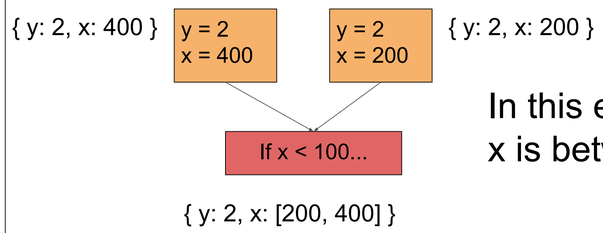
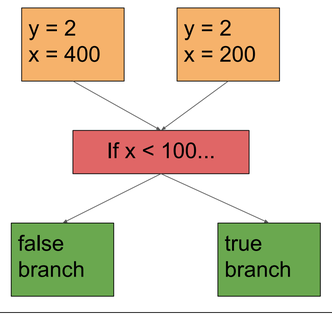
Compilers use range analysis to gather information for later optimization passes



In this example, range analysis figures out x is between 200 and 400 at the red join point.

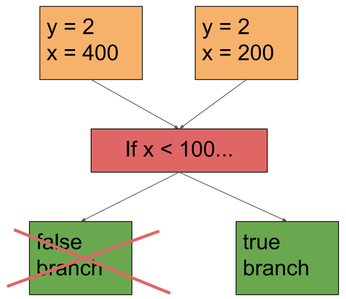
The compiler uses this information to optimize the program:



Since x is between 200 and 400 at the red program point, the compiler can eliminate the case where x is less than 100

Incorrect range analysis can introduce security issues into seemingly-correct code

Broken range analysis can cause the compiler to incorrectly optimize programs.



If range analysis is wrong about x, it could tell optimization passes to remove the wrong code!

In the browser, *attackers can use these kinds of bugs to remotely compromise victim machines.*

Firefox implements a range analysis pass in its browser JIT...

JavaScript values are floating point, so Firefox's range analysis object must handle floating point edge cases:

```
MOZ_INIT_OUTSIDE_CTOR int32_t lower_;
MOZ_INIT_OUTSIDE_CTOR int32_t upper_;

MOZ_INIT_OUTSIDE_CTOR bool hasInt32LowerBound_;
MOZ_INIT_OUTSIDE_CTOR bool hasInt32UpperBound_;

MOZ_INIT_OUTSIDE_CTOR FractionalPartFlag canHaveFractionalPart_ : 1;
MOZ_INIT_OUTSIDE_CTOR NegativeZeroFlag canBeNegativeZero_ : 1;
MOZ_INIT_OUTSIDE_CTOR uint16_t max_exponent_;
```

For each operator Op (e.g., absolute value) and its input ranges, Firefox updates the range object to reflect the new range after Op:

```
Range* Range::abs(TempAllocator& alloc, const Range* op) {
  int32_t l = op->lower_;
  int32_t u = op->upper_;
  FractionalPartFlag canHaveFractionalPart = op->canHaveFractionalPart_;
```

… which we partially verify, discovering a bug that has existed in the browser for over six years

Given range analysis routines in a subset of C++, our tool, VeRA, automatically verifies them correct according to JavaScript semantics: