JAIST Repository
https://dspace.jaist.ac.jp/

Title	A Binary Code Analysis and Approximation Techniques
Author(s)	BINH, NGO Thai
Citation	
Issue Date	2015-03
Type	Thesis or Dissertation
Text version	author
URL	http://hdl.handle.net/10119/12640
Rights	
Description	Mizuhito Ogawa, School of Information Science, Master

JAIST
JAPAN
ADVANCED INSTITUTE OF
SCIENCE AND TECHNOLOGY

Japan Advanced Institute of Science and Technology

A binary code analysis and approximation techniques

BINH, NGO Thai (1310022)

School of Information Science,
Japan Advanced Institute of Science and Technology

February 12, 2015

Keywords: constant propagation, BE-PUM, binary analysis.

There are two main targets of binary code analysis. The first one is *System software*, which is compiled code but its source is inaccessible, due to legacy software and/or commercial protection. It is often large, but relatively structured from the compiled nature, and the main obstruction is scalability. The second one is *Malware*, which is distributed in binary only. It is often small, but with tricky code for obfuscation.

Many model generation tools for binary programs have been proposed. They attempt to infer a *control flow graph (CFG)* of the program, on which popular techniques like model checking can be adopted. A CFG of high-level programming language is straightforward and is obtained by syntactical parsing. However, binary code consists of bit sequences, which is parsed dynamically and has no distinction between code and data, hence the CFG can only be built on-the-fly. Based on Jakstab, a static-based binary disassembler, our collaborator at VNU-HCM has developed BE-PUM (Binary Emulation for PUshdown Model generation) that focuses on building the CFG of malware programs.

There is a proplem that although we have assumed that programs are terminating, there are cases that BE-PUM *almost* does not terminate, even for a simple program. One reason is because the initial environment is not fixed. We apply constant propagation analysis on the partial CFG and use the analyzed data to decide whether BE-PUM will or will not continue the path. A variable is called a *constant* at one location ℓ if regardless

Copyright © 2015 by BINH, NGO Thai

of the initial environment, after executing on the same path, its the value at ℓ must be the same. Our approach is that if a jump revisits a node that was explored in the graph, we will continue checking the node only if the condition is a constant. This process is similar to loop unrolling, but we only unroll if there is a guarantee that the normal execution will also follow the same path, and since we have assumed that input programs are terminating, the unrolling process will also terminate.

Contribution

1. We have modified BE-PUM in a way that instead of blindly follow any path, we only follow if there is a guarantee that the path is always taken for any initial environment. The judgment bases on a constant propagation analysis on the partial CFG, which is done on-the-fly during graph construction.

2. We have designed a constant propagation analysis that can be applied to X86 assembly programs (including memory and the stack, although they are restricted), and can be done on-the-fly together with BE-PUM execution.

3. The runs of BE-PUM with constant propagation application will eventually terminate, if the input programs are correctly implemented ,i.e. terminating for all input environments, and the set of the possible locations of the next instructions is finite, for any instruction in the program (this is due to the exhaustive destination check of BE-PUM).

4. Experiments have been done to compare the three versions: normal unrolling, convergence check and unrolling based on the result of constant propagation, on 1000 from real-world malware samples.

One remark is that although `CP` eventually finishes, its result is an under-approximation of the binary program: there have the cases that the skipped paths eventually go to new locations. To generate the complete CFG, we will need further analysis such as loop invariant, refinements, ... Such problems are regarded as future works.