Type-Centric Kotlin Compiler Fuzzing: Preserving Test Program Correctness by Preserving Types

D. Stepanov, M. Akhin, M. Belyaev

14 april 2021

POLYTECH
Peter the Great
St.Petersburg Polytechnic
University

Problem

- All compilers contains bugs
- Relatively new programming language Kotlin compiler — not an exclusion

Main problem in compiler fuzzing

How to generate random, non-trivial and semantically valid program to test the compiler?

Existing methods

- Grammar-aware generation “CSmith” approaches
  - CSmith
  - YARPGen
  - jsfunfuzz
  - ...

Existing methods

- Mutation approaches
  - jsfunfuzz
  - ...
  - skeletal program enumeration

Skeletal program enumeration

Program P:

```
var a: Int = 1
var b: Int = 1

while (b < 100) {
    val c = b
    b = a + b
    a = c
}
```

Skeleton P':

```
var [_]: Int = 1
var [_]: Int = 1

while ([_] < 100) {
    val [_] = [_]
    [_] = [_] + [_]
    [_] = [_]
}
```

Skeletal program enumeration

Skeleton P':

```
var [_]: Int = 1
var [_]: Int = 1

while ([_] < 100) {
    val [_] = [_]
    [_] = [_] + [_]
    [_] = [_]
}
```

Produced program P1:

```
var a: Int = 1
var c: Int = 1

while (a < 100) {
    val b = a
    c = a + c
    a = c
}
```

200+ GCC/Clang bugs

Type-centric fuzzing

- Inspired by skeletal program enumeration
- Placeholders — expressions
- Fill placeholders by compatible typed generated expressions

SPE:

```
val a: Int = 1
```

↓

```
val [_]: Int = 1
```

TCE:

```
val a: Int = 1
```

↓

```
val a: Int = [Int]
```

Type centric fuzzing example

Program P:

```
class A(val a: Int)

fun f(): Int {
    ...
}

var a: Int = 1
var b: Int = 1

while (b < 100) {
    val c = b
    b = a + b
    a = c
}
```

Skeleton P':

```
class A(val a: Int)

fun f(): Int {
    ...
}

var a: Int = [Int]
var b: Int = [Int]

while ([Int] < [Int]) {
    val c = [Int]
    [Int] = [[Int] + [Int]]
    [Int] = [Int]
}
```

Type centric fuzzing example

Skeleton P':

```
class A(val a: Int)

fun f(): Int {
    ...
}

var a: Int = [Int]
var b: Int = [Int]

while ([Int] < [Int]) {
    val c = [Int]
    [Int] = [[Int] + [Int]]
    [Int] = [Int]
}
```

Produced program P1:

```
class A(val a: Int)

fun f(): Int {
    ...
}

var a: Int = -100
var b: Int = a * f()

while (a < b) {
    val c = 12345
    b = A(f()).a
    b = b
}
```

Type-centric fuzzing scheme

- Generate expressions to fill the placeholders (generation phase)
- Fill skeleton by generated expressions (mutation phase)

Generation phase

Given:

- Set of variables V
- Available callables C

```
class A(                                    //c1
    val a: Int                              //c2
) {
    fun f(a: String): Int { ... }           //c3
}

val a: Int = 1                              //v1
```

Generation phase

We need to generate set of typed expressions using the C:

```
class A(                                    //A(1) -> A
    val a: Int                              //A(1).a -> Int
) {
    fun f(a: String): Int { ... }           //A(1).f("") -> Int
}
```

Generation phase algorithm

INPUT: *file* with a seed program P

OUTPUT: list of generated expressions $c_0 ... c_N$

```
1:  function GENERATIONPHASE(file)
2:      res ← []
3:      for callee ∈ getCallables(file) do
4:          if callee is Class then
5:              instance ← genClassInstance(callee, file)
6:              for iCallee ∈ getCallables(instance) do
7:                  res ← res + genCall(iCallee, file)
8:              end for
9:          else
10:             res ← res + genCall(callee, file)
11:         end if
12:     end for
13:     return res
14: end function
```

Class instances generation algorithm

```
1:  function genClassInstance(class, file)
2:      typeParams ← genTypeParams(class, file)
3:      class ← parameterize(class, typeParams)
4:      if ¬hasOpenConstructor(class) then
5:          impl ← findImplementation(class)
6:          if impl ≠ null then
7:              impl ← adaptTypeParams(impl, class, typeParams)
8:              return genClassInstance(impl, file)
9:          else
10:             return null
11:         end if
12:     end if
13:     randomCtor ← getRandomConstructor(class)
14:     return genConstructorCall(randomCtor, file)
15: end function
```

Mutation phase

Given:

- Set of generated on previous step expressions E
- Seed program for mutation phase

```
// A(1) -> A                | e1
// A(1).a -> Int            | e2
// A(1).f("") -> Int        | e3
// a -> Int                 | e4

fun factorial(n: Int): Double {
    var result = 1.0
    for (i in 1..n) {
        result *= i
    }
    return result
}
```

Mutation phase

Need to make type placeholders and fill it by $e \in E$

```
fun factorial(n: Int): Double {
    var result = [Double]
    for (i in [IntRange]) {
        [Double] *= [Int]
    }
    return [Double]
}
```

Mutation phase example

```
val a: Int = 1

class A(val a: Int) {
    fun f(a: String): Int { ... }
}

fun f(a: Int): Int { ... }

fun factorial(n: Int): Double {
    var result = f(1).toDouble()
    for (i in A(1).f("")..a) {
        result *= i
    }
    return result
}
```

How to choose expression to fill?

