Björn Kirschner

Technische Universität München

GetInTUM - Door Access System

Idea Report for Android Practical Course (F13)

May 18, 2015

Contents

---

1 Introduction

2 Selective Symbolic Execution

Symbolic execution is an advanced analysis technique particularly suited for automated software testing and malware analysis . Instead of concrete input (7, "string", ...) symbolic execution uses symbolic values (λ, β, ...) when processing code. Assignments in the program path have impacts on these symbolic values. The integer calculation $x = x - 2$, for instance, would update the symbolic expression representing the input x to $\lambda - 2$. Conditional statements (if <condition> then ... else ...) fork program execution into two new paths. Both paths are then constrained by an additional condition, the 'then' branch with the if-condition and the 'else' branch with the negated if-condition respectively.

cite!

cite

Following this procedure results in a tree-like structure of constrained symbolic expressions. A constraint solver can now take all constraints along one execution path as input and find one concrete input (e.g., $\lambda = 5$) which would lead to the program following exactly this path. Such results greatly alleviate writing reproducible test cases .

cite

On a technical level, symbolic execution engines save state information (program memory, constraint information, ...) in a custom data structure. Each conditional statement involving symbolic values results in a $fork$ of the program state. The two newly created branches are completely independent and can therefore be processed in parallel.

But the exponential growth of conditionals soon reveals scaling problems of this forking strategy. Despite heavy research on optimisations mitigating this *path explosion* problem only relatively small programs (≅ thousands of lines of code) can be analysed symbolically . cite cite

Additionally, symbolic execution faces problems when the program under analysis *interacts with its environment*. If it calls a system library like *libc*, in theory the whole system stack including invoked libraries, operating system and drivers would have to be executed symbolically. Considering the path explosion problem mentioned before, the resulting complexity makes such a profound analysis hardly feasible.

One way to solve this problem is to build abstract models of the program's environment . However, due to the complexity of real-world systems, building a model of the entire system is both tedious and unnecessary - the user usually wants to analyse one single program and not the whole system . cite cite

In order to overcome typical problems of conventional symbolic execution, Chipounov et al. from the Swiss Federal Institute of Technology in Lausanne (EPFL) developed the concept of **selective symbolic execution** (S^2E) . Based on a virtual execution platform S^2E gives users the illusion of running the entire system symbolically. By limiting the scope of interest (i.e. which parts of the system should be executed symbolically), users can effectively restrain the path explosion problem. Program code within this defined scope is executed symbolically, whereas out-of-scope parts, which are irrelevant to the analysis, switch to concrete execution. cite

Definition of the scope of interest (what to execute symbolically) is highly flexible. Users may specify whole executables, code regions, or even single variables to be executed symbolically. Everything else will be treated concretely.

But since on a technical level symbolical and concrete execution are handled very differently - concrete code may run natively while symbolic instructions need to be emulated - switching back and forth these two modes is a major challenge. Hence one of the main contributions of the EPFL team around Chipounov is the transparent and consistent management of switching between symbolic and concrete execution modes.