Microsoft Research

Microsoft Research

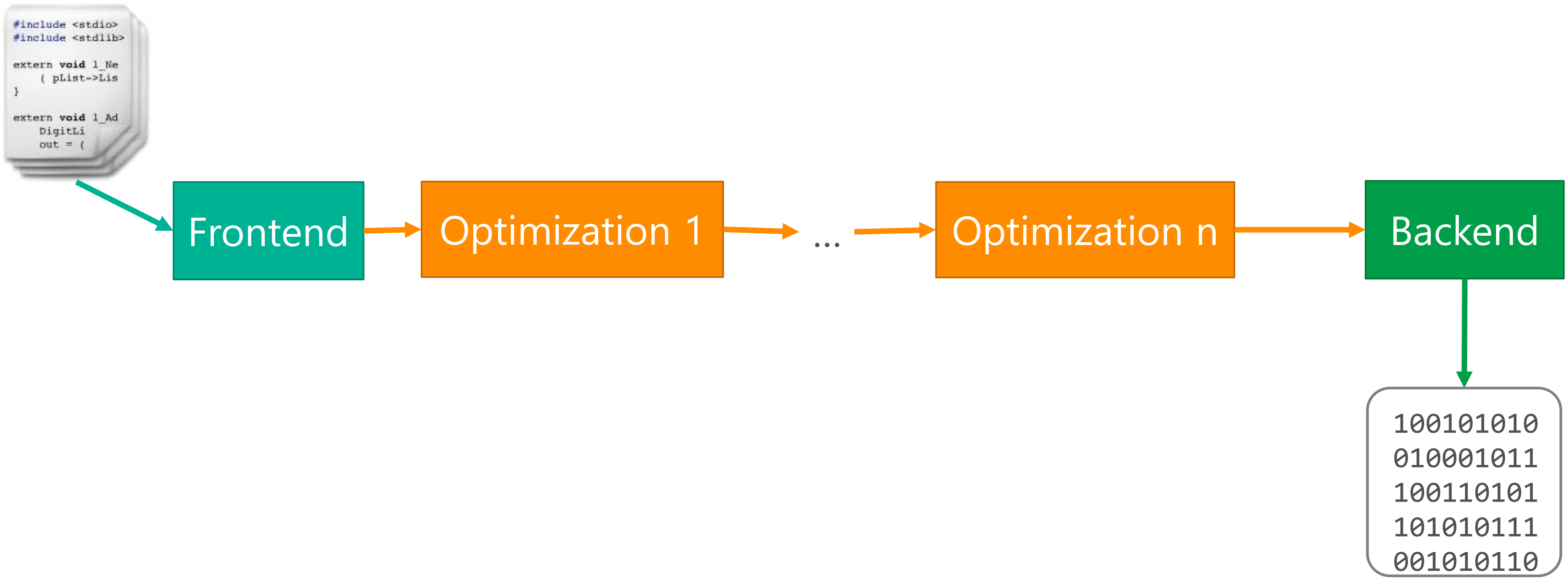
Microsoft

Improving Reliability of Compilers

Nuno Lopes

Joint work with David Menendez, Santosh Nagarakatte, John Regehr, Sarah Winkler

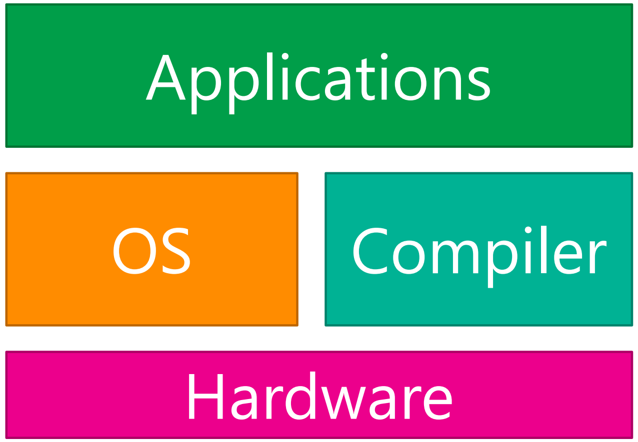
Compilers...



