

Contents

Introduction	vii
1 Symbolic Execution	1
1.1 Overview	1
1.1.1 Example	2
1.2 Limitations and Popular Solutions	3
1.2.1 Handling Symbolic Memory	4
1.2.2 Path Explosion	4
1.2.3 Interaction With Environment	5
1.2.4 Constraint Solving	5
1.3 Concolic Execution	6
1.3.1 Algorithm	6
1.3.2 Limitations	7
1.4 Discussion	7
2 A Dissertation on Synchronization	9
2.1 A note about debuggers	9
2.2 The purpose of the technique	10
2.3 A simple overview	11
2.4 Limitations	11
2.5 Different approaches to memory synchronization	11
2.6 Differencies with Concolic Execution	12

3	A Debugger-Agnostic Implementation	13
3.1	Angr Overview	13
3.1.1	Fundamental Objects	14
3.1.2	Memory Plugin	14
3.1.3	Simprocedures	15
3.2	AngrDBG Overview	15
3.2.1	Abstract Debugger	15
3.2.2	Memory Types	15
3.2.3	Memory Plugin Patch	15
3.3	Using AngrDBG API	15
3.3.1	Remote Server	15
4	Frontends	17
4.1	IDAngr	17
4.1.1	Using IDAngr from IDAPython	17
4.1.2	Using the GUI	17
4.2	AngrGDB	17
4.2.1	Using AngrGDB	17
5	Use case	19
6	Conclusion	21
6.1	Further work	21
6.2	Final Words	21

Introduction

Binary analysis is one of the most important problems in computer security. A wide family of manual and automatic techniques was developed among the years. They are divided into static and dynamic analysis depending on the need to execute the program or not.

Despite the continuous innovation on the front of the automatic analysis, the human part remains essential. A reverse engineer is a person who tries to understand what a binary program does and how. This process usually involves reading and understanding the disassembled code and its effects during execution with the aid of a debugger.

One of the most used automatic techniques nowadays is Symbolic Execution. The idea came about in the '70s but only recently its applications became relevant in computer security. Symbolic execution is used for different tasks, from deobfuscation to vulnerability research (joined with fuzzing) or automatic exploit generation.

It is classified as a static analysis technique because in the pure version of symbolic execution the binary is not really executed. The approach is borderline (we will discuss it later) and the evolution of symbolic execution, Dynamic Symbolic Execution, is a dynamic analysis technique without any doubt.

In the last years, symbolic execution becomes also a first-class technique used by anyone who deals with manual reverse engineering.

Execute symbolically an entire complex software (like a web server for example) is a huge task for a machine, even for a supercomputer so the analyst often uses it in a surgical manner on small pieces of code. A reverse engineer often needs to reverse custom encryption functions or obfuscated code during the dynamic analysis.

In this thesis, we introduce the idea of combining symbolic execution with dynamic analysis for reverse engineering. The synchronization between a debugger and a symbolic executor can enhance manual dynamic analysis and allow a reverser to easily solve small portions of code without leaving the debugger.

We implemented a synchronization mechanism on top of the binary analysis framework `angr`. This means a method to transfer the state of the debugged process in the `angr` environment and back.

The backend library is debugger agnostic and can be extended to work with various frontends. We implemented a frontend for the IDA Pro debugger and one for the GNU Debugger.