```
1 class T {
2   long a = 0;
3   int b() {
4     int d = 2, e = 10;
5     for (int m = 7; m < 149; ++m)
6       for (int c = 1; c < 4; c++)
7         for (int n = 1; n < 2; ++n)
8           d += e;
9     return d;
10  }
11  void f() {
12    for (int w = 0; w < 6361; w++)
13      a = b();
14  }
15  public static void main(String[] g) {
16    T t = new T();
17    for (int h = 1; h < 212; h++)
18      t.f();
19    System.out.println(t.a);
20  }
21 }
```

Figure 4. Issue-*****: Mis-compilation. The highlighted neutral loop is inserted by `Artemis`. Code shown in this example is a cleaned-up version from a very large test program.

A More Examples

`Artemis` is fruitful in finding diverse bugs such as segmentation faults (SIGSEGV), fatal arithmetic error (SIGFPE), emergency abort (SIGABRT), assertion failures, mis-compilations, and performance issues. We discuss a small selection to highlight the diversity. All tests shown in this section originate from JavaFuzzer [19] and are mutated by `Artemis`; they are reduced from much larger mutants with a combination of Perses, C-Reduce, and further manual cleanup if needed.

A.1 Illustrative OpenJ9 Example

Figure 4 triggers a mis-compilation in OpenJ9. `Artemis` found the bug in OpenJ9 0.32 (revision `3d06b2f9c`, based on OpenJDK 1.8.0_342). However, it can be reproduced at least as far back as 0.24 with JDK 11 and immediately marked as `blocker`, the most severe, release-blocking type of bugs.

The seed program assigns `T.a` by a value returned by `T.b()` repeatedly (Line 13). Because `T.b()` is pure, it should always return a fixed integer `4262` and the program output should be `4262` as well. For the seed program, the method counter of `T.b()` reaches the `warm`-level compilation threshold and `T.b()` is JIT-compiled at `warm` level; all other methods are barely interpreted until the program exits. `Artemis` alters the JIT compilation by inserting the highlighted loop into Line 12, which leads to an additional JIT compilation of `T.f()` at `warm` level and two additional OSR compilations of the inserted loop at `veryHot` and `scorching` levels, respectively. Considering `T.b()` constantly returns `4262`, the inserted loop is neutral and the insertion does not affect the seed's semantics. Therefore, the mutant should output `4262` as the seed. Yet, OpenJ9's JIT compiler mis-compiles the mutant and outputs `1422`.

```
1 class T {
2  public static void main(String[] g) {
3   for (int i=0; i<10; i+=1)
4    for (int j=1; j<197; j+=1)
5     for (int k=-4910; k<314; k+=1)
6      try {
7       boolean b = false;
8       byte[] a = new byte[1 << 14];
9       try (ByteArrayOutputStream o =
10                new ByteArrayOutputStream();
11           ZipOutputStream z =
12              new ZipOutputStream(o)) {
13        final byte[] x = { 'x' };
14        z.putNextEntry(new ZipEntry("a.gz"));
15        GZIPOutputStream g = new GZIPOutputStream(z);
16        g.write(x);  g.finish();
17        if (b) z.write(x);
18        z.closeEntry();
19        z.putNextEntry(new ZipEntry("b.gz"));
20        GZIPOutputStream k = new GZIPOutputStream(z);
21        k.write(x);  k.finish();
22        z.closeEntry();  z.flush();
23        a = o.toByteArray();
24       }
25      } catch (Throwable x) {}
26  }
27 }
```

Figure 5. JDK-*****: Performance Issue. When executing the C2 compiled code, the HotSpot process running this test is directly killed on Ubuntu while becomes much slower on Windows. Code shown in this example is a cleaned-up version from a very large test program.

The root cause is that the Expressions Simplification pass is buggy when considering whether an expression is an invariant which should be hoisted out of its residing loop. This causes the JIT compiler to calculate an incorrect number of iterations (142 instead of 426) when hoisting the expression at Line 8 out of the loop at Line 5. To fix this, the developers updated the invariant-check policy.

It should be noted that this bug manifests merely when `T.b()` is JIT-compiled and `T.f()` is hot enough such that the inserted loop is OSR-compiled at `scorching` level. That said, JIT-compiling `T.b()` and scorchingly JIT-compiling `T.f()` of the seed program[7] cannot trigger the bug as it does not involve OSR. Although it is theoretically possible for a test generator to generate such a bug-triggering program without `Artemis`, the time budget is generally unpredictable. As a comparison, `Artemis` can trigger this bug within 8 mutations of the seed program.

A.2 More Examples

Figure 6a. When HotSpot C1 compiles the given code, it tries to inline the method `invokeExact()` but requires the receiver to be non-null (Line 14). HotSpot developers forgot the null-check, and it thus triggers an assertion failure.

[7] `-Xjit:{T.b()I}(count=0),{T.f()V}(count=0,optLevel=scorching)`

Figure 6b. The DLT (Dynamic Loop Transfer which facilitates OSR) optimization pass in OpenJ9 fails to preserve the type information of the byte array `e` (Line 7), leading to an unexpected byte-bit truncation.

Figure 6c. OpenJ9 crashes with a segmentation fault when the method `T.e()` is compiled at `scorching` level. This is because OpenJ9's JIT compiler accesses a removed or invalid block when traversing predecessor blocks to compute and extend the live ranges of some variables.