#include <stdio>
#include <stdlib>
extern void l_Ne
{ pList->Lis
}
extern void l_Ad
DigitLi
out = (
Frontend
Optimization 1
...
Optimization n
Backend
100101010
010001011
100110101
101010111
001010110

Why care about Compilers?

- Today's software goes through at least one compiler
- Correctness and safety depends on compilers



Pressure for better Compilers

- Improve performance
- Reduce code size
- Reduce energy consumption

Compilers do deliver

- LLVM 3.2 introduced a Loop Vectorizer
  - Performance improvement of 10-300% in benchmarks

- Visual Studio 2015 Update 3: > 10% on SPEC CPU 2k/2k6

Compilers also have Bugs

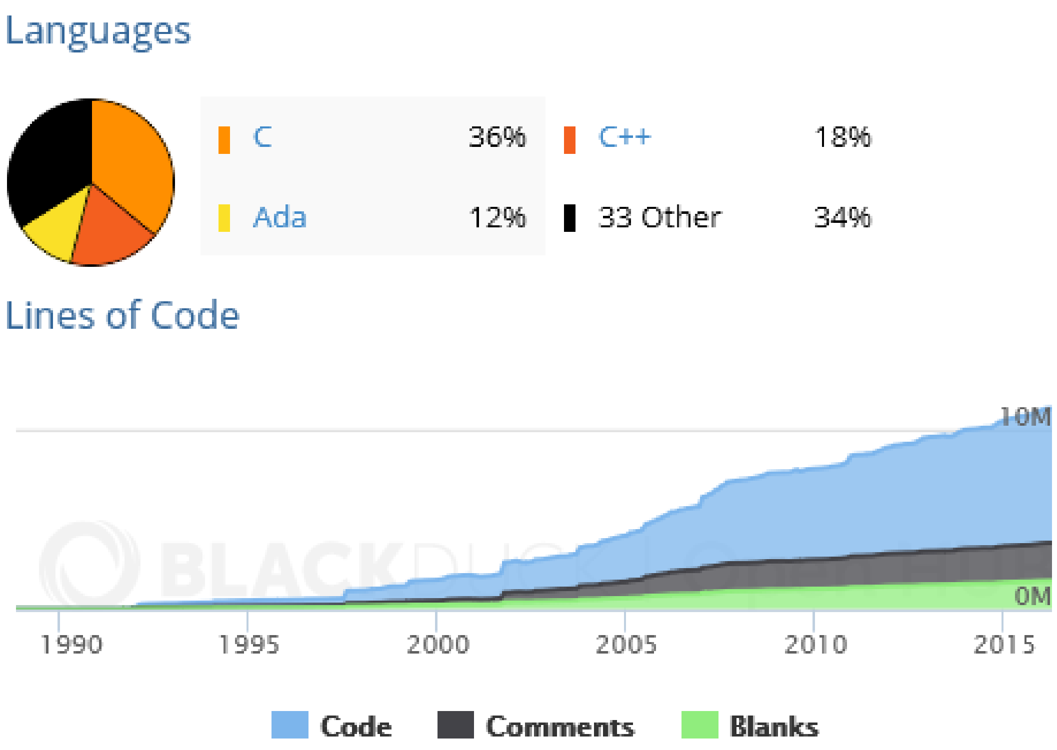
- Csmith [PLDI’11]:
  - 79 bugs in GCC (25 P1)
  - 202 bugs in LLVM
- Orion [PLDI’14]:
  - 40 wrong-code bugs in GCC
  - 42 wrong-code bugs in LLVM
- Last Week:
  - 476 open wrong-code bug reports in GCC (out of 9,930)
  - 22 open wrong-code bug reports in LLVM (out of 7,002)

