Information and Software Technology 66 (2015) 40–57

Contents lists available at ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof

Goal-oriented dynamic test generation

TheAnh Do [a], Siau-Cheng Khoo [b], Alvis Cheuk Ming Fong [a], Russel Pears [a,*], Tho Thanh Quan [c]

[a] Auckland University of Technology, 2-14 Wakefield St, Auckland 1010, New Zealand
[b] National University of Singapore, COM1, 13 Computing Drive, Singapore 117417, Singapore
[c] Ho Chi Minh City University of Technology, 268 Ly Thuong Kiet St, Ho Chi Minh City, Viet Nam

ARTICLE INFO

Article history:
Received 5 July 2014
Received in revised form 22 April 2015
Accepted 30 May 2015
Available online 6 June 2015

Keywords:
Buffer overflow vulnerabilities
Dynamic symbolic execution
Data and control dependence analysis
Type inference analysis

ABSTRACT

Context: Memory safety errors such as buffer overflow vulnerabilities are one of the most serious classes of security threats. Detecting and removing such security errors are important tasks of software testing for improving the quality and reliability of software in practice.
Objective: This paper presents a goal-oriented testing approach for effectively and efficiently exploring security vulnerability errors. A goal is a potential safety violation and the testing approach is to automatically generate test inputs to uncover the violation.
Method: We use type inference analysis to diagnose potential safety violations and dynamic symbolic execution to perform test input generation. A major challenge facing dynamic symbolic execution in such application is the combinatorial explosion of the path space. To address this fundamental scalability issue, we employ data dependence analysis to identify a root cause leading to the execution of the goal and propose a path exploration algorithm to guide dynamic symbolic execution for effectively discovering the goal.
Results: To evaluate the effectiveness of our proposed approach, we conducted experiments against 23 buffer overflow vulnerabilities. We observed a significant improvement of our proposed algorithm over two widely adopted search algorithms. Specifically, our algorithm discovered security vulnerability errors within a matter of a few seconds, whereas the two baseline algorithms failed even after 30 min of testing on a number of test subjects.
Conclusion: The experimental results highlight the potential of utilizing data dependence analysis to address the combinatorial path space explosion issue faced by dynamic symbolic execution for effective security testing.

1. Introduction

Automated software testing is increasingly being seen as an important means for improving the quality and reliability of software in industry. It mitigates the hardship of manual testing, which is labor-intensive and error-prone, and alleviates the expensive cost of software testing, which often accounts for around half of the total software development costs. One way of enhancing automated software testing is to automate the process of test input generation. Over the last three decades, considerable research effort has attempted to achieve this goal, ranging from random testing [21], symbolic execution [39], search-based testing [28], the chaining approach [20, 37], to dynamic symbolic execution [9, 23, 45].

Among these proposed techniques, dynamic symbolic execution has been gaining a considerable amount of attention in the current industrial practice [11]. Through the power of the underlying constraint solver, it intertwines the strengths of random testing and symbolic execution to achieve the scalability and high precision of dynamic analysis. One of the most important insights of dynamic symbolic execution is the ability to reduce the execution into a mix of concrete and symbolic execution when facing complicated pieces of code, which are the real obstacle to *classical* symbolic execution. The technique has been applied to the testing of many industrial software systems and uncovered "million-dollar" bugs [5, 26]. While effective, the fundamental scalability issue limiting the capability of dynamic symbolic execution is the combinatorial explosion of the path space, which can be extremely huge or often infinite in sizable and complex programs. This phenomenon has been significantly highlighted in several research studies:

* Corresponding author. Tel.: +64 9 921 9999x5344.
E-mail address: russel.pears@aut.ac.nz (R. Pears).

http://dx.doi.org/10.1016/j.infsof.2015.05.007
0950-5849/

> ... path explosion represents one of the biggest challenges facing symbolic execution, and given a fixed time budget, it is critical to explore the most relevant paths first. [14]

> A significant scalability challenge for symbolic execution is how to handle the exponential number of paths in the code. [11]

> In theory, systematic dynamic test generation can lead to full program path coverage, i.e., program verification. In practice, however, the search is typically incomplete both because the number of execution paths in the program under test is huge ... [25]

The impact of this particular limitation of dynamic symbolic execution on the efficiency of software testing is significant. If dynamic symbolic execution is carried out in a way that exhaustively and systematically explores all feasible paths of the program under test, then it often ends up with only small regions of the code explored. Consequently, in practice the objective of achieving high structural coverage of software testing is hard to realize using dynamic symbolic execution. More importantly, the capability of detecting errors can be limited since the code harboring errors may not even be exercised. The `CheckArray` function in Fig. 1 could be a good example to illustrate this phenomenon.

