Poster: DBB-DSE : Dynamic Backward-Bound Dynamic Symbolic Execution

Jonghwan Yang
Chungnam National Univ.
South Korea
jhy7185@o.cnu.ac.kr

Seokwoo Choi
The Affiliated Institute of ETRI
South Korea
seogu.choi@gmail.com

Eun-Sun Cho
Chungnam National Univ.
South Korea
eschough@cnu.ac.kr

***Abstract*—In order to help deobfuscation of malware, this poster focuses on detecting on opaque predicates, a special kind of conditional expressions that always represent true or always false, which are essential for obfuscators to increase the complexity of analyses. We propose a dynamic opaque predicate detection method, focusing on data dependency aware selection of the portion of instructions to be symbolically analyzed. We adopt the backward bound dynamic symbolic execution (BB-DSE) scheme presented in [2], but our method enables handling loops in better ways; experimental results with real-world obfuscators such as Code Virtualizer [3] show that our proposed one performs slightly better than the previous one.**

I. Introduction and Related Work

Code Obfuscation is done to intentionally transform a code so as to make it hard to recognize the behavior of the original code. It is very useful for protecting programs from illegal modification and distribution by third parties. One common obfuscation technique is to add control movements and meaningless operations to the original program. As more practically-qualified obfuscation tools become widely available, an increasing number of malware attacks are obfuscated to avoid diagnoses of malware analysts. On the other hand, difficulties in deobfuscating malware due to the lack of sophisticated analysis tool is one of the major obstacles in malware analyses. To overcome this problem, we worked on a helper to deobfuscate malware, focusing on detecting "opaque predicates,". Opaque predicate is a special kind of conditional expression that represent always true or always false. Its behavior is like nothing but a skip operation, but still complicates the code without serious runtime cost.

As with existing deobfuscation methods, our tool extracts execution traces to feed a dynamic symbolic execution engine, since static binary analyses do not seem effective due to the various dynamic properties of obfuscated binary programs. The set of conditions collected from the symbolic execution is used for the SMT solver [6] to identify opaque predicates by means of logical verification. Note that a conservative analysis is necessary for opaque predicate detection, because to lower the complexity of analysis it entails the removal of the code blocks led by probable opaque predicates. That is, we might lose code when eliminating code blocks led by non-opaque predicates.

BB-DSE (Backward Bound Dynamic Symbolic Execution) [2] is a conservative opaque predicate detector, aiming to overcome the coverage problem of dynamic symbolic execution. With this tool, symbolic execution on a small number (N) of instructions prior to a given predicate is enough to detect opaque predicates conservatively; thus, it is safe to get rid of the code block led by the predicate. Since a predicate is opaque if one of the branches never occur, they conduct a feasibility test on the branch not shown in the trace to see if that branch never occurs (that is, it is infeasible.)

However, what they claimed might not always be applied well to other situations because they stick to the fixed number (N) of instructions to be symbolically analyzed beforehand, derived from the experimental results and heuristics; thus, when N is too small, it will get false positive errors in the feasibility test due to the lack of information, meaning that it missed opaque predicates in the detection. When N is too big, (for an extreme example, N covers the size of the entire program,) it will get false negative errors in the feasibility test, like symbolic execution on a single concrete execution trace, and evaluate conditional branches with concrete values in many cases; thus, normal predicates may be mistaken for opaque predicates. Therefore, finding the proper number N is critical to the quality of the analysis, and moreover it might vary from program to program, getting trickier with loop structures. This poster proposes DBB-DSE (Dynamic Backward Bound Dynamic Symbolic Execution), which adopts BB-DSE, but considers only the set of relevant instructions for symbolic execution, to improve the quality of opaque predicate detection.