Buggy Compilers = Security Bugs

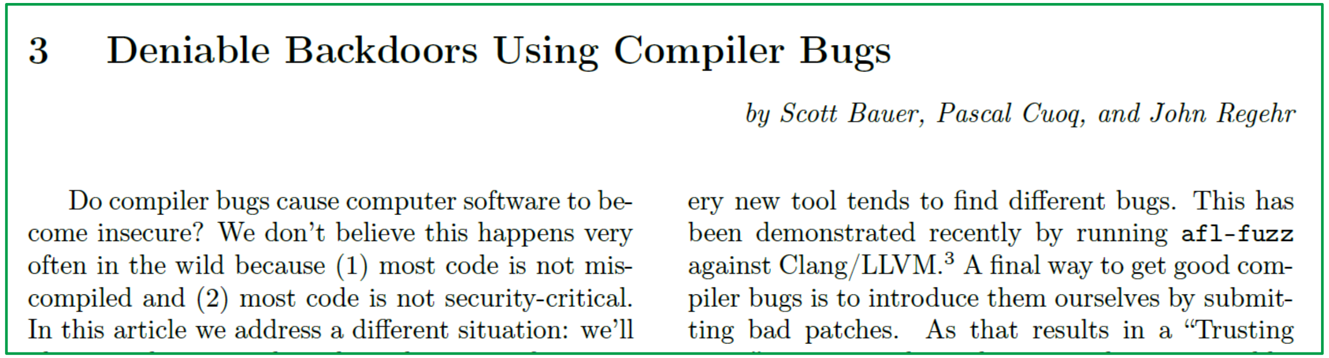
- CVE-2006-1902
- GCC 4.1/4.2 (fold-const.c) had a bug that could remove valid pointer comparisons
- Removed bounds checks in, e.g., Samba

Churn in Compiler's code

gcc:



March to Disaster?

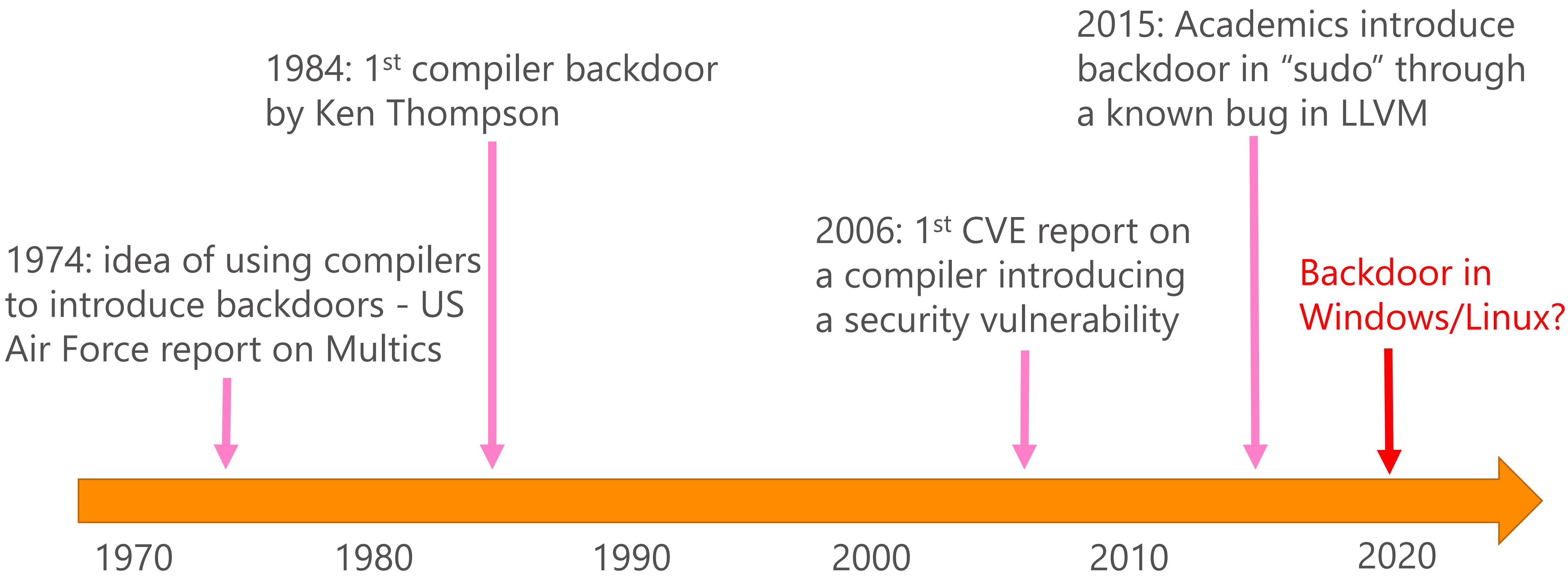


3 Deniable Backdoors Using Compiler Bugs

by Scott Bauer, Pascal Cuoq, and John Regehr

Do compiler bugs cause computer software to become insecure? We don't believe this happens very often in the wild because (1) most code is not miscompiled and (2) most code is not security-critical. In this article we address a different situation: we'll

