Systematic Bit-Flip Fault Injection and Exploration using Java PathFinder

Pu Yi
Peking University
Beijing, China
lukeyi@pku.edu.cn

Pavel Parízek
Charles University
Prague, Czech Republic
parizek@d3s.mff.cuni.cz

Cyrille Artho
KTH Royal Institute of Technology
Stockholm, Sweden
artho@kth.se

ABSTRACT

Computer hardware faults or radiation may induce bit flips in memory. To assess the impact of such events, systematic fault injection can test the resiliency of software against bit flips. We present a fault injection engine using Java PathFinder (JPF) to systematically inject and explore bit-flip faults in user-specified variables in Java programs. Our JPF extension verifies the behavior of several algorithms that detect single bit flips.

Keywords: Fault Injection, bit flip, concrete exploration, Java PathFinder

1. OVERVIEW

Computer hardware is susceptible to errors. Hardware defects or radiation can induce errors to the hardware, which can result in a memory bit being flipped. Considering the increasing complexity and scale of the computer hardware, it is important to improve the resiliency of software against hardware errors. However, these hardware errors are non-deterministic and hard to reproduce. To evaluate software's resiliency against bit flips, fault injection can simulate the outcome of such events at the software level.

We present a fault injection engine using Java PathFinder (JPF) [2] to systematically inject and explore bit-flip faults in the user specified variables in Java programs. Specifically, the users can specify a list of variables and for each variable v_i specify k_i, the number of bits to flip in it. Considering that any k_i bits of v_i are possible to be flipped in real hardware faults, we let JPF to execute the program in all possible ways to systematically evaluate programs' resiliency to bit-flip faults. For a simple example, consider the code in Figure 1. We want to know what happens if some error causes a bit to be flipped in the argument to method `foo`. Since `bar` is of type `int`, our implementation explores all the 32 cases in which bit is flipped, so the expected output of calling `foo(0)` is `1 2 4...-2147483648`.

Our implementation provides a `BitFlipListener`, a JPF listener that monitors the list of user-specified variables and performs bit-wise fault injection before the relevant instructions execute. Specifically, we support injecting bit-flip faults to three kinds of variables, (1) static and instance fields, (2) method arguments, and (3) local variables, whose type can be any primitive data types. For the fields and local variables, the bit flips are injected when they are written by the programs. For the method arguments, the bit flips are injected when the method is invoked and the arguments are assigned. `BitFlipListener` registers a Choice Generator to inject all possible bit flips to the corresponding operand in the operand stack before the store/write instruction or the invoke instruction, depending on the variable type. The users can specify the variables to flip in three ways: (1) calling `getBitFlip` API in the application code, (2) adding `@BitFlip` annotation to the variables, and (3) specifying in the command line arguments without changing the application code. For example, in Figure 1, we can (1) add `bar=getBitFlip(bar,1)` at the beginning of the method `foo`, (2) annotate `bar` with `@BitFlip(1)` (where `(1)` can be omitted because $k = 1$ by default), or (3) specify bit-flip fault injection in method `foo`, parameter `bar`, and $k = 1$ in the command line arguments. The `getBitFlip` API implementation is based on `Verify.getInt`. The `BitFlipListener` parses the annotations and the command line arguments and add the specified variables to a watch list when its instance is created. As a common practice, if the `@BitFlip` annotation and the command line specification are both present for a variable, the command line arguments take precedence.

```
public static void foo(@BitFlip int bar) {
  System.out.println(bar);
}
```

Figure 1: A simple example

A key challenge in the implementation is that JPF cannot register several choice generators at the same point of the application code. We resolve this issue by registering only one choice generator even when the number of bits to flip, k, in a variable is $k > 1$. Specifically, we register only one `IntIntervalGenerator` that produces an integer m in range $[0, \binom{n}{k})$ where n is the number of bits of the variable, and then decode the integer using binomial coefficients to get the set of k in n bits to flip. The decoding process is as follows: Because $\binom{n-1}{k}$ out of $\binom{n}{k}$ combinations do not select the n^{th} bit, if $m > \binom{n-1}{k}$, we select the n^{th} bit, let $m' = m - \binom{n-1}{k}$ and $k' = k - 1$; otherwise, we do not select the n^{th} bit and let $m' = m, k' = k$. If we then let $n' = n - 1$, with the same process on n', m', k', we can decode out the set of k' in the remaining n' bits, and then recursively get all the k bits to flip.

We implement a JPF regression test class that checks our injection engine in various scenarios and documents the basic usage. It verifies all bits are flipped exactly once in a variable using a global counter in JPF level. Besides, we use our tool to check the resiliency of Cyclic Redundancy Check (CRC) and International Standard Book Number (ISBN) algorithms against bit-flip faults. We confirm that both algorithms can detect all one-bit flips, but cannot detect all two-bit flips as expected. Our implementation has been included in JPF [1].

2. REFERENCES

[1] `https://github.com/javapathfinder/jpf-core/pull/295`.
[2] W. Visser, K. Havelund, G. Brat, and S. Park. Model checking programs. *Springer ASE-J*, 10, 2003.