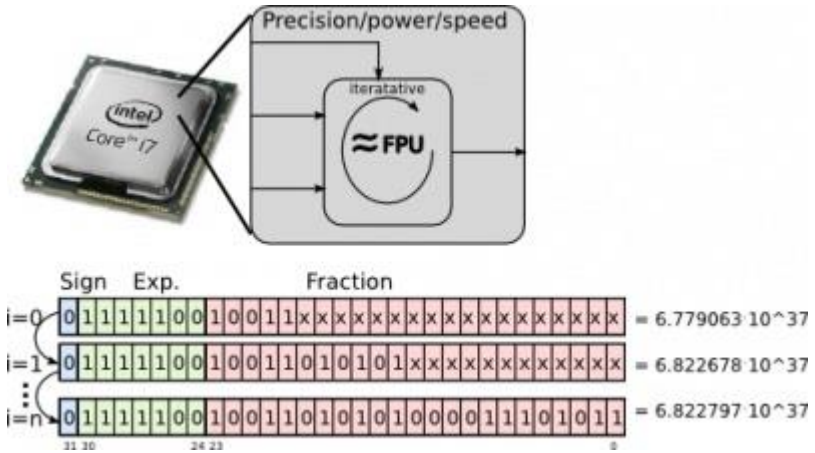
Improving Floating Point Symbolic Execution Coverage with Fixed Point Approximation

3rd International KLEE Workshop on Symbolic Execution

Key Idea

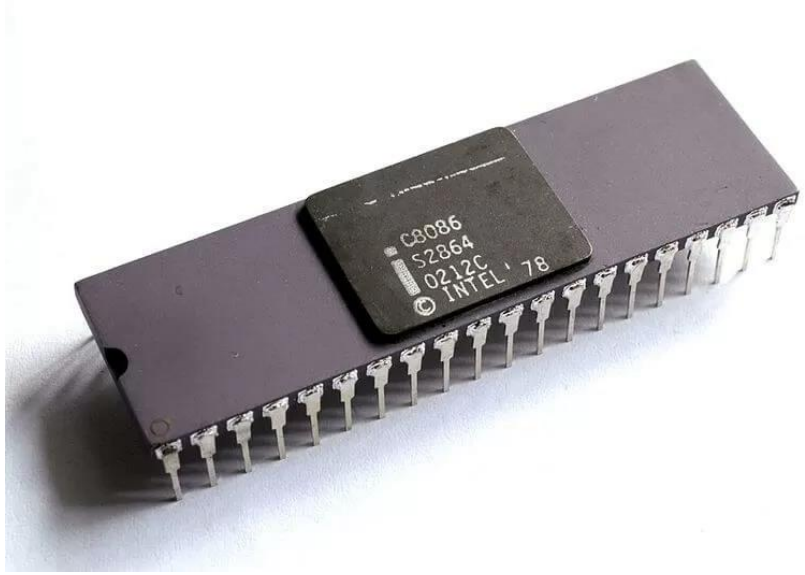
If precise, floating-point semantics are too difficult for SMT solvers, then substitute approximate, integer semantics.



Before floating point units (FPUs) became ubiquitous,

Key Idea

If precise, floating-point semantics are too difficult for SMT solvers, then substitute approximate, integer semantics.



Before floating point units (FPUs) became ubiquitous, there were fixed-point software libraries.

Fixed-Point background

- Developed as alternative to (then) expensive FPUs.
  - Early PC games (Doom) and game consoles (Playstation, Nintendo DS, etc.)
  - Still used in some simple embedded (IoT) devices
  - Also used in Q# for registers of qubits
- QM.N notation -> M integer bits and N fractional bits.
- E.g. Q16.16 is a 32 bit value that can represent numbers from -32768 to 32767 in increments of 1/65536