II. DBB-DSE: The Proposed Tool

Figure 1 depicts the sequence of processes when DBB-DSE detects opaque predicates. First, it extracts an execution trace from obfuscated malware using trace logging tools. DBB-DSE then scans the traces and refines them into useful information. A trace address (TA) is the sequence number of an instruction in a trace, while an in-program address (PA) is similar to the program counter (PC). With PA and TA, DBB-DSE can dissimilate multiple executions from normal executions, as well as light-weight loop detection. Second, it scans the execution trace to spot jump instructions, which are closely related to possible opaque predicates.

Third, it conducts backward slicing based on data dependency from the possible opaque predicates. DBB-DSE uses Relevant number of Instructions (RI) to determine the range N, while the BB-DSE just uses the Number of Instructions (NI). This property is inherently from the result of considering data dependency, and provides flexibility according to the various properties of the obfuscated programs. DBB-DSE skips

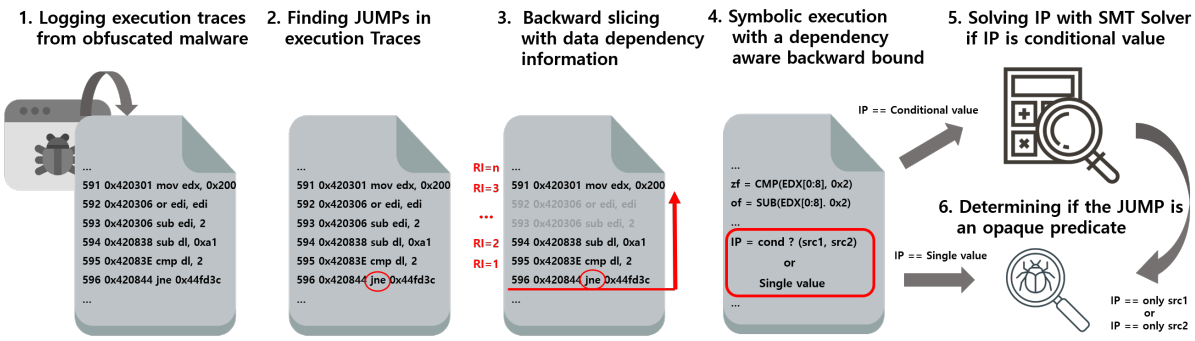


Fig. 1. Entire process of DBB-DSE

repetitive instructions in counting RI, which allows a flexible length for the list of instructions for symbolic execution. This is important because it enables us to handle even a long trace with repetitive instructions like loops, typically inserted by obfuscation tools, which is not possible with BB-DSE.

Fourth, it conducts dynamic symbolic execution on the proper portion of the trace. The portion is determined from the result of the previous step; the more sparsely the preceding instructions relevant to the predicates are located over the path, the longer the list of preceding instructions is needed to be symbolically executed. For symbolic execution, DBB-DSE first disassembles the trace of the obfuscated code into a low-level intermediate form. From the obfuscated code trace in Miasm Intermediate representation (IR), we are able to build basic blocks from the trace on the fly as well as conduct symbolic execution on the blocks in turns. Fifth, it determines if a given branch instruction depends on an opaque predicate by means of an SMT solver and the set of target addresses encoded with expressions of symbols. Note that if the encoded target is a constant instead of an expression the predicate the branch instruction is based on should be an opaque predicate, and thus it skips the symbolic expression solving step in this case.

III. Experiment and Conclusion

We build DBB-DSE with latest version of Miasm [5], Ubuntu 18.04.1, Z3 solver 4.8.5(64 bit)[6], and Python 2.7.15. Experiments are conducted on (1) real-world malware, obfuscated with Code Virtualizer with no input, (2) simple C programs with some opaque predicate complied with gcc and (3) obfuscated C programs with O-LLVM.

