

Program Reachability for Vulnerability and Malware Analysis

Problem

Highly skilled Department of Defense (DoD) malware and vulnerability analysts currently spend significant amounts of time manually coercing specific portions of executable code to run.

Solution

Automate the analysis of binary code, choosing program inputs that will trigger specific behavior to reduce the time that DoD cyber personnel spend performing complex software analysis.

Approach

Use model checking techniques to identify these inputs and generate a simplified executable free of complex and convoluted dependencies that can be analyzed by existing code analysis tools.

Intended Impact (FY18-20)

Improve the DoD's ability to measure and monitor the advancement of path-reachability research, especially as Ghidra decompilation quality improves.

Testing Method

A total of 91 test programs were compiled for three optimization levels, and two architectures. Each test attempted to find a path from a starting location to a reachable goal, and an unreachable goal. If both answers were correct, the test passed. The test timeout was 30 minutes.

2,184 test configurations found several successful approaches, but **none** that **consistently outperformed** the others, suggesting a needed hybrid **approach**.

Test Case Configuration

Optimized	Arch
None	32-bit
None	64-bit
Medium	32-bit
Medium	64-bit
High	32-bit
High	64-bit
Total	

Key

- Best result
- Second best result
- Third best result
- Worst result

Pharos Function Summaries

SEI's Pharos binary analysis framework computes symbolic function summaries by symbolically executing binary code. This technique converts these function summaries into light-weight constraints. The conversion uses a simple model of memory that is very efficient, but is known to be incorrect in the presence of interprocedural reasoning.

Fail	Timeout	Pass
55	2	34
47	0	44
40	0	51
53	0	38
50	0	41
32	1	58
257	3	266

This approach is very fast, but its imprecision results in a large number of failures and a small number of passing tests.

Weakest Precondition

The weakest precondition algorithm finds the weakest constraints on the program input that are required for the program to terminate successfully. We force execution to the desired program locations by adding assertions. This technique uses an array encoding of memory, which is precise but expensive to reason about. It also cannot reason generally about loops.

Fail	Timeout	Pass
16	2	73
15	3	73
9	3	79
9	4	78
6	2	83
28	3	60
83	17	446

This well-known approach is still the benchmark to beat. It performs well, but has significant deficiencies when analyzing code with loops.

Property Directed Reachability

Property Directed Reachability (PDR) is a technique used in source code software model checking. It iteratively generates an inductive invariant to prove that the target code is unreachable, and uses counter-examples to refine the invariant, so it can prove targets are unreachable even when there are loops. It uses the same array encoding of memory as the previous technique.

Fail	Timeout	Pass
3	29	59
2	36	53
1	13	77
1	17	73
1	12	78
2	16	73
10	123	413

This approach is very accurate, but has severe performance problems in the binary domain, due the array memory model, which is not necessary at the source code level.

Ghidra + Seahorn

This technique uses the NSA's Ghidra decompiler to raise the executable code to a C-like language rather than trying to express the binary semantics directly. The Seahorn software model checker is then used to check reachability using PDR. Because it operates on a source code representation, the encoding is very different than the other PDR approach.

Fail	Timeout	Pass
21	7	63
28	2	61
12	7	72
21	6	64
18	7	66
32	5	54
132	34	380

PDR can be fast when using a source code representation. Unfortunately, decompilation can fail in myriad ways, and this accounts for the majority of failures for this approach.

- Copyright 2020 Carnegie Mellon University.
- This material is based upon work funded and supported by the Department of Defense under Contract No. FA8702-15-D-0002 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center.
- The view, opinions, and/or findings contained in this material are those of the author(s) and should not be construed as an official Government position, policy, or decision, unless designated by other documentation.
- **NO WARRANTY. THIS CARNEGIE MELLON UNIVERSITY AND SOFTWARE ENGINEERING INSTITUTE MATERIAL IS FURNISHED ON AN "AS-IS" BASIS. CARNEGIE MELLON UNIVERSITY MAKES NO WARRANTIES OF ANY KIND, EITHER EXPRESSED OR IMPLIED, AS TO ANY MATTER INCLUDING, BUT NOT LIMITED TO, WARRANTY OF FITNESS FOR PURPOSE OR MERCHANTABILITY, EXCLUSIVITY, OR RESULTS OBTAINED FROM USE OF THE MATERIAL. CARNEGIE MELLON UNIVERSITY DOES NOT MAKE ANY WARRANTY OF ANY KIND WITH RESPECT TO FREEDOM FROM PATENT, TRADEMARK, OR COPYRIGHT INFRINGEMENT.**
- [DISTRIBUTION STATEMENT A] This material has been approved for public release and unlimited distribution. Please see Copyright notice for non-US Government use and distribution.
- Internal use:* Permission to reproduce this material and to prepare derivative works from this material for internal use is granted, provided the copyright and “No Warranty” statements are included with all reproductions and derivative works.
- External use:* This material may be reproduced in its entirety, without modification, and freely distributed in written or electronic form without requesting formal permission. Permission is required for any other external and/or commercial use. Requests for permission should be directed to the Software Engineering Institute at permission@sei.cmu.edu.
- * These restrictions do not apply to U.S. government entities.
- Carnegie Mellon® is registered in the U.S. Patent and Trademark Office by Carnegie Mellon University.
- DM20-0861