Example Program

```
01: #include <stdio.h>
02: #include <math.h>

03: char *foo(double bar) {
04:   double b = (2.0 * bar) + 1.0;
05:   char *result = ‘eq1’;
06:   if (b > 1.0)
07:     result = “gt1”;
08:   else if (b < 1.0)
09:     result = “lt1”;
10:   return result;
11: }

12: int main(int argc, char *argv[]) {
13:   double a;
14:   klee_make_symbolic(&a, sizeof(a), "a");
15:   printf(“%s\n“, foo(a));
16: }
```

Re-write Types

```
01: #include <stdio.h>
02: #include <math.h>

03: char *foo(double bar) {
04:   double b = (2.0 * bar) + 1.0;
05:   char *result = 'eq1';
06:   if (b > 1.0)
07:     result = "gt1";
08:   else if (b < 1.0)
09:     result = "lt1";
10:   return result;
11: }

12: int main(int argc, char *argv[]) {
13:   double a;
14:   klee_make_symbolic(&a, sizeof(a), "a");
15:   printf("%s\n", foo(a));
16: }
```

```
01: #include <stdio.h>
02: #include <math.h>

03: char *foo(f32_t bar) {
04:   f32_t b = (2.0 * bar) + 1.0;
05:   char *result = 'eq1';
06:   if (b > 1.0)
07:     result = "gt1";
08:   else if (b < 1.0)
09:     result = "lt1";
10:   return result;
11: }

12: int main(int argc, char *argv[]) {
13:   f32_t a;
14:   klee_make_symbolic(&a, sizeof(a), "a");
15:   printf("%s\n", foo(a));
16: }
```

Re-write Constants

```
01: #include <stdio.h>
02: #include <math.h>

03: char *foo(double bar) {
04:   double b = (2.0 * bar) + 1.0;
05:   char *result = ‘eq1’;
06:   if (b > 1.0)
07:     result = “gt1”;
08:   else if (b < 1.0)
09:     result = “lt1”;
10:   return result;
11: }

12: int main(int argc, char *argv[]) {
13:   double a;
14:   klee_make_symbolic(&a, sizeof(a), "a");
15:   printf(“%s\n“, foo(a));
16: }
```

```
01: #include <stdio.h>
02: #include <math.h>

03: char *foo(f32_t bar) {
04:   f32_t b = (F(2.0) * bar) + F(1.0);
05:   char *result = ‘eq1’;
06:   if (b > F(1.0))
07:     result = “gt1”;
08:   else if (b < F(1.0))
09:     result = “lt1”;
10:   return result;
11: }

12: int main(int argc, char *argv[]) {
13:   f32_t a;
14:   klee_make_symbolic(&a, sizeof(a), "a");
15:   printf(“%s\n“, foo(a));
16: }
```

Re-write Operators

```
01: #include <stdio.h>
02: #include <math.h>

03: char *foo(double bar) {
04:   double b = (2.0 * bar) + 1.0;
05:   char *result = ‘eq1’;
06:   if (b > 1.0)
07:     result = “gt1”;
08:   else if (b < 1.0)
09:     result = “lt1”;
10:   return result;
11: }

12: int main(int argc, char *argv[]) {
13:   double a;
14:   klee_make_symbolic(&a, sizeof(a), "a");
15:   printf(“%s\n“, foo(a));
16: }
```

```
01: #include <stdio.h>
02: #include <math.h>

03: char *foo(f32_t bar) {
04:   f32_t b = f32_add(f32_mul(F(2.0), bar),F(1.0));
05:   char *result = ‘eq1’;
06:   if (b > F(1.0))
07:     result = “gt1”;
08:   else if (b < F(1.0))
09:     result = “lt1”;
10:   return result;
11: }

12: int main(int argc, char *argv[]) {
13:   f32_t a;
14:   klee_make_symbolic(&a, sizeof(a), "a");
15:   printf(“%s\n“, foo(a));
16: }
```

