Designing New OS Primitive(s) for Efficiently Fuzzing COTS Software

JAY BOSAMIYA and ANJA KALABA, Carnegie Mellon University

Fuzzing is a well-known technique for automatically finding bugs in common-off-the-shelf (COTS) software. It consists of repeatedly (smartly) generating random inputs to run on a program-under-test (PUT), until a bug is found. Since the same program is run on multiple related inputs repeatedly, it begs the question of whether it would be possible to reduce the time taken for each execution, thereby increasing the efficiency of the fuzzer and increasing the number of bugs it identifies. In this paper, we propose a new syscall, with multiple alternative designs, and perform a feasibility analysis for it. We show that for some applications, up to 75% of the cost of execution can be amortized using our syscall.

1 INTRODUCTION

Common Over The Shelf (COTS) software is generally buggy. These bugs can range from simple hangs and crashes, all the way up to security-relevant vulnerabilities that can threaten critical data and systems. As a technique for finding bugs, fuzzing [Miller and Fredriksen 1990] has remained popular due to its empirically demonstrated ability to find real-world software flaws and vulnerabilities. Many techniques for improving the state-of-the-art for fuzzing have been shown over the years (see [Manès et al. 2019] for a recent survey). However, certain aspects of the *fuzzing loop* appear to be under-studied. We believe that by designing new OS primitive(s), we can improve the *program execution* part of the loop, thereby improving the efficiency of state-of-the-art fuzzers.

At a very high-level, all fuzzers follow a simple loop where they perform (1) input generation (2) program execution (3) evaluation of the output of the program and potentially (4) updating the fuzzer state. Fuzzing practitioners have explored various design decisions in each of these 4 stages, with varying success. In this project however, we are interested in the interplay between input generation and program execution, with an emphasis on the latter. In particular, we are interested in increasing the performance of fuzzers by reducing the cost of program execution through amortization of initialization, while also potentially (ab)using properties of the input generation.

Fuzzing is usually done in *campaigns* of a fixed cost or time budget, where the *program under test* (PUT) is repeatedly executed with different inputs. Since the core component of such a campaign consists of a single hot loop, any improvements in performance would increase the number of potentially useful *executions-per-second*, which allows for further exploration of the program's state space under the same budget. A common bottleneck for fuzzing programs and libraries tends to be a large initialization cost, before any "real" execution begins. There are many examples for such initialization cost, such as web browsers, word processors, PDF readers, etc. To combat this, the fork-server and in-memory execution techniques have been suggested. The former uses the PUT as a fork-server, similar to certain web-servers like *gunicorn*, thereby amortizing initialization cost, and the latter goes a step further by repeatedly executing the PUT within the same process. These have significant drawbacks however, requiring careful manual insertion of safe fork points, or even ensuring that there is no global state, as is required in the case of the in-memory execution technique.

Since the in-memory technique is applicable only for very specific programs that behave *essentially* like libraries, we will not discuss it any further. The fork-server model however, is more general. The most naïve approach to this model amortizes the expensive `execve` syscall by performing a `fork` immediately after the `execve`, and then waiting on a signal from the fuzzer to continue execution. An obvious extension to this approach allows the programmer to manually insert the `fork` into the program, after any initialization code. However, it is known that the `fork` syscall does not scale extremely well across multiple cores (amongst other issues [Baumann et al. 2019]). Thus, in [Xu et al. 2017], the authors suggest the introduction of a new operating system primitive, namely a `snapshot` syscall, which can scale well. With their implementation, they show substantial improvements in parallelism. However, this still suffers the drawback of requiring manual insertion of the `snapshot` call into the PUT, taking care to place it right before anything useful happens, but after the initialization is complete.

We propose a new syscall that achieves similar behavior in an automated way, and could also provide further improvements that a static manually-inserted `snapshot` call cannot provide. In particular, we propose to design new primitive(s) that can be used, with minimal programmer overhead (by amortizing even across different PUTs), to achieve similar or (hopefully) better performance gains as previous techniques. Additionally, we notice that for *mutational fuzzing*, in which input generation is done by selecting inputs from a *seed corpus* and mutating them to produce novel inputs, there are scenarios within which the PUT might execute the same early execution phases too. For example, if the mutation operators are designed to stay within a specific video codecs, then the initial parsing code that decides which decoder to use will