How fuzzing can help improve compiler quality

Stepanov
Daniil

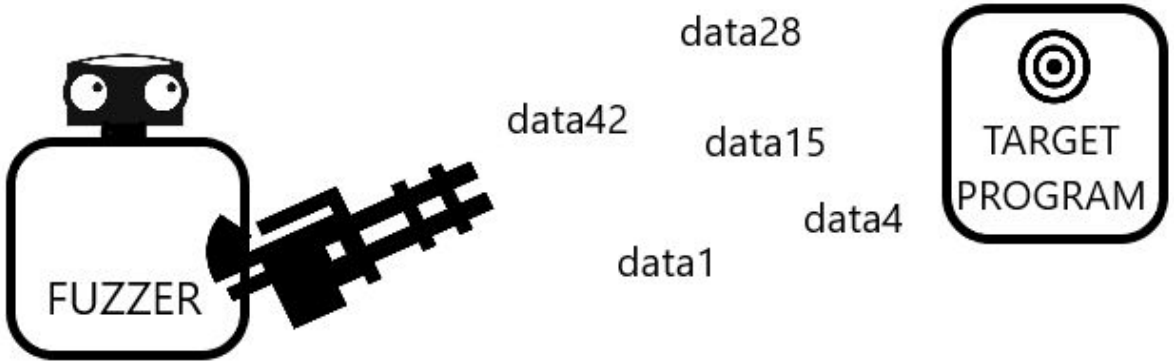
POLYTECH
Peter the Great
St.Petersburg Polytechnic
University

Problem

- All programs contains bugs
- Compilers are not an exclusion
  - Clang, GCC, javac, etc. bug trackers contains tens of thousands issues
- Manual testing is not enough
- Is there any way to improve compilers quality?

Fuzzing

“Program that generates a stream of random characters to be consumed by a target program”, Barton Miller et al., 1988



Fuzzers diversity

Black-box:

- Working with I/O of program

Grey-box:

- Gets partial information from the target program

White-box:

- Dynamic symbolic execution + coverage-maximizing heuristic

Disadvantages:

- Completeness

Disadvantages:

- Trying to find a balance between black-box and white-box fuzzing

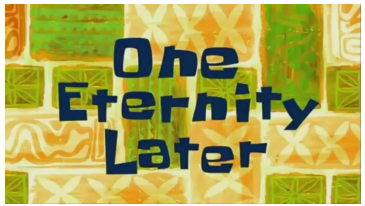
Disadvantages:

- Implementation complexity
- All problems of symbolic execution

Black-box fuzzing example

```
fun getIthEl(a: Int, i: Int): Int {
    val list = listOf(1, 2, 3)
    return if (a == 1) {
        if (i > 0) {
            list[i]
        } else -1
    } else -1
}
```


getIthEl(-1, 0)
getIthEl(1, 2)
getIthEl(-100, 21545135)
getIthEl(125123590, 5123553)

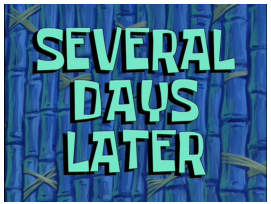


getIthEl(1, 123)

Grey-box fuzzing example

```
fun getIthEl(a: Int, i: Int): Int {
  val list = listOf(1, 2, 3)
  return if (a == 1) {
    if (i > 0) {
      list[i]
    } else -1
  } else -1
}
```


getIthEl(-1, 0)
getIthEl(-1, -2142135)
getIthEl(-125, 352)
getIthEl(1, 0)



getIthEl(1, 1)
getIthEl(1, 123)

White-box fuzzing example

```
fun getIthEl(a: Int, i: Int): Int {
  val list = listOf(1, 2, 3)
  return if (a == 1) {
    if (i > 0) {
      list[i]
    } else -1
  } else -1
}
```

```
 @S %0 = new java/lang/Integer[3]
 @S %1 = java/lang/Integer.class.valueOf(1)
 @S *(%0[0]) = %1
 @S %2 = java/lang/Integer.class.valueOf(2)
 @S *(%0[1]) = %2
 @S %3 = java/lang/Integer.class.valueOf(3)
 @S *(%0[2]) = %3
 @S %4 = kotlin/collections/CollectionsKt.class.listOf(%0)
 @S %5 = arg$0 != 1
) -> (BEGIN
 <OR> (
  @P %5 = false
  @S %6 = arg$1 <= 0
) -> (
  @P %6 = false
  @S %7 = %4.get(arg$1)
  @S %8 = (%7 as java/lang/Number)
  @S %9 = %8.intValue()
  @S %10 = %9
),  <OR> (
  @P %5 = false
  @S %6 = arg$1 <= 0
) -> (
  @P %6 = true
  @S %10 = -1
),  <OR> (
  @P %5 = true
  @S %10 = -1
) END) -> (
  @S <retval> = %10
```