ery new tool tends to find different bugs. This has been demonstrated recently by running `afl-fuzz` against Clang/LLVM.[3] A final way to get good compiler bugs is to introduce them ourselves by submitting bad patches. As that results in a "Trusting



Motivation


Pressure to Improve Compilers
More Bugs in Compilers
Potential Crashes and Vulnerabilities in Compiled Software

Software Development Practices

- Unit tests
- End-to-end tests
- Fuzzing
- Static Analysis
- Verification of pseudo-code
- …

Unit tests, End-to-end tests: Since ever?

Fuzzing, Static Analysis: Since 2000s (SLAM, PREfast, SAGE, Z3…)

Verification of pseudo-code: ?

~~Software~~ Compiler Development Practices

- Unit tests
- End-to-end tests
- Fuzzing
- Static Analysis
- Verification of pseudo-code
- Semi-automated verification – CompCert (2008)
- …

Since ever?

~~Since 2000s~~

Csmith – industry standard [PLDI'11]

?

?

Why no Static Analysis for Compilers?

- Static analysis for general software targets safety (i.e., no crashes)
- Compiler correctness needs functional verification

- Safety checking still an open research problem
- Functional verification is harder

“Static Analysis” for Compilers

- Compiler verification: verify compiler once and for all (e.g., CompCert, Alive [PLDI’15])

- Compiler validation: verify output of compiler on every compilation (utcTV)

- For new code: specify in a DSL and verify automatically
- For legacy code: validation

Optimizations are Easy to Get Wrong

```
int a = x << c;
int b = a / d;
```

➡

```
int t = d / (1 << c);
int b = x / t;
```

$x * 2^c / d$

$x / (d / 2^c) = x / d * 2^c$

$= x * 2^c / d$

(c and d are constants)

Optimizations are Easy to Get Wrong

```
int a = x << c;
int b = a / d;
```

➡

```
int t = d / (1 << c);
int b = x / t;
```

```
ERROR: Domain of definedness of Target is smaller
than Source's for i4 %b

Example:
%X i4 = 0x0 (0)
c  i4 = 0x3 (3)
d  i4 = 0x7 (7)
%a i4 = 0x0 (0)
(1 << c) i4 = 0x8 (8, -8)
%t i4 = 0x0 (0)
Source value: 0x0 (0)
Target value: undef
```

LLVM bug #21245

Implementing Peephole Optimizers

```
{
  Value *Op1C = Op1;
  BinaryOperator *BO = dyn_cast<BinaryOperator>(Op0);
  if (!BO ||
      (BO->getOpcode() != Instruction::UDiv &&
       BO->getOpcode() != Instruction::SDiv)) {
    Op1C = Op0;
    BO = dyn_cast<BinaryOperator>(Op1);
  }
  Value *Neg = dyn_castNegVal(Op1C);
  if (BO && BO->hasOneUse() &&
      (BO->getOperand(1) == Op1C || BO->getOperand(1) == Neg) &&
      (BO->getOpcode() == Instruction::UDiv ||
       BO->getOpcode() == Instruction::SDiv)) {
    Value *Op0BO = BO->getOperand(0), *Op1BO = BO->getOperand(1);

    // If the division is exact, X % Y is zero, so we end up with X or -X.
    if (PossiblyExactOperator *SDiv = dyn_cast<PossiblyExactOperator>(BO))
      if (SDiv->isExact()) {
        if (Op1BO == Op1C)
          return ReplaceInstUsesWith(I, Op0BO);
        return BinaryOperator::CreateNeg(Op0BO);
      }

    Value *Rem;
    if (BO->getOpcode() == Instruction::UDiv)
      Rem = Builder->CreateURem(Op0BO, Op1BO);
    else
      Rem = Builder->CreateSRem(Op0BO, Op1BO);
    Rem->takeName(BO);

    if (Op1BO == Op1C)
      return BinaryOperator::CreateSub(Op0BO, Rem);
    return BinaryOperator::CreateSub(Rem, Op0BO);
  }
}
```