IEEE-754 Special Values

qNaN, sNaN, +Inf, -Inf

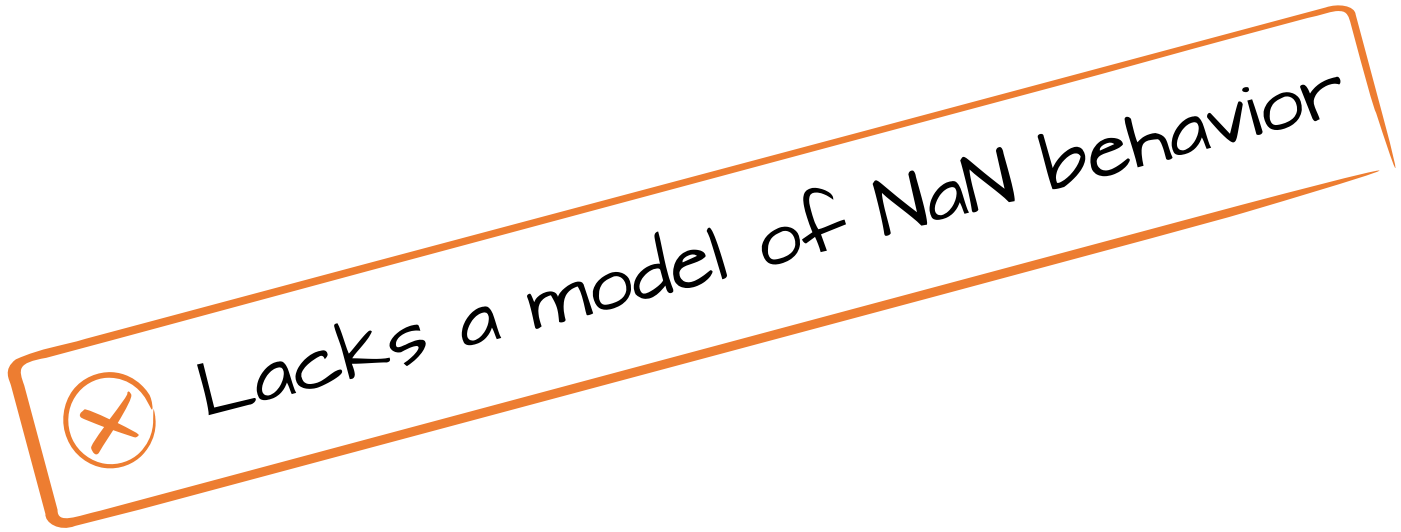
- Reserved Values: f32_nan and f32_inf
- Partial Semantic Model:
  - isnan(x), isinf(x)
  - isnan(x) → isnan(unary_op x)
  - isnan(x) ᵛ isnan(y) → isnan(x binary_op y)
  - x < 0.0 → isnan(sqrt(x))