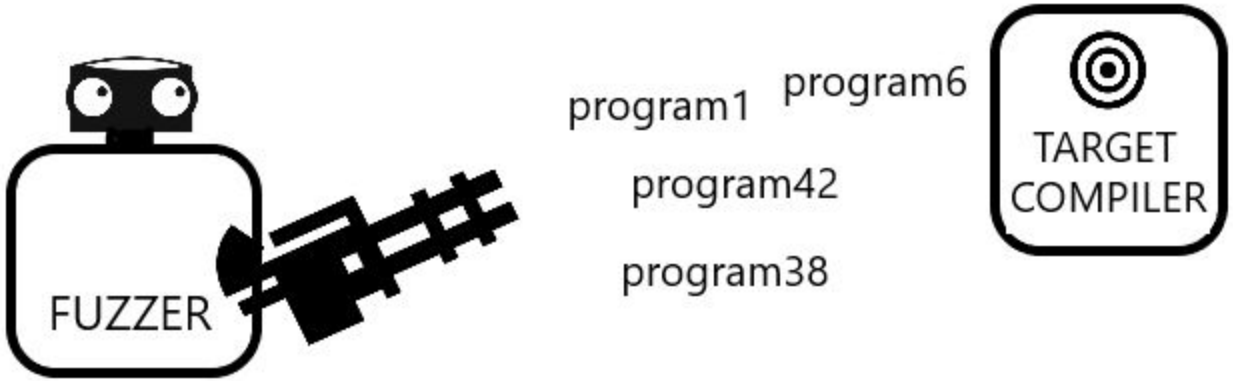
```
[DEBUG] - Args: [2, 123]
[DEBUG] - Collected trace: (
  @S %0.1 = new java/lang/Integer[3]
  @S %1.2 = java/lang/Integer.class.valueOf(1)
  @S *(%0.1[0]) = %1.2
  @S %2.3 = java/lang/Integer.class.valueOf(2)
  @S *(%0.1[1]) = %2.3
  @S %3.4 = java/lang/Integer.class.valueOf(3)
  @S *(%0.1[2]) = %3.4
  @S %4.5 = kotlin/collections/CollectionsKt.class.listOf(%0.1)
  @S %5.6 = arg$0 != 1
  @P %5.6 = true
  @S %10.7 = -1
  @S true = true
)
```

White-box fuzzing example

```
[DEBUG] - Args: [2, 3]
[DEBUG] - Collected trace: (
  @S %0.1 = new java/lang/Integer[3]
  @S %1.2 = java/lang/Integer.class.valueOf(1)
  @S *(%0.1[0]) = %1.2
  @S %2.3 = java/lang/Integer.class.valueOf(2)
  @S *(%0.1[1]) = %2.3
  @S %3.4 = java/lang/Integer.class.valueOf(3)
  @S *(%0.1[2]) = %3.4
  @S %4.5 = kotlin/collections/CollectionsKt.class.listOf(%0.1)
  @S %5.6 = arg$0 != 1
  @P %5.6 = true
  @S %10.7 = -1
  @S true = true
)
```

Compiler fuzzing

“Program that generates a random programs to be consumed by a compiler”



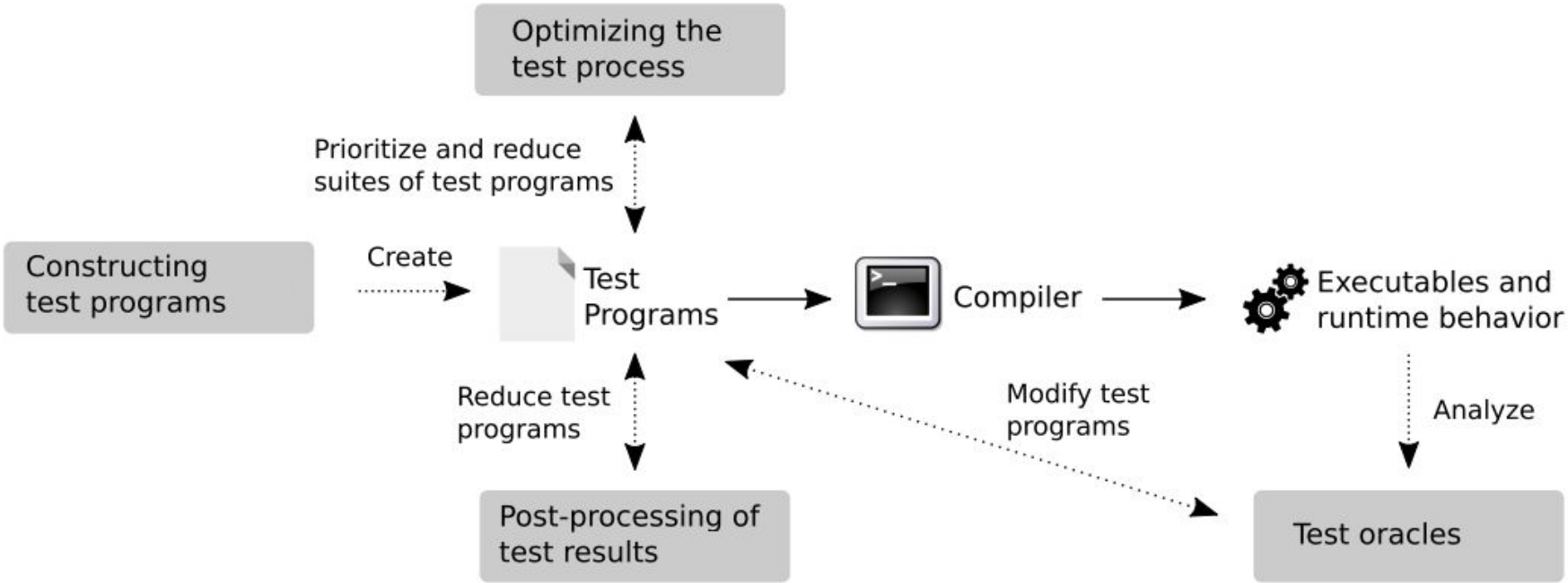
Compiler fuzzing

- Separate research area
- Started to develop in 1972
- Which language compilers are tested the most?
  - Javascript
  - C/C++
- Biggest complexity - input program generation

