2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks - Supplemental Volume (DSN-S) | 978-1-6654-3566-6/21/$31.00 ©2021 IEEE | DOI: 10.1109/DSN-S52858.2021.00014

Iteratively Checking Program Properties Involving Non-Boolean Values

Long Wang
Institute for Network Sciences and Cyberspace
Tsinghua University
longwang@tsinghua.edu.cn

***Abstract*—This paper presents our experience of iteratively checking program properties involving non-boolean values. We abstract a program into a set of transition rules and apply rewriting-logic based bounded model checker to do property computation. Exact values of integers and strings are captured in the abstract model of the program. Counterexamples found in the bounded model checking are used to refine the abstraction for pruning infeasible counterexamples. We demonstrated the effectiveness of the methodology by discovering a property violation in a real-world application (WuFTPD).**

Keywords—program property, program flow, rewrite logic, property computation

I. Introduction

Model-checking techniques can be leveraged to uncover program execution scenarios violating a certain software property. For example, a configuration error may result in granting the root access of a file to a regular user. The configuration values and the file access permissions involve non-boolean values.

Tools based on counterexample-guided abstraction refinement [1] apply the principle of iteratively abstracting programs for reducing the state space of the model checking. Existing tools of such type, e.g. SLAM [2], BLAST [3], use symbolic model checkers and abstract non-boolean values (e.g. integers and strings) in the original program into boolean variables or predicates. For example, typical data abstraction may transform a=3 into a>0. This may not be desirable in the scenarios where an exact value for a variable is required.

This paper presents our experience of iteratively checking program properties involving non-boolean values. We abstract a program into a set of rewriting rules representing symbolic execution of those program statements *directly* involved in the property computation. Exact values of integers and strings are captured in our abstract model. A rewriting-logic based bounded model checker (the Maude system [4] is used in our study) is employed to symbolically execute the abstract model. Rewriting logic based model checking allows the exact integer/string values to be efficiently manipulated, and also supports constraint computation for resolving predicates. Counterexamples found in the bounded model checking (BMC) are used to refine the abstraction and then rerun the analysis.

II. Methodology Overview

Our methodology performs property checking for arbitrary programs by employing a process flow involving several steps as illustrated in Figure 1 (the ovals represent processes, and the rectangulars represent inputs/outputs of the processes).

Step 1: Identification of program property and property variables

The user should provide information about the program property as input to our methodology. The user first identifies a program property (or program properties) to be checked, e.g., absence of double-locking. Based on the program property, the program variables, functions, and statements directly used in checking the property (property-related element set) are also identified. Then the user decides the property variables and represents the property as a specification to be checked against during BMC. Property variables are symbols corresponding to actual program variables in the property-related element set, or newly introduced to support computation necessary to check the program property.

Step 2: Extraction of CFGs

Our approach extracts from the program source code the CFGs (a CFG for each function) and the call graph with information of call sites, and singles out the statements that directly manipulate one of the property-related elements. We use the LLVM compiler [5] and implement extensions that can be executed as extra compilation passes.

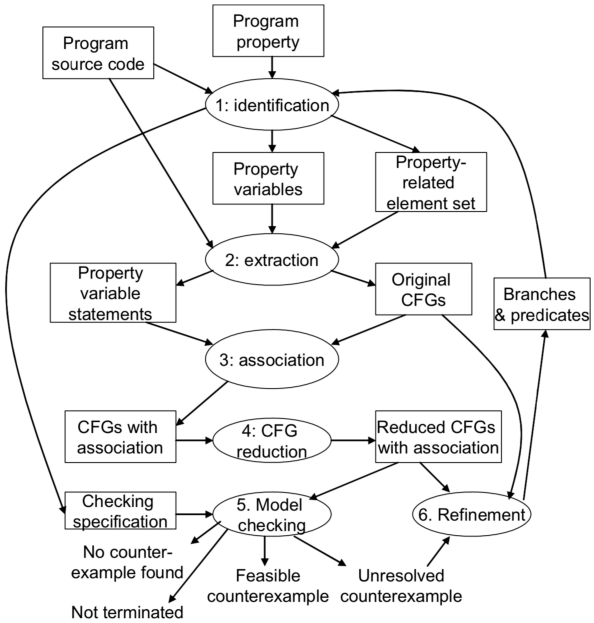

Figure 1: Process flow of our methodology

Step 3: Association of property variable statements with CFG nodes

We use a property variable statement to encapsulate a source code statement singled out in Step 2. Then the property variable statement is associated with the CFG node to which the source code statement belongs.

Step 4: CFG reduction

The output from Step 3 is a set of CFGs associated with property variable statements. We designed an algorithm to prune the nodes, edges, and function calls in the CFGs irrelevant to the

978-1-6654-3566-6/21/$31.00 ©2021 IEEE
DOI 10.1109/DSN-S52858.2021.00014

property to be checked. After the node/edge pruning, the original program CFGs are reduced to minimum CFGs while preserving property variable computation semantics.