- Select a type-compatible expression from E
- Create a random value of a built-in type (int, bool, etc.)
- Generate a valid call to the standard library

Generation of calls to the standard library

```
                    // a -> Int   |   [Double]

n = 1:
fun and(other: Int): Int
fun dec(): Int
...
fun toDouble(): Double
fun minus(other: Double): Double

n = 2:
fun toByte(): Byte
fun times(other: Double): Double

n = 3:
fun toFloat(): Byte
fun compareTo(other: Int): Int
fun plus(other: Double): Double

...

Generated:
a.toDouble()
a.minus(0.0)
a.toByte().times(2.2)
a.toFloat().compareTo(1).plus(0.0)
```

Mutation phase algorithm

INPUT: seed program from generation phase *gen*
INPUT: generated expressions *exprs*
INPUT: seed program for mutation phase *seed*
OUTPUT: program will filled type placeholders

```
1: function MUTATIONPHASE
2:     anon ← anonymize(seed)
3:     anon ← merge(anon, gen)
4:     for ph ∈ getPlaceholders(anon) do
5:         e ← genPhExpr(ph, exprs)
6:         anon ← replacePhWithExpr(anon, ph, e)
7:     end for
8:     return anon
9: end function
```

Generate phi expression algorithm

```
1:  function genPhExpr(ph, exprs)
2:      type ← getType(ph)
3:      r ← []
4:      r ← r + genRandomValue(type)
5:      r ← r + genStdLib(type)
6:      for e ∈ exprs do
7:          eType ← getType(ph)
8:          if compatible(type, eType) then
9:              r ← r + e
10:         end if
11:     end for
12:     return random(r)
13: end function
```

Mutation phase?

Mutation phase can be applied many times:

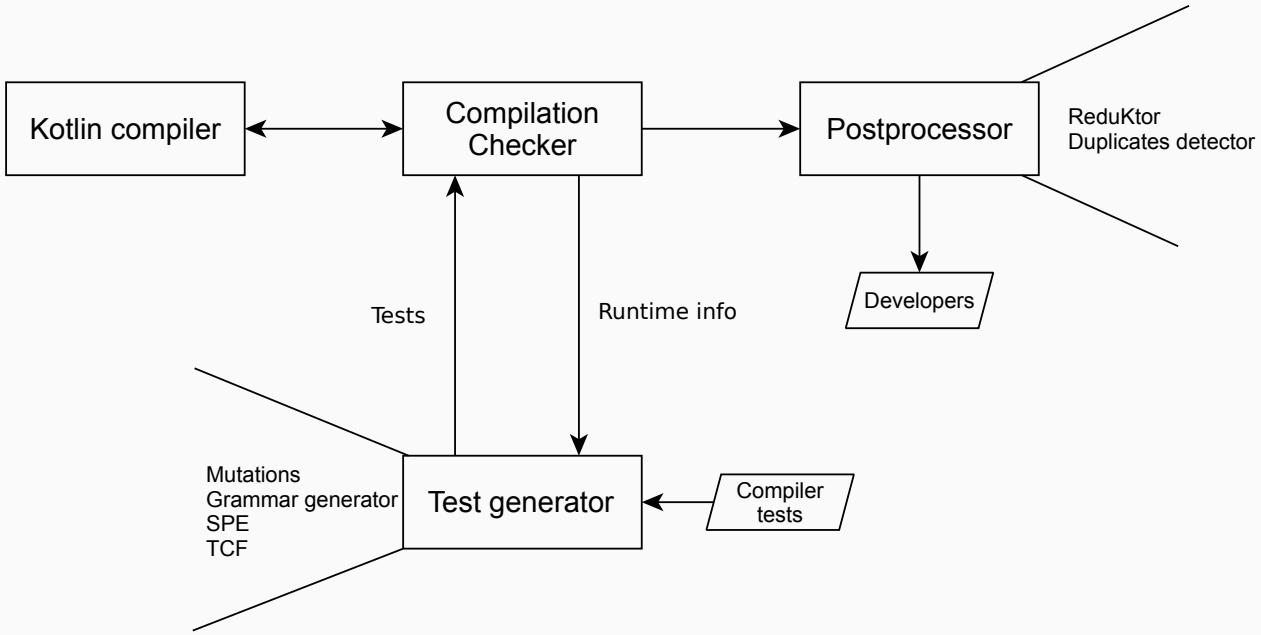
```
[List<Int>]

↓

listOf(1, 2, 3)

↓

listOf([Int], [Int], [Int])
```

TCE problems

- The generation process could potentially never terminate
- With iterative TCE run time could increase nonlinearly
- Long chains of callables

Implementation

We have implemented our approach inside a tool for Kotlin compiler fuzzing named Backend Bug Finder (BBF):



Postprocessing

Evaluation

Interesting bugs found

Discussion?

Conclusion

Contact information

{stepanov, akhin, belyaev}@kspt.icc.spbstu.ru