Fuzzers diversity

Language	Approach
C/C++	Eide and Regehr [47], Yang et al. [123], Nagai et al. [87], Nagai et al. [88], Lindig [78, 79], Groce et al. [51], Le et al. [72], Le et al. [73], Morisset et al. [85], Zhang et al. [131], Sun et al. [111], Amodio et al. [4], Alipour et al. [3]
JavaScript	Holler et al. [58], Groce et al. [49], Patra and Pradel [93], Bastani et al. [8]
PL/I	Hanford [52]
Ada	Duncan and Hutchison [45], Austin et al. [6], Mandl [80]
Fortran	Callahan et al. [17], Dongarra et al. [44], Burgess and Saidi [16]
Java	Sirer et al. [108], Boujarwah et al. [14], Yoshikawa [125], Chen et al. [31]
Algol	Houssais [61]
APL	Ching and Katz [32]
Arden	Wolf et al. [120]
Haskell	Palka et al. [92]
Lusture	Garoche et al. [48]
mpC	Kalinov et al. [64, 65]
OCaml	Midtgaard et al. [83]
Pascal	Burgess [34], Bazzichi and Spadafora [9]
PLZ/SYS	Bazzichi and Spadafora [9]
Python	Bastani et al. [8]
Ruby	Bastani et al. [8]
Rust	Dewey et al. [41]
Scala	Zhang et al. [131]
GLSL	Donaldson et al. [42]

Compiler fuzzing problems



Methods of input programs construction

- Generational approach
  - Grammar-directed
  - Grammar-aided
  - Others
- Mutational approach
  - Semantics-preserving
  - Non-semantics-preserving

Grammar-directed approach

Purdom P. A sentence generator for testing parsers //BIT Numerical Mathematics. – 1972. – Т. 12. – №. 3. – С. 366-375.

Example of modern grammar-directed approach (LaLa specification*)

```
use SymbTab;

class Program {
  prog("${stmts : StmtList}\n") {
    stmts.symbs = (SymbTab:empty);
  }
}
@list
class StmtList {
  inh symbs : SymbTab;
  one("${s: Stmt}") {
    stmt.s_before = this.symbs;
  }
  @weight(3)
  mult("${s: Stmt}\n${r : StmtList}") {
    s.s_before = this.symbs;
    r.symbs = s.s_after;
  }
}
```

```
class Stmt {
  inh s_before : SymbTab;
  syn s_after : SymbTab;
  assign("${i : Identifier} := ${e : Expr};") {
    e.symbs = this.s_before;
    this.s_after = (SymbTab:put this.s_before i.str);
  }
}
class Expr {
  inh symbs : SymbTab;
  num("${val : Number}") {}
  use_var("${var : UseVariable}") {
    var.symbs = this.symbs;
  }
  binop("(${l : Expr}) ${op : Op} (${r : Expr})") {
    l.symbs = this.symbs;
    r.symbs = this.symbs;
  }
}
```

```
class UseVariable {
  inh symbs : SymbTab;
  grd defined;
  var("${var : Identifier}") {
    this.defined =
      (SymbTab:contains this.symbs var.str);
  }
}

# alternatively:
class UseVariableGen {
  inh symbs : SymbTab;
  var(SymbTab:defs this.symbs) : String {}
}

class Identifier("[a-z]{1,3}");
class Number("0|[1-9][0-9]{0,2}");
class Op("+|-|*|/");
```

* P. Kreutzer, S. Kraus, and M. Philippsen, “Language-agnostic generation of compilable test programs,” in 2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST). IEEE, 2020, pp. 39–50.

Grammar-aided approaches

- CSmith
  - Generator of programs on a subset of the C language
  - Uses grammar + a lot of analyses to ensure test validity

Problem	Code-Generation-Time Solution	Code-Execution-Time Solution
use without initialization	explicit initializers, avoid jumping over initializers	—
qualifier mismatch	static analysis	—
infinite recursion	disallow recursion	—
signed integer overflow	bounded loop vars	safe math wrappers
OOB array access	bounded loop vars	force index in bounds
unspecified eval. order of function arguments	effect analysis	—
R/W and W/W conflicts betw. sequence points	effect analysis	—
access to out-of-scope stack variable	pointer analysis	—
null pointer dereference	pointer analysis	null pointer checks

Table 1. Summary of Csmith's strategies for avoiding undefined and unspecified behaviors. When both a code-generation-time and code-execution-time solution are listed, Csmith uses both.

Other approaches

- A lot of approaches, implements different “local” ideas:
  - Testing certain features in compilers by writing a generator of the desired language constructions
  - Machine learning based
  - Based on the derivation of the rules for generating programs from projects in the target language
  - …

Mutation based approaches

- We are mutating existing part of program
  - Semantic-preserving
  - Non-Semantic-preserving

Semantics-preserving

- Metamorphic testing
  - We accidentally mutate a part of the program that is not covered at runtime
  - Insert “dead code”
  - Replacing expressions with equivalent ones

```
fun getIthEl(a: Int, i: Int): Int {
  val list = listOf(1, 2, 3)
  return if (a == 1) {
    if (i > 0) {
      list[i]
    } else -1
  } else -1
}
```

```
fun getIthEl(a: Int, i: Int): Int {
  val list = listOf(1, 2, 3)
  return if (a == 1) {
    if (i > 0) {
      if (i == 0) -1
      else list[i]
    } else -1
  } else -1
}
```