f32_add

```
01: f32_t f32_add(f32_t a, f32_t b) {
02:   return a + b;
03: }
```

f32_add

```
01: f32_t f32_add(f32_t a, f32_t b) {
02:   return a + b;
03: }
```

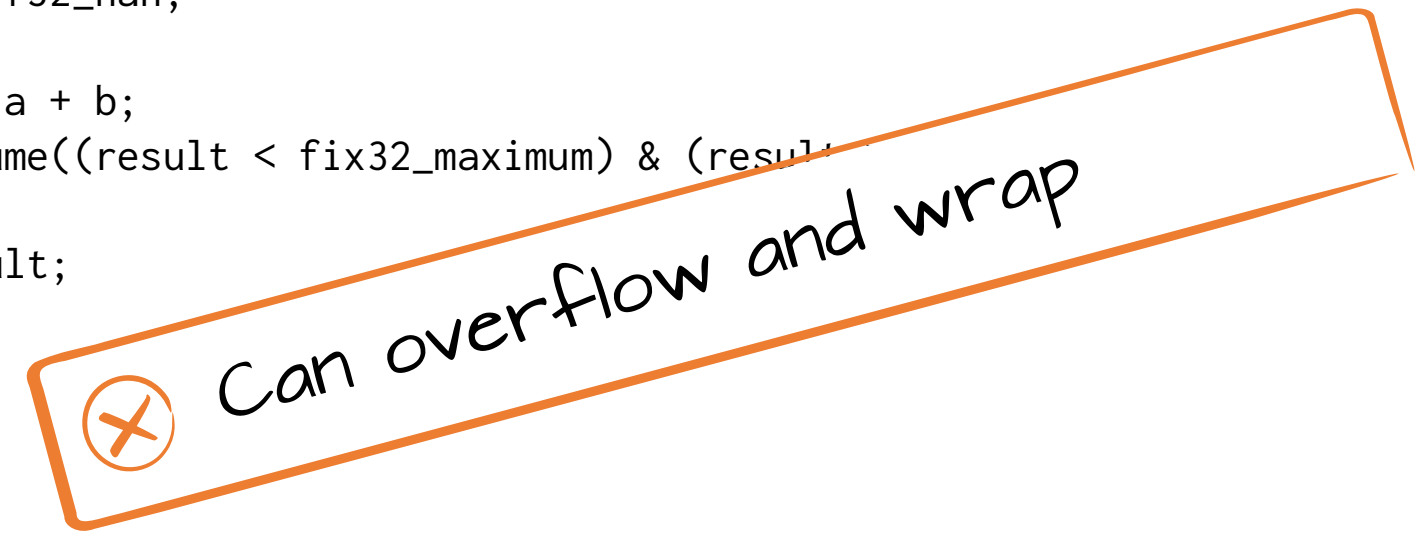


f32_add (nan semantics)

```
01: f32_t f32_add(f32_t a, f32_t b) {
02:   f32_t result;
03:   if (a == f32_nan | b == f32_nan) {
04:     result = f32_nan;
05:   } else {
06:     result = a + b;
07:     klee_assume((result < fix32_maximum) & (result > fix32_mimimum));
08:   }
09:   return result;
10: }
```

f32_add (nan semantics)

```
01: f32_t f32_add(f32_t a, f32_t b) {
02:   f32_t result;
03:   if (a == f32_nan | b == f32_nan) {
04:     result = f32_nan;
05:   } else {
06:     result = a + b;
07:     klee_assume((result < fix32_maximum) & (resul
08:   }
09:   return result;
10: }
```



f32_add (nan and no-overflow)

```
01: f32_t f32_add(f32_t a, f32_t b) {
02:   f32_t result;
04:   if (a == f32_nan | b == f32_nan) {
05:     result = f32_nan;
06:   } else {
        // Use unsigned integers because overflow with signed integers is undefined
07:     uint64_t _a = a, _b = b;
08:     uint64_t sum = _a + _b;

        // Overflow can only happen if sign of a == sign of b, and then
        // it causes sign of sum != sign of a.
09:     klee_assume(((_a ^ _b) & 0x8000000000000000) | !((_a ^ sum) & 0x8000000000000000));
10:     fix32_t result = sum;
11:     klee_assume((result < fix32_maximum) & (result > fix32_mimimum));
12:   }
14:   return result;
15: }
```

f32_mul

```
01: fix32_t fix32_mul(fix32_t a, fix32_t b) {
02:   f32_t result;

03:     if (a == f32_nan | b == f32_nan) {
04:       result = f32_nan;
05:     } else {
06:       int64_t A = (a >> 32), C = (b >> 32);
07:       uint64_t B = (a & 0xFFFFFFFF), D = (b & 0xFFFFFFFF);
08:       int64_t AC = A*C;
09:       int64_t AD_CB = A*D + C*B;
10:       uint64_t BD = B*D;
11:       int64_t product_hi = AC + (AD_CB >> 32);
          // Handle carry from lower 32 bits to upper part of result.
12:       uint64_t ad_cb_temp = AD_CB << 32;
13:       uint64_t product_lo = BD + ad_cb_temp;
14:       if (product_lo < BD) {
15:         product_hi++;
16:       }
          // No overflow.
17:       klee_assume(product_hi >> 63 == product_hi >> 31);
18:       fix32_t result = (product_hi << 32) | (product_lo >> 32);
19:       klee_assume((result < fix32_maximum) & (result > fix32_mimimum));
20:     }
21:   return result;
22: }
```