It takes as input an array of 20 elements and checks if all elements equal 25. This yields 2^{20} (=1,048,576) paths with just 20 symbolic predicates. In practice, this path space explosion problem becomes worse as the input of programs can be a stream of data with too large (or unknown) size [24]. In the attempt to "*explore the most relevant paths first*" [14], a major challenge arising from path exploration is among the far too many program paths, how to mine for appropriate paths for quickly achieving desired testing criteria. Consider the execution of branch (5, 6) in `CheckArray` function, for example. The first observation is that this branch does not form any symbolic predicate as its conditional expression depends on the locally declared variable `success`; any attempt to flip its alternative branch to trigger its execution will fail. The second observation is that among the 1,048,576 paths, there is only one path that executes all "*else*" branches at the conditional statement 3 to propagate the desired `true` value of `success` down to statement 5 to execute branch (5, 6). These observations demonstrate difficulties in developing path exploration algorithms where the execution of code does not depend directly on the symbolic input. This is widely adopted in programming practices, however. For instance, Cadar et al. [12], when testing a number of medium-sized applications, found that less than 42% of the executed statements depend on the symbolic input. Independently, Binkley et al. [6] studied the testability transformation problem in search-based testing, and observed that the variety usage of Boolean-typed variables complicates test input generation and degrades program testability. Of the 23 buffer overflow vulnerabilities in our experimental study, none depends directly on the symbolic input.

To cope with such challenges, we present in this paper an approach to improve the dynamic symbolic execution-based path exploration process in the context of *goal-oriented testing*. Stated formally:

> *Given a test goal g (e.g. statement or branch) in the program P, the goal is to find a test input t with which g is executed.*

To begin with, we utilized the chaining approach [20, 37] to form a search mechanism. Particularly, given a test goal to explore, the chaining approach first performs data dependence analysis to identify statements that affect the execution of the test goal, and then uses these statements to create sequences of events that are to be executed prior to the execution of the test goal. The advantage of doing this is twofold: (1) it precisely focuses on the cause of getting the test goal to be executed and (2) it slices away code segments that are irrelevant to the execution of the test goal.

Next, we propose a search algorithm, named Guider, which is driven by the chaining mechanism and utilizes dynamic symbolic execution to perform path exploration for exploring the test goal. Guider distinguishes itself from existing search algorithms in three major aspects: (1) it mitigates the path explosion problem by centralizing on data dependences which truly affect the executability of the test goal; (2) it is able to refine path exploration when the local search space is saturated; and (3) it determines control dependences on the fly and exploits the static program structure to optimize path exploration.

Lastly, we develop a dynamic symbolic execution-based buffer overflow testing framework, named Sebo. Sebo works in two phases. In the first phase, it uses Deputy [13]—an advanced type system for pointers, to diagnose potential runtime violations on buffer operations in the program under test. In the second phase, it uses Crest [7]—an extensible symbolic execution engine, to perform dynamic symbolic execution for test input generation.

We implemented our proposed algorithm—Guider, in the Sebo framework and conducted experiments against 23 buffer overflow vulnerabilities to evaluate its effectiveness. We observed a significant improvement of Guider over two widely adopted search algorithms in dealing with the path explosion problem to uncover buffer overflow vulnerabilities.

```
Node    typedef enum {false, true} bool;
        #define N 20
(s)     bool CheckArray (int A[N]) {
            int i;
(1)         bool success = true;
(2)         for (i = 0; i < N; i ++) {
(3)             if (A[i] != 25)
(4)                 success = false;
            }
(5)         if (success) {
(6)             // target
            }
(7)         return success;
(e)     }
```

Fig. 1. The function `CheckArray` checks if all elements of an input array equal 25. This example is used to illustrate the path explosion problem facing dynamic symbolic execution and the difficulty of developing path exploration algorithms in which the code under test does not directly depend on the symbolic input. It is also used to illustrate the search mechanism in the chaining approach.