Bitween: Automated Inference of Program Invariants

Explore Bitween: https://bitween.fun/. The web interface provides a dropdown menu for selecting and editing predefined benchmarks. Although Bitween operates as a black-box tool, it is currently integrated with C, supporting a subset that excludes arrays, pointers, and structs. We are actively working to expand this support. When users click the "Analyze Code" button, Bitween replaces `vtrace` calls with inferred properties in `assert` statements. Users can verify correctness through fuzzing or bounded model checking.

OVERVIEW OF BITWEEN

Bitween combines machine learning with formal methods to automatically infer program properties and invariants. Through random fuzzing and static analysis, it identifies assertions, deriving loop invariants and post-conditions for C programs. Users instrument their code by adding functions prefixed with `vtrace` at key locations, such as after entering loops, to monitor terms likely to appear in invariants.

The interface includes an **example dropdown** for users to select predefined benchmarks. Key parameters, such as the **entry method** (the function where fuzzing begins) and the **degree** of inferred properties, can be configured to capture more complex relationships.

While the backend offers advanced configuration options, the graphical interface emphasizes simplicity for demonstration purposes. By default, Bitween infers properties up to degree 2, using the terms passed to the `vtrace` function.

For instance, in Figure 1, Bitween aims for learning a loop invariant, by using `vtrace1` at line 8 as $F[X, Y, x, y, v] = 0$, where F is an unknown algebraic function. Bitween aims to interpret F in the hypothesis class of polynomials constructed by the terms passed to F and bounded up to a given degree.

Users can validate these inferred properties through fuzzing or bounded model checking using 'correctness check' dropdown.

Additionally, users can include `assume` statements to refine the analysis, focusing on specific cases of interest.

To infer post-conditions, users can place `vtrace` calls before function exits or within function calls, ensuring the invariants are inferred at key points.

For more sophisticated input generation, users can define intervals using the `vdistr` function to guide the fuzzing process with uniform distributions:

```
1 vdistr(X, 0, 1000);
2 vdistr(Y, 0, 1000);
```

Although Bitween currently supports only a subset of C, ongoing development will extend it to handle arrays, pointers, and structs, and improve its handling of inequalities.

EXAMPLE PROGRAMS

In the example programs, Bitween starts by using 20 random inputs to find invariants about the variables. Here's an overview of two specific examples:

(1) **Bresenham's Line Drawing Algorithm**: This algorithm is used to draw lines on a grid, selecting the best-fitting pixels between two points. In the first version (Figure 1a), the program is instrumented with `vtrace` calls to track variables. Bitween processes these traces and infers

Author's address:

```
int bresenham(int X, int Y) {
  vdistr(X, 0, 1000);
  vdistr(Y, 0, 1000);
  assume(X >= 0);

  int v = 2 * Y - X, x = 0, y = 0;
  while (1) {
    vtrace1(X, Y, x, y, v);

    if (!(x <= X)) break;
    if (v < 0) {
      v = v + 2 * Y;
    } else {
      v = v + 2 * (Y - X); y++;
    }
    x++;
  }

  vtrace2(X, Y, x, y, v);
  return v;
}
```

(a) Input program instrumented with `vtrace1` and `vtrace2`.

```
int bresenham(int X, int Y) {
  vdistr(X, 0, 1000);
  vdistr(Y, 0, 1000);
  assume(X >= 0);

  int v = 2 * Y - X, x = 0, y = 0;
  while (1) {
    assert(2*X*y + X - 2*Y*x - 2*Y + v == 0);

    if (!(x <= X)) break;
    if (v < 0) {
      v = v + 2 * Y;
    } else {
      v = v + 2 * (Y - X); y++;
    }
    x++;
  }
  assert(X - x + 1 == 0);
  assert(2*Y*x +2*Y -v -2*x*y -x +2*y +1==0);
  return v;
}
```

(b) Output program with `vtrace` locations replaced by inferred invariants.

Fig. 1. Input and output for Bresenham's line drawing algorithm, based on Srivastava et al.'s *From Program Verification to Program Synthesis*, POPL '10.

```
float z3sqrt(float a) {
  assume(a >= 1);

  float x = a;
  float q = 0.5 * (x + a / x);
  float err = 0.0001;

  while(x - q > err or q - x > err){
    vtrace1(a, x, err, q);

    x = q;
    q = 0.5 * (x + a / x);
  }

  return q;
}
```

(a) Instrumented input program for computing square roots, by Zuse.

```
float z3sqrt(float a) {
  assume(a >= 1);

  float x = a;
  float q = 0.5 * (x + a / x);
  float err = 0.0001;

  while(x - q > err or q - x > err){
    assert(a - 2*q*x + pow(x, 2) == 0);
    assert(err == 0);
    x = q;
    q = 0.5 * (x + a / x);
  }

  return q;
}
```

(b) Output program with `vtrace` replaced by inferred invariants.

Fig. 2. Input and output programs for computing square roots, based on Zuse's implementation https://www.cs.upc.edu/~erodri/webpage/polynomial_invariants/z3sqrt.htm.

invariants about the relationships between variables involved in the line drawing process. The output (Figure 1b) replaces the `vtrace` calls with assertions that represent the inferred invariants, ensuring that the program correctly draws lines under different conditions.

(2) **Square Root Computation by Zuse**: This example computes square roots using an iterative method. Although the calculation is approximate, based on a fixed error (‘err = 0.0001‘), Bitween mathematically finds the correct relationships between the variables. The program is first instrumented with `vtrace` calls to track key variables (Figure 2a). Using 20 random inputs, Bitween analyzes the program and infers the necessary relationships (invariants) between these variables during computation. The output (Figure 2b) contains the inferred assertions in place of the `vtrace` calls, ensuring the correctness of the square root calculation, despite the approximate nature of the iterative method.