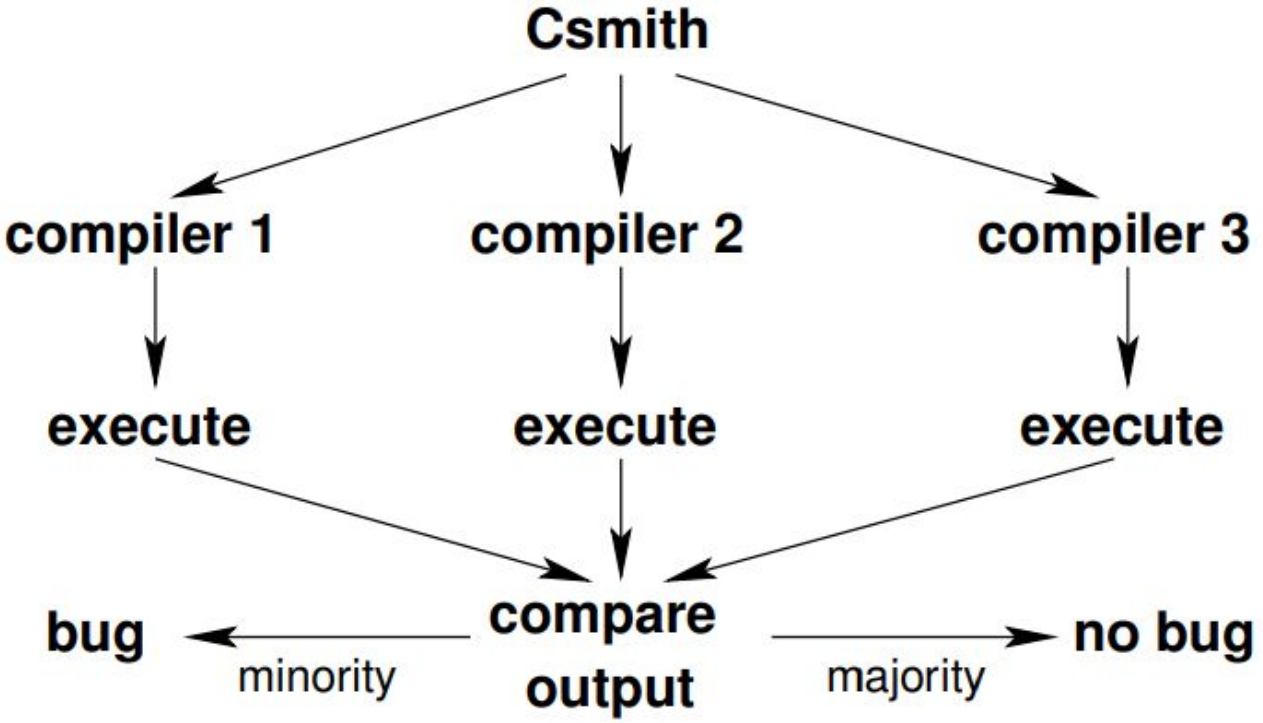
Non-semantics preserving

- Different mutations over different kinds of code representation
- If resulting test is incorrect, roll back

```
fun getIthEl(a: Int, i: Int): Int {
  val list = listOf(1, 2, 3)
  return if (a == 1) {
    if (i > 0) {
      list[i]
    } else -1
  } else -1
}
```

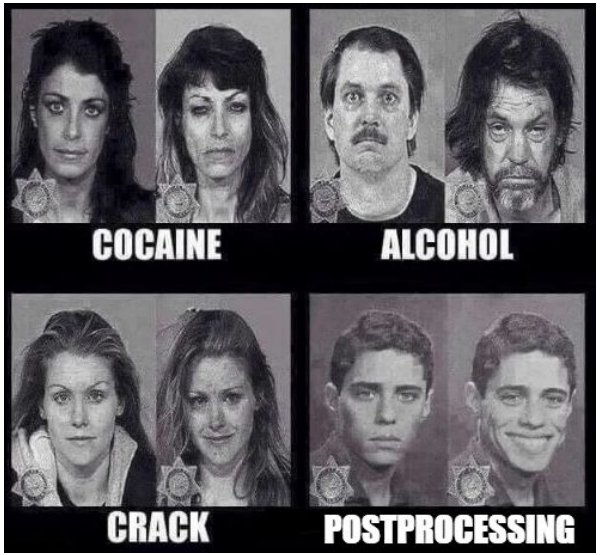
```
fun getIthEl(a: Int, i: Int): Int {
  val list = listOf(1, 2, "3")
  return if (a > 1) {
    if (i == 0) {
      list[i]
    } else -1
  } else -1
}
```

Test oracles problem. How to detect miscompilation?



Postprocessing

- Reduction
- Duplicates filtration
- Bug isolation



Reduction. Problem

```
open class A1(y: String) {
   val x = "[A1.x,$y]"
}
open class A2(y: String) {
   val x = "[A2.x,$y]"
   inner open class B1 : A1 {
       constructor(p: String) : super("[B1.param,$p]")
       fun foo() = x + ";" + this@A2.x + ";"
   }
   fun bar(): String {
       return with(A2("#bar")) {
           class C : B1("bar") {}
           C().foo()
       }
   }
   fun foo() = A2("#baz").baz()
   fun A2.baz(): String {
       class C : B1("baz") {}
       return C().foo()
   }
}
fun box(): String {
   val r3 = A2("f").bar()
   if (r3 != "[A1.x,[B1.param,bar]];[A2.x,#bar];") return "$r3:"
   val r4 = A2("gg").foo()
   if (r4 != "[A1.x,[B1.param,baz]];[A2.x,#baz];") return "$r:"
   return "OK"
}
```

Reduction. Problem

```
open class A1

class A2 {
   open inner class B1 : A1()

   fun A2.baz() {
       class C : B1()
   }
}
```