Alive

- New language and tool for:
  - Specifying peephole optimizations
  - Proving them correct (or generate a counterexample)
  - Generating C++ code for a compiler
- Design point: both practical and formal

A Simple Peephole Optimization

```
{
  Value *Op1C = Op1;
  BinaryOperator *BO = dyn_cast<BinaryOperator>(Op0);
  if (!BO ||
      (BO->getOpcode() != Instruction::UDiv &&
       BO->getOpcode() != Instruction::SDiv)) {
    Op1C = Op0;
    BO = dyn_cast<BinaryOperator>(Op1);
  }
  Value *Neg = dyn_castNegVal(Op1C);
  if (BO && BO->hasOneUse() &&
      (BO->getOperand(1) == Op1C || BO->getOperand(1) == Neg) &&
      (BO->getOpcode() == Instruction::UDiv ||
       BO->getOpcode() == Instruction::SDiv)) {
    Value *Op0BO = BO->getOperand(0), *Op1BO = BO->getOperand(1);

    // If the division is exact, X % Y is zero, so we end up with X or -X.
    if (PossiblyExactOperator *SDiv = dyn_cast<PossiblyExactOperator>(BO))
      if (SDiv->isExact()) {
        if (Op1BO == Op1C)
          return ReplaceInstUsesWith(I, Op0BO);
        return BinaryOperator::CreateNeg(Op0BO);
      }

    Value *Rem;
    if (BO->getOpcode() == Instruction::UDiv)
      Rem = Builder->CreateURem(Op0BO, Op1BO);
    else
      Rem = Builder->CreateSRem(Op0BO, Op1BO);
    Rem->takeName(BO);

    if (Op1BO == Op1C)
      return BinaryOperator::CreateSub(Op0BO, Rem);
    return BinaryOperator::CreateSub(Rem, Op0BO);
  }
}
```

```
int f(int x, int y) {
  return (x / y) * y;
}
```

Compile to LLVM IR

```
define i32 @f(i32 %x, i32 %y) {
  %1 = sdiv i32 %x, %y
  %2 = mul i32 %1, %y
  ret i32 %2
}
```

Optimize

```
define i32 @f(i32 %x, i32 %y) {
  %1 = srem i32 %x, %y
  %2 = sub i32 %x, %1
  ret i32 %2
}
```

A Simple Peephole Optimization

```
{
  Value *Op1C = Op1;
  BinaryOperator *BO = dyn_cast<BinaryOperator>(Op0);
  if (!BO ||
      (BO->getOpcode() != Instruction::UDiv &&
       BO->getOpcode() != Instruction::SDiv)) {
    Op1C = Op0;
    BO = dyn_cast<BinaryOperator>(Op1);
  }
  Value *Neg = dyn_castNegVal(Op1C);
  if (BO && BO->hasOneUse() &&
      (BO->getOperand(1) == Op1C || BO->getOperand(1) == Neg) &&
      (BO->getOpcode() == Instruction::UDiv ||
       BO->getOpcode() == Instruction::SDiv)) {
    Value *Op0BO = BO->getOperand(0), *Op1BO = BO->getOperand(1);

    // If the division is exact, X % Y is zero, so we end up with X or -X.
    if (PossiblyExactOperator *SDiv = dyn_cast<PossiblyExactOperator>(BO))
      if (SDiv->isExact()) {
        if (Op1BO == Op1C)
          return ReplaceInstUsesWith(I, Op0BO);
        return BinaryOperator::CreateNeg(Op0BO);
      }

    Value *Rem;
    if (BO->getOpcode() == Instruction::UDiv)
      Rem = Builder->CreateURem(Op0BO, Op1BO);
    else
      Rem = Builder->CreateSRem(Op0BO, Op1BO);
    Rem->takeName(BO);

    if (Op1BO == Op1C)
      return BinaryOperator::CreateSub(Op0BO, Rem);
    return BinaryOperator::CreateSub(Rem, Op0BO);
  }
}
```

```
define i32 @f(i32 %x, i32 %y) {
  %1 = sdiv i32 %x, %y
  %2 = mul i32 %1, %y
  ret i32 %2
}
```

