indicating the heap is initialized from address `x` to `y-1`. With this succinctness and theoretical results of [LX: ASL solving paper :XL], SESL can deal with analysis and checking efficiently. The main contribution of this tool is the implementation of our symbolic execution engine based on ASL. With the help of existing frontend tool SMACK[2], we implemented a low level semantic of intermediate representation which supports byte-level manipulation of heap data. The algorithm of falsification utilizes the result of symbolic execution for later feasibility checking and error analysis using our ASL solver CompSpen[LX: CompSpen paper or website :XL].

The workflow of symbolic execution in SESL is stated as follows.

1. Construct a control flow graph (CFG) $\mathcal{G} = (V, E)$ after parsing and analyzing the input source code, where V and E are the finite set of states and edges. Each state can be regarded as a basic block.
2. Sample out a set S of paths π of $\mathcal{G}$. The finiteness of S is guranteed by restricting the maximum number that a state appears on one path.

3. For each path in S, do the symbolic execution and insert the corresponding ASL formula at each program point. The initial configuration is abstracted by initial formula $\langle true|emp\rangle$ and executions are conducted with rules we implemented, which will alter the formula with respect to the semantic of program. If errornous behavior like double-free and invalid-dereference appears, the execution ends the path.
4. After the execution π, the feasibility of the path and property violation checking will be encoded into an ASL formula as verification condition (VC). If the path is feasible and errors are found violating input properties, the tool find a violation path and stop to output. If no error is found for all the paths, an unknown result will be returned.

Several techniques are applied to improve performance and maintain the accuracy. On the aspect of checking, the tool only checking feasibility for program points that may make the constraints loose. Besides, to maintain the accuracy of the analysis. We use a byte-level implementation of syntactic ASL library which bytifies spatial predicates only when needed. With the information obtain by preset analysis, the memory region will be properly splitted for pointer arithmetic. Regarding the execution process, the formula will bytify the corresponding part of spatial predicate lazily to make the representation as simple as possible. The ASL library also provides interface to translate the syntactic formula into data structure used for solving.

2 Architecture

The architecture framework of our tool is shown in figure 1. Technically, our front end and back end are all implemented in the manner of LLVM passes in language C++. In this section we will discuss our front end and back end in detail.

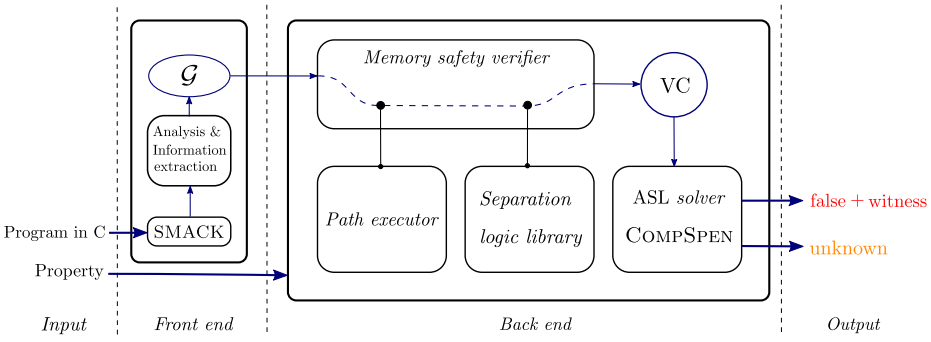

Fig. 1. Architecture framework of SESL

2.1 Front end

The front end of SESL is built on the existing tool SMACK, which translates the source code into Boogie intermediate representation (IR). The command-line interface first takes the input of user and converts source code to LLVM IR. Then after applying several LLVM optimization passes, LLVM IR will be translated into Boogie IR data structure. During the translation we give a record of type information and global variable initialization which will be of use in later analysis. The front end will finally construct a CFG for each function with states representing basicblocks and edges stand for `goto` statements in the function.

2.2 Back end

The back end of SESL is a symbolic execution engine based on separation logic. The engine takes the constructed CFG and property parameter as inputs and either give a property violating program path or gives a unknown output. The back end consists of four parts: memory safety verifier, separation logic library, path executor and separation logic solver.