The range of proper RI values, reflecting data dependency, is narrower than that of NI regardless of sizes and number of blocks of the target We found that both NI 39 and RI 15 provide the best set of instructions to analyze for opaque

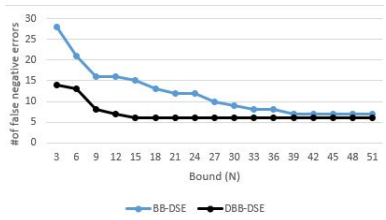


Fig. 2. False negative errors with the same bound N

TABLE I. Summary of the best detection environments, comparing BB-DSE and DBB-DSE

	BB-DSE	DBB-DSE
♯ of preceding instr.	NI=39	RI=15
♯ of true pos.	119	120
♯ of true neg.	NI=38	RI=38
♯ of false pos.	0	0
♯ of false neg.	7	6
f-measure	0.971	0.976

predicates without any false positive error. However, note that the range of 12 to 24, which the authors of BB-DSE suggested as the best NI values for opaque predicate detection for O-LLVM-based obfuscation, does not work well in this case; NI 39 is beyond the range of 12 to 24. Figure 2 shows the number of false negative errors as N grows, which bounds NI in BB-DSE and RI in DBB-DSE. Table I is a summary of the best results from BB-DSE (with NI=39) and DBB-DSE (with RI=15). At their best BB-DSE detects one less opaque predicate (that is, 119) than DBB-DSE (that is, 120) because with data dependency information, BB-DSE is not robust enough against the repetition of opaque predicates, especially when a preceding instruction relevant to the suspicious predicate is not repeating with it.

The contribution of DBB-DSE is that it introduces data dependency-aware backward bounds for dynamic symbolic execution (1) to allow an even bound (near 15) for the number of instructions preceding suspicious predicates, regardless of the various properties of obfuscated codes, so useful in practice and (2) to enable more detection of opaque predicates than previous work, despite loops and repetitions. We are currently working on further experiments, and planning to combine logical reasoning with our DBB-DSE.

References

[1] *A Dynamic Backward Bound Dynamic Symbolic Executor for Opaque Predicate Detection* https://github.com/PLASLaboratory/op-eliminator

[2] S. Bardin at.al, *Backward-Bounded DSE: Targeting Infeasibility Questions on Obfuscated Codes*, IEEE Symposium on Security and Privacy(IEEE S&P), 2017.

[3] *Code Virtualizer*, https://www.oreans.com/CodeVirtualizer.php

[4] *Obfuscator-LLVM*, https://github.com/obfuscator-llvm/obfuscator/wiki

[5] *Miasm*, https://github.com/cea-sec/miasm

[6] *Z3 solver*, https://github.com/Z3Prover/z3

DBB-DSE : Dynamic Backward-Bound Dynamic Symbolic Execution

Jonghwan Yang
Chungnam National Univ.
South Korea
jhy7185@o.cnu.ac.kr

Seokwoo Choi
The Affiliated Institute of ETRI
South Korea
seogu.choi@gmail.com

Eun-Sun Cho
Chungnam National Univ.
South Korea
eschough@cnu.ac.kr

Motivation

Due to the lack of sophisticated analysis tools, difficulties in deobfuscating malware is one of the major obstacles in malware analyses.

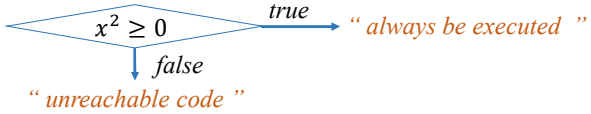
We concentrate **on opaque predicate detection**, and propose DBB-DSE, a conservative opaque predicate detection with dynamic symbolic execution.

What is an Opaque Predicate?

A predicate that represent always false or always true.

Its behavior is like nothing but a skip operation, but still complicates the code without serious runtime cost.