Step 5: BMC of the reduced CFGs

We use the Maude formal language and bounded model checker [4] to symbolically execute the abstract model, i.e. the reduced CFGs associated with property variable statements. The model state in the BMC for an arbitrary sequential program is encoded in a four-entry tuple <*node; property_value_list; stack; stack_depth*>. In the tuple *node* identifies the last executed CFG node; *property_value_list* is the list of property variables with current values; *stack* is a list of *node* identifications representing the current call stack; *stack_depth* indicates the depth of the current call stack. Representing the call stack explicitly in the model state allows for executing function calls, including recursive function calls, during BMC. Symbolic execution of the reduced CFGs starts with an initial model state, which consists of initial values of all property variables, including symbolic constants representing undefined values. Each model state transition involves symbolic rewriting of *node, property_value_list, stack,* and/or *stack_depth* accordingly.

Bounded Model Checking outcomes. There are three kinds of outcomes from the bounded model checking:

(1) No counterexample found: this outcome shows that the property is checked as correct.

(2) Counterexamples found: A number of counterexamples are reported, including both feasible counterexamples and unresolved counterexamples. A feasible counterexample is reported when all branches in the reduced CFGs taken by the counterexample path are feasible. When it is not possible to resolve a predicate value for a branch taken along a counterexample path, we report an unresolved counterexample path; then the refinement procedure is performed to improve the program abstraction.

(3) Not terminated: The BMC does not terminate if the abstract model grows too large.

Step 6: Refinement

When an unresolved counterexample path is reported, a list of unresolved branches along the path in the reduced CFGs is exposed to the user. The source code is investigated to locate the predicates corresponding to these unresolved branches. Then the user identifies which of these predicates should be resolved in the next iteration of BMC. Variables and functions directly used in evaluating the identified predicates are added to the property-related element set (if they are not yet in the set). New property variables are also defined accordingly. Then the procedure in step 1 through step 6 is repeated.

Discussions. Now in our prototype, the steps 2, 4, and 5 are automated, by our LLVM extension, pruning algorithm, and the BMC, respectively. The steps 1, 3, and 6 are manually operated. The manual operations are labor-intensive and time-consuming. Automating these manual steps and minimizing relevant cost of the process is our next objective of this project.

III. Case Study

We performed a case study in the WuFTPD 2.6 software. If a regular user of the FTP service is able to access files as root, there is security vulnerability (privilege escalation) in the system. In our case study we check the property that "a regular user is unable to access files as root".

Identification of property variables. Two property variables, *euid* and *senddata*, are identified for the checking. *Euid* corresponds to the value of the effective user id; *senddata* indicates if a regular user is accessing a file.

Euid is assigned to a symbolic value *nonzero* when *seteuid(euid) or setreuid(rid, euid)* is executed (here the *euid* is a regular user ID) in the application; *euid* is assigned to 0 when *seteuid(0)* or *setreuid(rid, 0)* is executed. Any file access of a remote user is via the *send_data()* function in WuFTPD. When *send_data()* executes, *senddata=1;* otherwise *senddata=0.* During BMC we check if there is a scenario that "*euid=0 and senddata=1*".

Results. Our approach conducts three iterative procedures in this case study. Table 1 summarizes the experimental results when a stack size of 10 is employed, i.e. the states with *stack_depth* more than 10 are not explored during the BMC. The original row gives the baseline results when there is no CFG reduction.

Table 1: Results for the Case Study

Program CFGs	#functions	#nodes	Model size (lines)	Result
Iteration 1	65	463	979	Unresolved counterexamples
Iteration 2	65	578	1423	Unresolved counterexamples
Iteration 3	70	719	1615	Feasible and unresolved counterexamples
Original	220	7132	14299	Not terminated

After 3 iterations of checking, our approach confirms a feasible counterexample, and in that counterexample the property "a regular user is unable to access files as root" does not hold if there is a configuration error (so, restricted and unrestricted users in WuFTPD are not properly configured).

In another case study we confirmed that there is no double-locking in the WuFTPD 2.6 software as no counterexample is found in the first iteration.

IV. Conclusions

We present our iterative methodology that abstracts the software source code into a set of rewriting rules which capture exact values of non-Boolean variables. A certain property is checked by symbolically executing the abstract model in rewriting-logic based bounded model checking. Our case study demonstrates that we were able to check properties in large programs like FTP servers.

References

[1] F. Aarts, et al. Automata learning through counterexample guided abstraction refinement. In International Symposium on Formal Methods (pp. 10-27). Springer, Berlin, Heidelberg, 2012.

[2] T. Ball, et al. Thorough static analysis of device drivers, Proceedings of the EuroSys conference, 2006.

[3] D. Beyer, et al. The Software Model Checker BLAST, In International Journal on Software Tools for Technology Transfer (STTT), Vol. 9, No. 5-6, 2007.

[4] Maude 3.1 download and installation, Oct. 2020

[5] The LLVM Compiler Infrastructure, https://llvm.org