f32_div (libfixmath implementation)

```
fix32_t fix32_div(fix32_t a, fix32_t b) {
    // This uses a 64/64 bit division multiple times, until
    // computed bits in (a<<33)/b. Usually takes 1-3 iters.
    if (b == 0) return fix32_minimum;

    uint64_t remainder = (a >= 0) ? a : (-a);
    uint64_t divider = (b >= 0) ? b : (-b);
    uint64_t quotient = 0;
    int bit_pos = 33;

    // Kick-start the division a bit.
    // This improves speed in scenarios where N and D are large
    // gets a lower estimate for the result by N/(D >> 33 + 1).
    if (divider & 0xFFF0000000000000) {
        uint64_t shifted_div = ((divider >> 33) + 1);
        quotient = remainder / shifted_div;
        remainder -= ((uint64_t)quotient * divider) >> 17;
    }

    // If the divider divisible by 2^n, take advantage of it.
    while (!(divider & 0xF) && bit_pos >= 4) {
        divider >>= 4;
        bit_pos -= 4;
    }
```

continued in next column

continued from last column

```
    while (remainder && bit_pos >= 0) {
        // Shift remainder as much as we can without overflowing
        int shift = clz(remainder);
        if (shift > bit_pos) shift = bit_pos;
        remainder <<= shift;
        bit_pos -= shift;

        uint64_t div = remainder / divider;
        remainder = remainder % divider;
        quotient += div << bit_pos;
        if (div & ~(0xFFFFFFFFFFFFFFFF >> bit_pos))
          return fix32_overflow;
        remainder <<= 1;
        bit_pos--;
    }
    fix32_t result = quotient >> 1;

    // Figure out the sign of the result
    if ((a ^ b) & 0x8000000000000000) {
        if (result == fix32_minimum) return fix32_overflow;
        result = -result;
    }
    return result;
}
```

f32_div (division is hard!)

```
fix32_t fix32_div(fix32_t a, fix32_t b) {
    // This uses a 64/64 bit division multiple times, until
    // computed bits in (a<<33)/b. Usually takes 1-3 iters.
    if (b == 0) return fix32_minimum;

    uint64_t remainder = (a >= 0) ? a : (-a);
    uint64_t divider = (b >= 0) ? b : (-b);
    uint64_t quotient = 0;
    int bit_pos = 33;

    // Kick-start the division a bit.
    // This improves speed in scenarios where N and D are large
    // gets a lower estimate for the result by N/(D >> 33 + 1).
    if (divider & 0xFFF0000000000000) {
        uint64_t shifted_div = ((divider >> 33) + 1);
        quotient = remainder / shifted_div;
        remainder -= ((uint64_t)quotient * divider) >> 17;
    }

    // If the divider divisible by 2^n, take advantage of it.
    while (!(divider & 0xF) && bit_pos >= 4) {
        divider >>= 4;
        bit_pos -= 4;
    }
```

continued in next column

continued from last column

```
    while (remainder && bit_pos >= 0) {
        // Shift remainder as much as we can without overflowing
        int shift = clz(remainder);
        if (shift > bit_pos) shift = bit_pos;
        remainder <<= shift;
        bit_pos -= shift;

        uint64_t div = remainder / divider;
        remainder = remainder % divider;
        quotient += div << bit_pos;
        if (div & ~(0xFFFFFFFFFFFFFFFF >> bit_pos))
            return fix32_overflow;
        remainder <<= 1;
        bit_pos--;
    }
    fix32_t result = quotient >> 1;

    // Figure out the sign of the result
    if ((a ^ b) & 0x8000000000000000) {
        if (result == fix32_minimum) return fix32_overflow;
        result = -result;
    }
    return result;
}
```