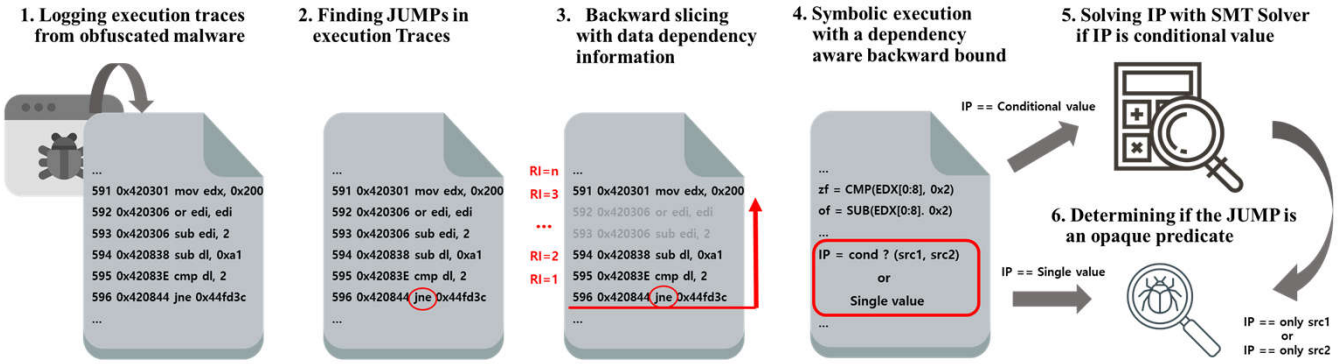
So, opaque predicate is essential for obfuscators to increase the complexity of analysis.



Overall Process of the Proposed Method

Our DBB-DSE adopts the backward bound dynamic symbolic execution scheme presented in BB-DSE[1], where suggested that dynamic symbolic execution on a small number (N) of instructions prior to a given predicate is enough to detect opaque predicates conservatively

Different from the previous tool, we consider data dependency in the subtrace extraction, to improve the quality of detection including in loop-structured programs; by extracting only the set of relevant instructions using program slicing



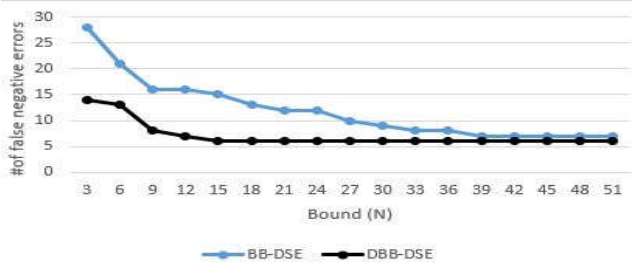
[1] S. Bardin at.al, Backward-Bounded DSE: Targeting Infeasibility Questions on Obfuscated Codes, IEEE Symposium on Security and Privacy(IEEE S&P), 2017

Experiments

We used Intel Pin tool for logging execution trace, Miasm for symbolic execution and disassemble, Z3 4.8.5 as a constraint solver and Python 2.7.15 in developing the customized program slicer

Experiments are conducted on (1) plain programs including opaque predicate , (2) obfuscated programs with Code Virtualizer and (3) obfuscated programs with Obfuscator-LLVM,

the no. of Instruction (NI) 39 and no. of Relevant Instruction (RI) 15 provide the best set of instructions to analyze for opaque predicates without any false positive error



False negative errors with the same bound N

	BB-DSE	DBB-DSE
♯ of preceding instr.	NI=39	RI=15
♯ of true pos.	119	120
♯ of true neg.	NI=38	RI=38
♯ of false pos.	0	0
♯ of false neg.	7	6
precision	1	1
recall	0.944	0.952
f-measure	0.971	0.976
elapsed time	3.800	4.972

Summary of the best detection environments comparing BB-DSE and DBB-DSE

Conclusions

The contribution of DBB-DSE is as follows :

(1) We introduces data dependency-aware backward bounds for dynamic symbolic execution to allow an even bound (near 15) for the number of instructions preceding suspicious predicates, **flexible to cope with various situations**.

(2) Our tool is **more useful in practice** and to enable **more detection** of opaque predicates than previous work, **despite loops and repetitions**.