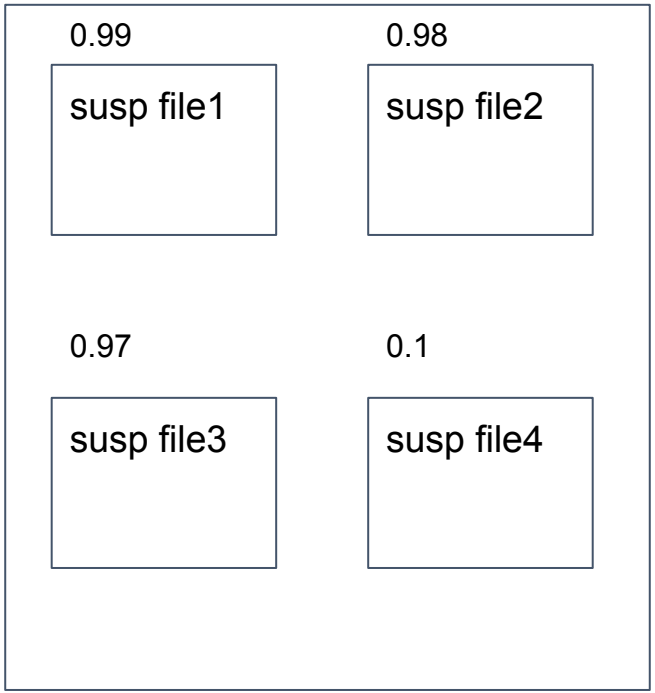
Reduction. Existing methods

- Language-agnostic methods
  - Delta-debugging
  - Slicing
  - Hierarchical delta-debugging
  - …
- Language-specific methods
  - Transformations over different types of code representation
- Different combinations of approaches
  - CReduce

Duplicates filtration. For what?

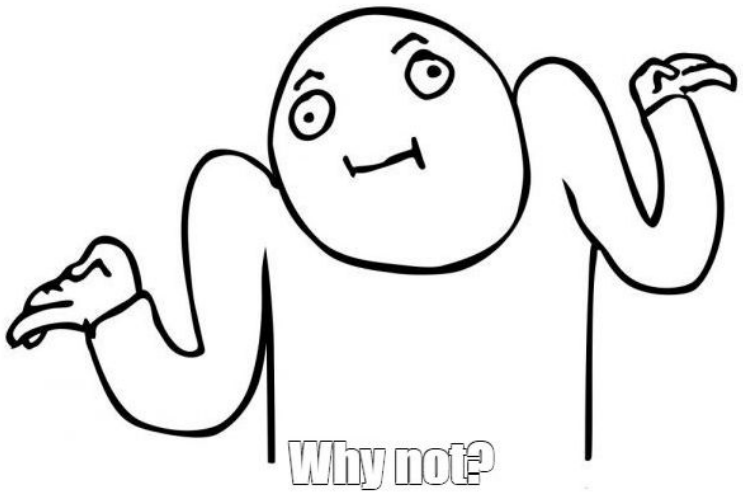
```
/*1*/ fun loop() { {}!!!! > 1 }

/*2*/ var a = {}!!!!

/*3*/ fun box() { -{}!!!! }

/*4*/ fun box() { ({}!!!! == 1).not() }

// …
```

Bug isolation



BBF(Backend bug finder). Origins

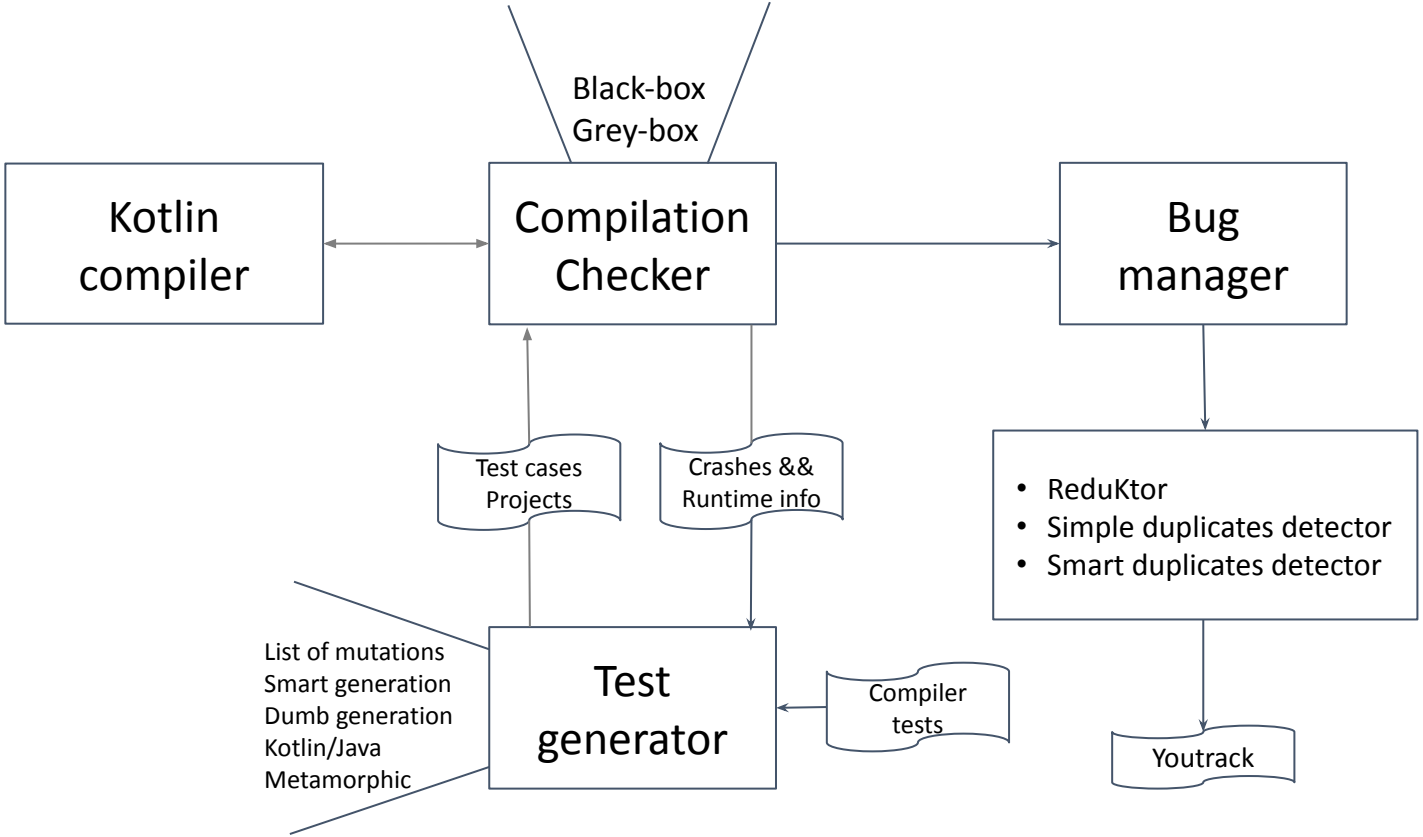
- 2017 - simple fuzzer
- 2018 - reducer
- 2018 -



What types of bugs can BBF find?