=>

Optimize

```
define i32 @f(i32 %x, i32 %y) {
  %1 = srem i32 %x, %y
  %2 = sub i32 %x, %1
  ret i32 %2
}
```


```
define i32 @f(i32 %x, i32 %y) {
  %1 = srem i32 %x, %y
  %2 = sub i32 %x, %1
  ret i32 %2
}
```

A Simple Peephole Optimization

```
{
  Value *Op1C = Op1;
  BinaryOperator *BO = dyn_cast<BinaryOperator>(Op0);
  if (!BO ||
      (BO->getOpcode() != Instruction::UDiv &&
       BO->getOpcode() != Instruction::SDiv)) {
    Op1C = Op0;
    BO = dyn_cast<BinaryOperator>(Op1);
  }
  Value *Neg = dyn_castNegVal(Op1C);
  if (BO && BO->hasOneUse() &&
      (BO->getOperand(1) == Op1C || BO->getOperand(1) == Neg) &&
      (BO->getOpcode() == Instruction::UDiv ||
       BO->getOpcode() == Instruction::SDiv)) {
    Value *Op0BO = BO->getOperand(0), *Op1BO = BO->getOperand(1);

    // If the division is exact, X % Y is zero, so we end up with X or -X.
    if (PossiblyExactOperator *SDiv = dyn_cast<PossiblyExactOperator>(BO))
      if (SDiv->isExact()) {
        if (Op1BO == Op1C)
          return ReplaceInstUsesWith(I, Op0BO);
        return BinaryOperator::CreateNeg(Op0BO);
      }

    Value *Rem;
    if (BO->getOpcode() == Instruction::UDiv)
      Rem = Builder->CreateURem(Op0BO, Op1BO);
    else
      Rem = Builder->CreateSRem(Op0BO, Op1BO);
    Rem->takeName(BO);

    if (Op1BO == Op1C)
      return BinaryOperator::CreateSub(Op0BO, Rem);
    return BinaryOperator::CreateSub(Rem, Op0BO);
  }
}
```

```
  %1 = sdiv i32 %x, %y
  %2 = mul i32 %1, %y

=>

  %t = srem i32 %x, %y
  %2 = sub i32 %x, %t
```

A Simple Peephole Optimization

```
{
  Value *Op1C = Op1;
  BinaryOperator *BO = dyn_cast<BinaryOperator>(Op0);
  if (!BO ||
      (BO->getOpcode() != Instruction::UDiv &&
       BO->getOpcode() != Instruction::SDiv)) {
    Op1C = Op0;
    BO = dyn_cast<BinaryOperator>(Op1);
  }
  Value *Neg = dyn_castNegVal(Op1C);
  if (BO && BO->hasOneUse() &&
      (BO->getOperand(1) == Op1C || BO->getOperand(1) == Neg) &&
      (BO->getOpcode() == Instruction::UDiv ||
       BO->getOpcode() == Instruction::SDiv)) {
    Value *Op0BO = BO->getOperand(0), *Op1BO = BO->getOperand(1);

    // If the division is exact, X % Y is zero, so we end up with X or -X.
    if (PossiblyExactOperator *SDiv = dyn_cast<PossiblyExactOperator>(BO))
      if (SDiv->isExact()) {
        if (Op1BO == Op1C)
          return ReplaceInstUsesWith(I, Op0BO);
        return BinaryOperator::CreateNeg(Op0BO);
      }

    Value *Rem;
    if (BO->getOpcode() == Instruction::UDiv)
      Rem = Builder->CreateURem(Op0BO, Op1BO);
    else
      Rem = Builder->CreateSRem(Op0BO, Op1BO);
    Rem->takeName(BO);

    if (Op1BO == Op1C)
      return BinaryOperator::CreateSub(Op0BO, Rem);
    return BinaryOperator::CreateSub(Rem, Op0BO);
  }
}
```

```
%1 = sdiv i32 %x, %y
%2 = mul i32 %1, %y
  =>
%t = srem i32 %x, %y
%2 = sub i32 %x, %t
```