Figure 6d. When compiling the loop in `T.f()`, the HotSpot C2 generates incorrect code, making HotSpot crash with a fatal arithmetic error when running the compiled code.

Figure 6e. OpenJ9 fails to vectorize the loop in `T.p()` and as a result, crashes as a segmentation fault because of a null pointer dereference when checking for loop independence.

Figure 6f. When an array access is out-of-bound in the compiled code, ART should de-optimize along a bounds-check slow path where the exception should be caught if there is an exception handler. However, ART's bounds-check slow path generator fails to compute the correct array index and length, which causes the exception to be caught at an incorrect index.

Figure 5. This presents the only performance issue found by `Artemis`. When running the code OSR-compiled by HotSpot C2, the native memory keeps increasing unexpectedly. This finally drains the machine of its memory and the process running this test is thereby killed by the underlying Ubuntu system (16GiB RAM). In contrast, despite not being killed on Windows (16GiB RAM), the process becomes far slower even than interpreting the bytecode. This issue was confirmed as a potential memory leak bug immediately after we reported, but marked as "`Won't Fix`" after several weeks because the developers hypothesized that "the app is simply allocating memory faster than the GC can reclaim it." However, we believe this issue should be a real bug that affects end users because (1) GC should be opaque to the end users, and (2) it should not be necessary for end users to be concerned with how fast the GC is when developing Java code.

```java
class T {
 boolean b;

 void a() {
  int c, v;
  double d = 2.61331;
  for (int z=830; z>51; --z)
    b = b;
  v = 1;
  while (++v < 908)
   for (c=-3230; c<9840; c+=2)
    try {
     MethodHandle m = null;
     m.invokeExact();
    } catch (Throwable t) {
    } finally {}
   System.out.println(d);
 }

 public static void main(...) {
  T t = new T();
  for (;;)
    t.a();
 }
}
```

(a) JDK-*****: Assertion Failure

```java
class T {
 void l() {
  int m, d;
  byte[] e = new byte[6];
  for (int j=2; j<222; ++j) {
   for (m=d=1; d<4; d+=2)
    e[m] -= d;
   for (int z=5; z<948; z++) {
    boolean[] x = new boolean[576];
   }
  }
  System.out.println(f(e));
 }
 long f(byte[] a) {
  long k = 0;  int i = 0;
  while (i < a.length) {
   k += a[i]; i++;
  }
  return k;
 }
 public static void main(...) {
  T t = new T(); t.l();
 }

}
```

(b) Issue-*****: Mis-compilation

```java
class T {
 void e() {
  for (int i=-4605; i<2; i++)
   try {
    double[] d = new double[1];
    d[0] = 0;
   } catch (Throwable x) {
   }
 }

 public static void main(...) {
  T s = new T();
  for (;;)
    s.e();
 }
```

(c) Issue-*****: Segmentation Fault

```java
class T {
 long c; int x;
 void f() {
  int i, p, q = 42252;
  int j, g, k = 5;
  double m;
  long[] l = new long[256];
  for (int o=2; o<87;) {
   boolean z = false;
   for (j=737; j<16822; j+=3)
    if (!z) {
     z = true;
     for (p=1; p<6; ++p) {
      for (m=1; m<2; m++) {
       l[(int) m] -= 16;
       c >>>= o;
      }
      for (g=1; 2>g; g+=3) {
       l[g] -= k; i = (int) c;
       try { k = q / i; }
       catch (ArithExcp w) {}
      }
      switch (o) {
       case 73: x = p;
      }
     }
    }
  }
 }
 void h() { f(); }
 public static void main(...) {
  T t = new T();  t.h();
 }
}
```

(d) JDK-*****: Fatal Arithmetic Error

```java
class T {
 long i;

 void p(long l) {
  int f = 0;
  int g[] = new int[0];
  long[] h = new long[0];
  for (int j=7; j<84;)
   try {
    long[] a = {1};
    for (int z=1; z<=f; z++)
     l += a[z - 1];
   } catch (Throwable r) {
   } finally {
    l = 7;
   }
  i = l + j(g) + j(h);
 }

 void k() { p(0l); }

 long j(long[] a) { return 0; }

 long j(int[] a) { return 0; }

 public static void main(...) {
  T t = new T();
  t.k();
 }

}
```

(e) Issue-*****: Segmentation Fault

```java
class T {
 int r;

 void f() {
  int i, j, o = 5788, t = 127;
  byte[] a = new byte[400];
  for (i=14; 297>i; ++i)
   for (j=151430; j<235417; j+=2);
  try {
   for (int d=4; 179>d; ++d) {
    o *= o;
    for (int k=1; k<58; k++)
     for (int z=k; 1+400>z;
       z++) {
      a[z] -= t;
      o += z;
      switch (d % 5) {
       case 107: d >>= r;
      }
     }
   }
  } catch (AIOOBExcp x) {}
  System.out.println(i + "," + o);
 }

 public static void main(...) {
  T t = new T(); t.f();
 }
}
```

(f) Issue-*****: Mis-compilation

Figure 6. A small selection of example tests finding various JVM bugs. For simplicity, we omit the required imports and the parameter of `T.main(String[] args)` method. Acronym `ArithExcp` represents `ArithmeticException`. Acronym `AIOOBExcp` stands for `ArrayIndexOutOfBoundsException`. Code shown in these examples are cleaned-up versions from very large test programs generated with a combination of JavaFuzzer and `Artemis`.