- Crashes
- Miscompilations
- Performance issues
- Bugs in plugin

BBF scheme


Black-box
Grey-box
Kotlin compiler
Compilation Checker
Bug manager
Test cases Projects
Crashes && Runtime info
• ReduKtor
• Simple duplicates detector
• Smart duplicates detector
List of mutations
Smart generation
Dumb generation
Kotlin/Java
Metamorphic
Test generator
Compiler tests
Youtrack

Why not generative approach

- The more different “features” in the language, the more difficult it is to generate it:
  - Smart cast-s
  - Tailrec
  - Null safety
  - Reflection
  - Interacting with libraries
  - …
- Either we generate only a subset of the language, or we spend a lot of time on each feature

Test generator

- 30+ mutators of programs
  - Adding an argument to a function
  - Changing modifiers
  - Replacing a node in AST with a node of the same type
- Type-centric enumeration
- Metamorphic testing
- Random data structures generator

Skeletal program enumeration

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

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

(a) P

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

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

(b) P'

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

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

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

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

* Zhang Q., Sun C., Su Z. Skeletal program enumeration for rigorous compiler testing //Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation. – 2017. – С. 347-361.

Type-centric fuzzing

```
class A {
    val a: Int
}

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

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

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

```
class A {
    val a: Int
}

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

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

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

(a) P

```
class A {
    val a: Int
}

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

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

while (a < b) {
    val c = 12345
    b = A(123).a
    b = b
}
```

(b) P'

```
class A {
    val a: Int
}

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

var a: Int = A(7).a
var b: Int = 3

while (b < 100) {
    val c = f(a * 10)
    b = c * (-10) + a
    a = b + c
}
```

Type-centric fuzzing (generation phase)

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:          else
9:              res ← res + genCall(callee, file)
10:     return res
11: end function
12:
13: function GENCLASSINSTANCE(class, file)
14:     typeParams ← genTypeParams(class, file)
15:     class ← parameterize(class, typeParams)
16:     if ¬hasOpenConstructor(class) then
17:         impl ← findImplementation(class)
18:         if impl ≠ null then
19:             impl ← adaptTypeParams(
20:                 impl, class, typeParams)
21:             return genClassInstance(impl, file)
22:         else
23:             return null
24:     randomCtor ← getRandomConstructor(class)
25:     return genConstructorCall(randomCtor, file)
26: end function
```

Fig. 4: Generation phase algorithm

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

interface B {
    val a: Int
}

class C(override val a: Int) : B

fun f(a: Int): Int

val a: Int = 1
```

(a) Seed program for generation phase

```
A(1) -> A
A(1).a -> Int
A(1).f("") -> Int
C(1) -> B
C(1) -> C
C(1).a -> Int
f(1) -> Int
a -> Int
```

(b) Generated expressions with their types

Fig. 3: Generation phase example

Type-centric fuzzing (mutation phase)

INPUT: seed program for mutation phase *seed*
INPUT: seed program from generation phase *gen*
INPUT: generated expressions *exprs*
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:      return anon
8:  end function
9:
10: function GENPHEXPR(ph, exprs)
11:     type ← getType(ph)
12:     r ← []
13:     r ← r + genRandomValue(type)
14:     r ← r + genStdLib(type)
15:     for e ∈ exprs do
16:         eType ← getType(ph)
17:         if compatible(type, eType) then
18:             r ← r + e
19:     return random(r)
20: end function
```

Fig. 5: Mutation phase algorithm

```
val a: Int = 1

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

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

// Generated expressions with their types:
//   A(1) -> A
//   A(1).a -> Int
//   A(1).f("") -> Int
//   f(1) -> Int
//   a -> Int
```

(a) Seed program after generation phase

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

(b) Seed program for mutation phase

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

(c) Typed skeleton