Memory safety verfier This component contains the basic structure of the symbolic execution algorithm, which connect the path sampling, path execution, ASL formula solving and result producing. SESL samples the program path in a limited methodology since the infinity of program paths in a CFG. The strong connected component(SCC) of CFG will be distinguished and within each SCC the tool samples one path with preset limit of maximum ocurrences of a state and depth of function callings.

Path executor Path executor symbolically executes each path sampled and inserts syntactic separation logic formulas at each program point. To auxillary the execution, several analysises are also implemented in the module such as constant propagation, type and pointer analysis. During the execution of one path, these analysises can be used to determine possible violation of property, which can be regarded as a prunning technique for exhausting the symbolic execution graph.

Separation logic library and solver Based on the implementation of program abstract syntax tree (AST) of SMACK, we implemented a syntax library for separation logic. The library gives a good support for establishment and manipulation of byte-level separation logic formula, especially when it encounters pointer arithmetic and byte-level operations. Our separation logic solver CompSpen is a key component which gives a decent support for execution engine of SESL.

3 Strengths and Weaknesses

The strength of the SESL lies on the back end implemented. With the components stated above, SESL can do symbolic execution with simpler spatial

predicate `blk(x, y)` which not only simplifies the representation on syntax but also improves the efficiency on solving. Despite the precise representation, the tool also remains accuracy with our bytifying techniques. Another advantage of our tool is enabling the byte-level representation and manipulation of ASL, realized by our syntax library. With the benefit of ASL, it acts well with pointer arithmetic. Besides it is also implemented and tuned with respect to the semantic of C and standard library functions.

The drawback of SESL is the capability of the tool. Since the tool only carries an execution engine, it can now only conduct falsification of programs. The tool currently cannot give a over-approximation for loop behaviors and recursive function calls. To further specify the obstacles encountered when trying to give a proof of correctness, SESL finds it hard to deal with nondeterministic pointer arithmetic and heap behaviors without use of heavier methods such as abstract interpretation and model checking.

4 Tool Setup and Configuration

The competition submission is based on SESL version 1.0.4. The tool is open source, releases and usage can be found at

`https://spencerl-y.github.io/SESL/`

The installation instructions are clearly stated on the site. With the competition version of the tool, it is recommended to use the following command to analyze one single file:

```
./sesl-svcomp.sh -bw 64 -t --add-line-info <input_file>
```

where `--add-line-info` will stabbed line information to the output and `-bw 64` indicates the architecture considered for the checking is 64bit machine.

Participation SESL is attending the competition in the MemorySafety track of SV-COMP.

5 Software Project

SESL is a property of Institute of Software, China Academy of Sciences. The tool is maintained by Xie Li and is public available under an MIT license.

Acknowledgement....

References

1. Berdine, J., Calcagno, C., O'Hearn, P.W.: Symbolic execution with separation logic. In: Yi, K. (ed.) Programming Languages and Systems, Third Asian Symposium, APLAS 2005, Tsukuba, Japan, November 2-5, 2005, Proceedings. Lecture Notes in Computer Science, vol. 3780, pp. 52–68. Springer (2005). https://doi.org/10.1007/11575467_5, https://doi.org/10.1007/11575467_5

2. Rakamaric, Z., Emmi, M.: SMACK: decoupling source language details from verifier implementations. In: Biere, A., Bloem, R. (eds.) Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings. Lecture Notes in Computer Science, vol. 8559, pp. 106–113. Springer (2014). https://doi.org/10.1007/978-3-319-08867-9_7, https://doi.org/10.1007/978-3-319-08867-9_7
3. Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: 17th IEEE Symposium on Logic in Computer Science (LICS 2002), 22-25 July 2002, Copenhagen, Denmark, Proceedings. pp. 55–74. IEEE Computer Society (2002). https://doi.org/10.1109/LICS.2002.1029817, https://doi.org/10.1109/LICS.2002.1029817