T-Fuzz: fuzzing by program transformation

概要

为了增加程序的代码覆盖率，以往的做法是启发式变异输入样本，以达到更高的代码覆盖率。

本文采用的方法是去除掉Target-Program中的sanity checks（即在fuzz过程中很难生成样本通过的校验指令），生成新的Transform-Program，然后对其进行fuzz，以达到更高的代码覆盖率。

这样做有两个比较难解决的问题：①对移除sanity check后的程序进行fuzz，得到的crash结果很可能会是false positive。②即使是真实的bug，在Transform-Program中生成的crashing-input，不一定能在original-Program中触发bug。（不太明白这两点有什么区别？）。T-Fuzzer用一种基于符号执行的方法来过滤掉false positive，并且利用true positive的crashing input在original program中复现crash。

（对比）T-Fuzzer和Driller的目标都是要提高fuzz时的代码覆盖率，而且都使用了符号执行技术。不同之处在于他们使用符号执行的时间，Driller使用符号执行是在fuzzer之前，遇到一个严格校验，先通过符号执行计算出能通过这个校验的input，然后再去fuzzer；而T-Fuzzer则是先将校验去掉，程序就可以直接进入这片代码，等发现了bug再调用符号执行解决路径的问题。这两种方法都是有各自的道理，首先Driller，比较严谨，先解决路径问题，之后得到的bug一定是true positive，一定能复现；而T-Fuzzer，先把严格校验去掉，进入那段之前从未进入过的代码区，fuzz出bug之后再调符号执行来解决路径和bug触发约束条件的问题，这样提高了fuzzer的效率（只有在可能出现bug时才调用符号执行，很大程度上减少了符号执行调用的次数，不见兔子不撒鹰），但是得到了crashing-input之后还得符号执行判断true/false positive，即使true positive，还得调用符号执行得到能触发原始程序crash的input。个人主观觉得T-Fuzzer的方法理论上更好。

T-Fuzzer Design

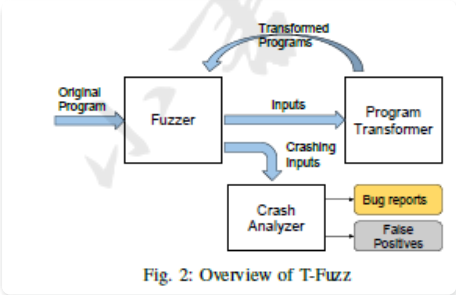

Fig. 2: Overview of T-Fuzz

Overview of T-Fuzz

- 识别NCC（Non-Critical Check）：fuzzer中未能通过的check都被认为是NCC。

Algorithm 2: Detecting NCC candidates

Input: *program*: The binary program to be analyzed
Input: CE: cumulative edge coverage
Input: CN: cumulative node coverage

```
cfg ← CFG(program)
NCC ← ∅
for e ∈ cfg.edges do
    if e ∉ CE ∧ e.source ∈ CN then
        NCC ← NCC ∪ {e}
```

Output: NCC: detected NCC candidates

enter description here

- 删除NCC：将Original-Program中的NCC移除，得到Transformed-Program-xxx。je->jne、ja->jna、jb->jnb...

Algorithm 4: Transforming program

Input: *program*: the binary program to transform
Input: *c_addrs*: the addresses of conditional jumps negated in the input program
Input: *NCC*: NCC candidates to remove

```
transformed_program ← Copy(program)
for e ∈ NCC do
    basic_block ← BasicBlock(transformed_program, e.source)
    for i ∈ basic_block do
        if i is a conditional jump instruction and i.addr ∉ c_addrs then
            negate_conditional_jump(program, i.addr)
            c_addrs ← c_addrs ∪ {i.addr}
            break
```

Output: *transformed_program*: the generated program with NCC candidate disabled
Output: *c_addrs*: the locations modified in the transformed program

enter description here

- fuzz：对Transformed-Program-xxx进行模糊测试，得到使其奔溃的输入样本T-crashing-input。
- Crash-Analyzer自动过滤false-positive 的T-crashing-input：利用符号执行，生成在Original-Program中触发此漏洞的路径约束条件（path constraints）和奔溃发生条件（crashing-condition），判断两个条件是否能同为真，是则true；否则false-positive，删除掉。