```
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
}
```

(d) Typed skeleton filled with the generated expressions

Evaluation

	NO SEVERITY	MINOR	NORMAL	MAJOR
Frontend	2	0	0	0
Backend	0	1	5	4
Miscompilation	0	0	2	4

TABLE I: Posted bug severity from the Kotlin YouTrack

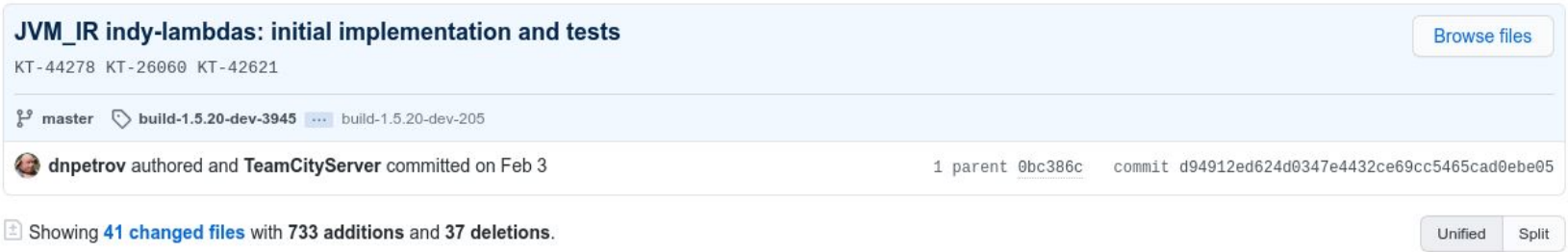
Results	M	G	EM	SPE	TCE	TCE + EM
Correct programs, %	11,9	0,05	10,7	24,2	63,4	13,4
Interesting bugs, %	25,0	0,0	20,0	100,0	100.0	16,6
Frontend crashes	3	212	4	0	0	5
Backend crashes	9	0	11	2	3	7
Miscompilations	0	0	0	0	3	0
Duplicates	49	77	38	1	4	14

TABLE II: Evaluation results

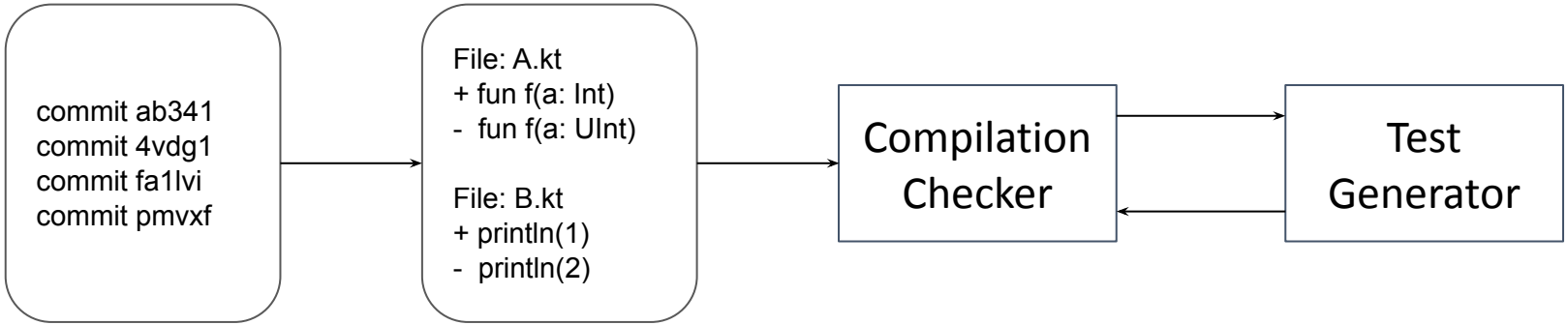
- (M) Mutation-based fuzzing;
- (G) Grammar based generation
- (EM) M + language-specific mutations
- (SPE) Skeletal program enumeration
- (TCE) Type-centric enumeration;
- (TCE + EM) TCE + language-specific mutations.

Black-box vs Grey-box target fuzzing

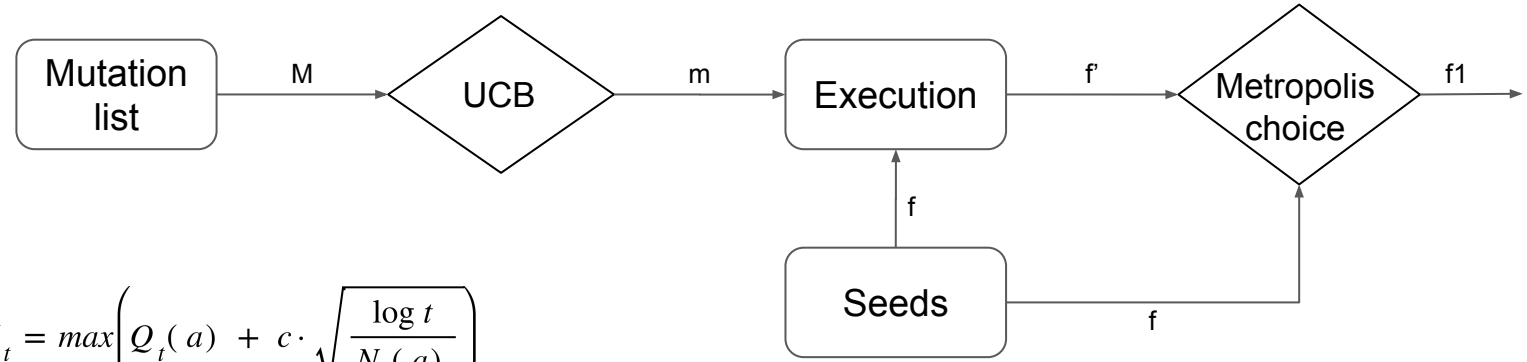
- Coverage is almost useless for standard testing
- Is the same also true for feature fuzzing?



Grey-box target fuzzing (main idea)



Grey-box target fuzzing (scheme)