f32_div (division is hard!)

```
fix32_t fix32_div(fix32_t a, fix32_t b) {
    // This uses a 64/64 bit division multiple times, until
    // computed bits in (a<<33)/b. Usually takes 1-3 iters.
    if (b == 0) return fix32_minimum;

    uint64_t remainder = (a >= 0) ? a : (-a);
    uint64_t divider = (b >= 0) ? b : (-b);
    uint64_t quotient = 0;
    int bit_pos = 33;

    // Kick-start the division a bit.
    // This improves speed in scena
    // gets a lower estimat
    if (divider & 0xFFF000
        uint64_t shifted_di
        quotient = remainde
        remainder -= ((uint6
    }

    // If the divider divisib
    while (!(divider & 0xF) &&
        divider >>= 4;
        bit_pos -= 4;
    }
```

continued in next column

continued from last column

```
    while (remainder && bit_pos >= 0) {
        // Shift remainder as much as we can without overflowing
        int shift = clz(remai    r);
        if (shift > bi      ft = bit_pos;
        remaind

                                  divider;
                                  ider;

                                  FF >> bit_pos))

                                  >> 1;

                                  the sign of the result
                                  b) & 0x8000000000000000) {
        if (result == fix32_minimum) return fix32_overflow;
        result = -result;
    }
    return result;
}
```

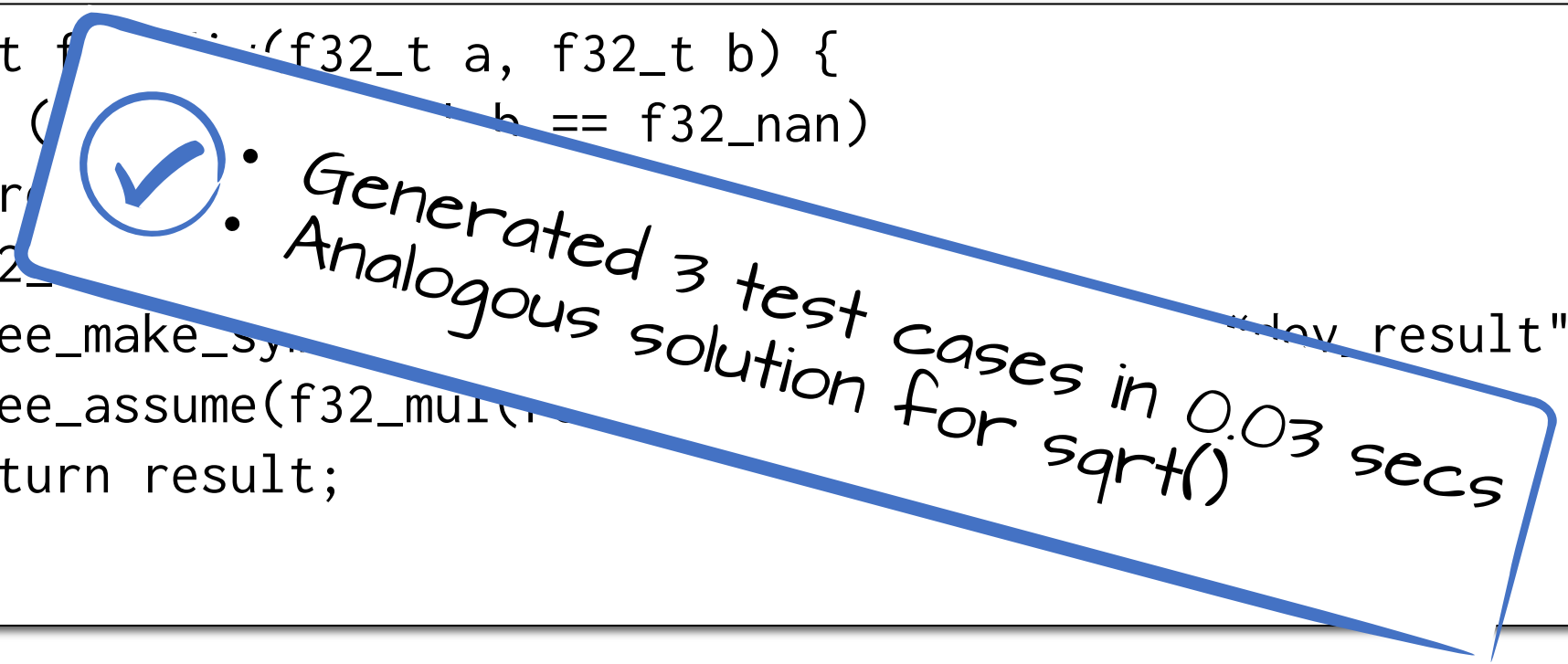
- A single division completed, but generated 124 inputs
- sqrt() was worse. At 60 secs, 1200 incomplete paths