Algorithm 5: Process to filter out false positives

Input: *transformed_program*: the transformed program
Input: *c_addrs*: addresses of negated conditional jumps
Input: *input*: the crashing input
Input: *CA*: the crashing address

```
PC ← make_constraint(input)
CT ← PC
CO ← ∅
TC, addr ← preconstraint_trace(transformed_program, CT, entry)
while addr ≠ CA do
    if addr ∈ c_addrs then
        CO ← CO ∪ ¬TC
    else
        CO ← CO ∪ TC
    TC, addr ← preconstraint_trace(transformed_program, CT, i)
CO ← CO ∪ extract_crashing_condition(TC)
result ← SAT(CO)
```

Output: *result*: A boolean indicating whether *input* is a false positive
Output: *CO*: The set of constraints for generating the inputs in the original program

enter description here

- 生成O-crashing-input：利用上一步中的路径约束条件和奔溃发生条件来生成使Original奔溃的O-crashing-input。

实验结果

1. CGC测试集上与AFL、Driller的效果比较：

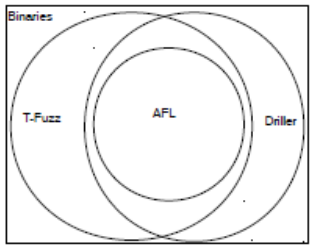

Fig. 5: Venn diagram of bug finding results

TABLE I: Details of experimental results

Method	Number of Binaries
AFL	105
Driller	121
T-Fuzz	166
Driller\AFL	16
T-Fuzz\AFL	61
Driller\T-Fuzz	10
T-Fuzz\Driller	45

enter description here

2. LAVA-M测试集上与FUZZER、SES、VUzzer、Steelix的效果比较：

TABLE II: LAVA-M Dataset evaluation results

program	Total # of bugs	FUZZER	SES	VUzzer	Steelix	T-Fuzz
base64	44	7	9	17	43	43
md5sum	57	2	0	1	28	49
uniq	28	7	0	27	24	26
who	2136	0	18	50	194	63

enter description here

3. 四个应用程序的测试效果：

Program	AFL	T-Fuzz
pngfix + libpng (1.7.0)	0	11
tiffinfo + libtiff (3.8.2)	53	124
magick + ImageMagick (7.0.7)	0	2*
pdftohtml + libpoppler (0.62.0)	0	1*

enter description here

4. 由于符号执行约束求解器的算力有限，过滤crashing-input时，会出现False-Negative（即False-False-Positive）的情况：T-crashing-input在Transform-Program中触发的漏洞，在Original-Program中真实存在，但是由于约束求解器的求解能力有限，不能算出O-crashing-input，从而被误判为false-positive。

- Alerts：T_Fuzz对Tansform-Program模糊测试发出的alert数目，相当于T-crashing-input的数目
- True Alerts：Alerts中，正确的alert数目，即T-crashing-input中能生成O-crashing-input的数目
- Report-Alerts：T_Fuzz最终上报的alert数目，相当于T-Fuzz最终生成O-crashing-input的数目

注：True Alerts数目与Reported Alerts数目不等的情况即是False-Negative造成的

TABLE IV: A sampling of T-Fuzz bug detections in CGC dataset, along with the amount of false positive alerts that were filtered out by its Crash Analyzer.

Binary	# Alerts	# True Alerts	# Reported Alerts	% FN
CROMU_00002	1	1	0	100%
CROMU_00030	1	1	0	100%
KPRCA_00002	1	1	0	100%
CROMU_00057	2	1	1	0
CROMU_00092	2	1	1	0
KPRCA_00001	2	1	1	0
KPRCA_00042	2	1	1	0
KPRCA_00045	2	1	1	0
CROMU_00073	3	1	1	0
KPRCA_00039	3	1	1	0
CROMU_00083	4	1	1	0
KPRCA_00014	4	1	1	0
KPRCA_00034	4	1	1	0
CROMU_00038	5	1	1	0
KPRCA_00003	6	1	1	0

enter description here

TABLE V: T-Fuzz bug detections in LAVA-M dataset, along with the amount of false positive alerts that were filtered out by its Crash Analyzer.

Program	# Alerts	# True Alerts	# Reported Alerts	% FN
base64	47	43	40	6%
md5sum	55	49	34	30%
uniq	29	26	23	11%
who	70	63	55	12%

enter description here

上面两个表的数据可以看出T-Fuzz的一个优点：产生的Alerts与True Alerts的数目比值为2.8（CGC），说明即使没有Crash-Analysis自动分析，使用人工分析，只需要每分析大概三个Alerts就可以得到一个真实的bug，相比于其他fuzz工具，优势非常明显