$$M_t = max\left(Q_t(a) + c \cdot \sqrt{\frac{\log t}{N_t(a)}}\right)$$

$Q_t(a)$ – *average points*

t – *iteration*

$N_t(a)$ – *number of "a" selections*

$Q_t(a)$ – *exploitation*

$N_t(a)$ – *exploration*

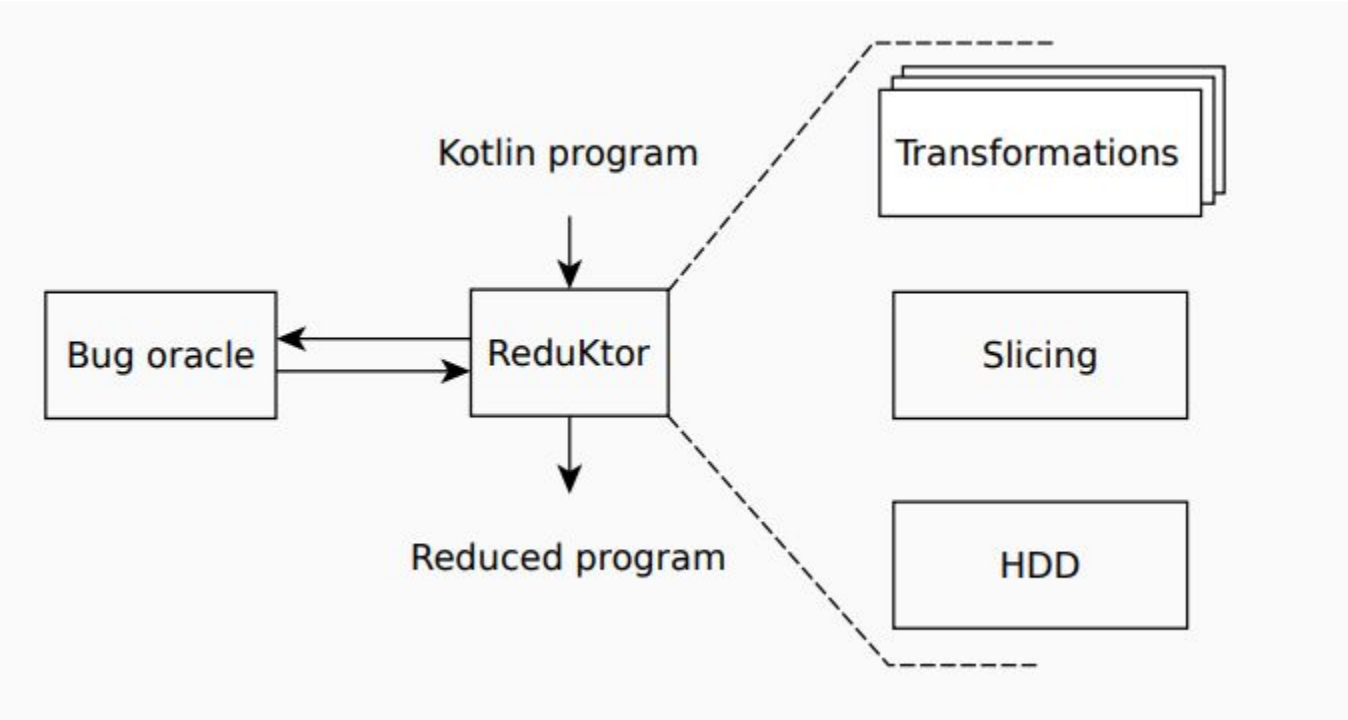
$$Score_{m_t} = \left(coverage(m_t) - coverage_{m_{t-1}}\right) \cdot K$$

$$A(f -> f_1) = min\left(1, exp\left(K \cdot \left(cov(f) - cov(f_1)\right)\right)\right)$$

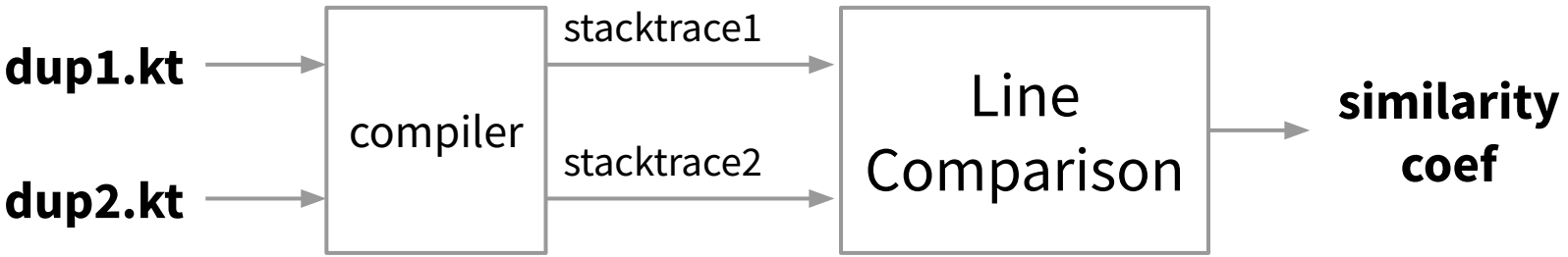
Postprocessors

- Reduction
- Duplicates filtration
- Bug isolation?

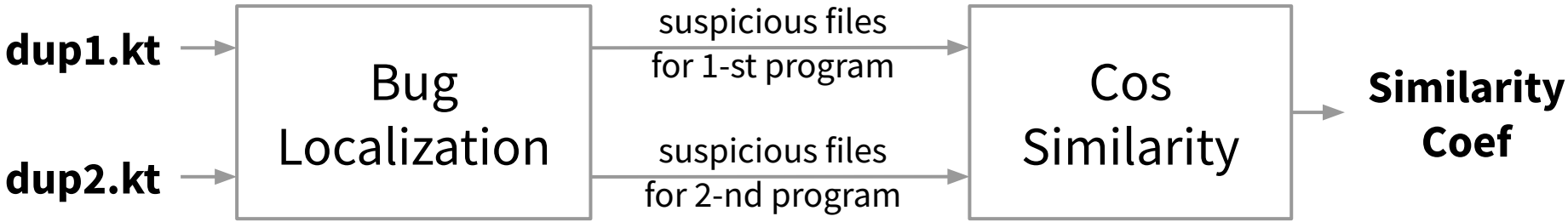
Scheme of ReduKtor



How duplicates filtration works?



Better duplicates filtration?



Results

- 213 bugs were posted found by BBF
  - 50 major
  - 39 normal
  - 9 minor
  - 88 not specified
- 83 fixed
  - 43 major
  - 20 normal
  - 5 minor
  - 15 not specified
- 34 open
- 6 in progress

How we helped to the Kotlin team?

- New backend ABI fuzzing
- Performance testing
- Fuzzing of new features
- Regressions search
- Kotlin plugin testing

Contacts

Email: stepanov0995@gmail.com
Repo: https://github.com/DaniilStepanov/bbfgradle