f32_div (but multiplication is easy)

```
01: f32_t f32_div(f32_t a, f32_t b) {
02:   if (a == f32_nan | b == f32_nan)
03:     return f32_nan;
04:   f32_t result;
05:   klee_make_symbolic(&result, sizeof(result), “dev_result");
06:   klee_assume(f32_mul(result, b) == a);
07:   return result;
08: }
```

f32_div (but multiplication is easy)

```
01: f32_t f                (f32_t a, f32_t b) {
02:   if (                    b == f32_nan)
03:     r
04:   f32_
05:   klee_make_sy                                  dev_result");
06:   klee_assume(f32_mul(
07:   return result;
08: }
```



Preliminary Trial

- 12 benchmarks from 14 Imperial synthetic contribution to symex-fp-bench*
  - Discarded 2:
  - memcpy_and_check… ,depends upon IEEE format
  - paranoia, could not find symbolic input?

* https://github.com/delcypher/symex-fp-bench

Preliminary Results

	vanilla klee		fixed point	
Benchmark	ICov(%)	BCov(%)	ICov(%)	BCov(%)
count_klee	33.87	22.22	91.85	85.00
interval_klee_no_bug	94.10	56.25	93.85	58.33
matrix_inverse_klee_double_4	89.20	75.00	97.06	89.47
memcpy_and_use_as_bitvector_klee	93.51	70.83	92.12	78.57
non_terminating_klee_no_bug	83.72	50.00	90.83	60.00
prefix_sum_klee_bug_double	95.65	75.00	94.81	85.00
rounding_sqrt_klee	21.90	12.50	72.73	65.00
sorted_search_klee_bug_float	81.11	54.55	95.29	91.67
sqrt_klee	74.44	40.91	93.97	67.65
sum_is_commutative_klee_double	97.96	75.00	94.19	75.00
sum_is_not_associative_klee_bug	97.96	75.00	94.19	75.00
vanishing_klee_bug	72.60	40.00	91.83	71.43
Total (12)	78.00	53.94	91.86	75.18

SymEx & FXP: Approach

```
01: #include <stdio.h>
02: #include <math.h>

03: char *foo(f32_t bar) {
04:   f32_t b = f32_add(f32_mul(F(2.0), bar),F(1.0));
05:   char *result = ‘eq1’;
06:   if (b > F(1.0))
07:     result = “gt1”;
08:   else if (b < F(1.0))
09:     result = “lt1”;
10:   return result;
11: }

12: int main(int argc, char *argv[]) {
13:   f32_t a;
14:   klee_make_symbolic(&a, sizeof(a), "a");
15:   printf(“%s\n“, foo(a));
16: }
```

- Automatically rewrite FP operations with Fixed Point (FXP) analogues
- **Advantages**:
  - Approximates FP arithmetic with Integer operations
  - Does not require a heavy-weight, FP-enabled solver
  - Proof of concept: Prototype FXP-enabled KLEE found all 3 paths thought sample program
- **Disadvantages**:
  - Risk of divergence between symbolic FXP solution and equivalent FP, due to approximation
  - Difficult to solve for special FP behavior, such as underflow and overflow


Questions