A Simple Peephole Optimization

```
{
  Value *Op1C = Op1;
  BinaryOperator *BO = dyn_cast<BinaryOperator>(Op0);
  if (!BO ||
      (BO->getOpcode() != Instruction::UDiv &&
       BO->getOpcode() != Instruction::SDiv)) {
    Op1C = Op0;
    BO = dyn_cast<BinaryOperator>(Op1);
  }
  Value *Neg = dyn_castNegVal(Op1C);
  if (BO && BO->hasOneUse() &&
      (BO->getOperand(1) == Op1C || BO->getOperand(1) == Neg) &&
      (BO->getOpcode() == Instruction::UDiv ||
       BO->getOpcode() == Instruction::SDiv)) {
    Value *Op0BO = BO->getOperand(0), *Op1BO = BO->getOperand(1);

    // If the division is exact, X % Y is zero, so we end up with X or -X.
    if (PossiblyExactOperator *SDiv = dyn_cast<PossiblyExactOperator>(BO))
      if (SDiv->isExact()) {
        if (Op1BO == Op1C)
          return ReplaceInstUsesWith(I, Op0BO);
        return BinaryOperator::CreateNeg(Op0BO);
      }

    Value *Rem;
    if (BO->getOpcode() == Instruction::UDiv)
      Rem = Builder->CreateURem(Op0BO, Op1BO);
    else
      Rem = Builder->CreateSRem(Op0BO, Op1BO);
    Rem->takeName(BO);

    if (Op1BO == Op1C)
      return BinaryOperator::CreateSub(Op0BO, Rem);
    return BinaryOperator::CreateSub(Rem, Op0BO);
  }
}
```

```
%1 = sdiv %x, %y
%2 = mul %1, %y
  =>
%t = srem %x, %y
%2 = sub %x, %t
```

A Simple Peephole Optimization

```
{
  Value *Op1C = Op1;
  BinaryOperator *BO = dyn_cast<BinaryOperator>(Op0);
  if (!BO ||
      (BO->getOpcode() != Instruction::UDiv &&
       BO->getOpcode() != Instruction::SDiv)) {
    Op1C = Op0;
    BO = dyn_cast<BinaryOperator>(Op1);
  }
  Value *Neg = dyn_castNegVal(Op1C);
  if (BO && BO->hasOneUse() &&
      (BO->getOperand(1) == Op1C || BO->getOperand(1) == Neg) &&
      (BO->getOpcode() == Instruction::UDiv ||
       BO->getOpcode() == Instruction::SDiv)) {
    Value *Op0BO = BO->getOperand(0), *Op1BO = BO->getOperand(1);

    // If the division is exact, X % Y is zero, so we end up with X or -X.
    if (PossiblyExactOperator *SDiv = dyn_cast<PossiblyExactOperator>(BO))
      if (SDiv->isExact()) {
        if (Op1BO == Op1C)
          return ReplaceInstUsesWith(I, Op0BO);
        return BinaryOperator::CreateNeg(Op0BO);
      }

    Value *Rem;
    if (BO->getOpcode() == Instruction::UDiv)
      Rem = Builder->CreateURem(Op0BO, Op1BO);
    else
      Rem = Builder->CreateSRem(Op0BO, Op1BO);
    Rem->takeName(BO);

    if (Op1BO == Op1C)
      return BinaryOperator::CreateSub(Op0BO, Rem);
    return BinaryOperator::CreateSub(Rem, Op0BO);
  }
}
```

```
Name: sdiv general
%1 = sdiv %x, %y
%2 = mul %1, %y
  =>
%t = srem %x, %y
%2 = sub %x, %t


Name: sdiv exact
%1 = sdiv exact %x, %y
%2 = mul %1, %y
  =>
%2 = %x
```

Alive Language

Precondition

```
Pre: C2 % (1<<C1) == 0
%s = shl nsw %X, C1
%r = sdiv %s, C2
  =>
%r = sdiv %X, C2/(1<<C1)
```

Source template

Target template

Predicates in preconditions may be the result of a dataflow analysis.

Alive Language

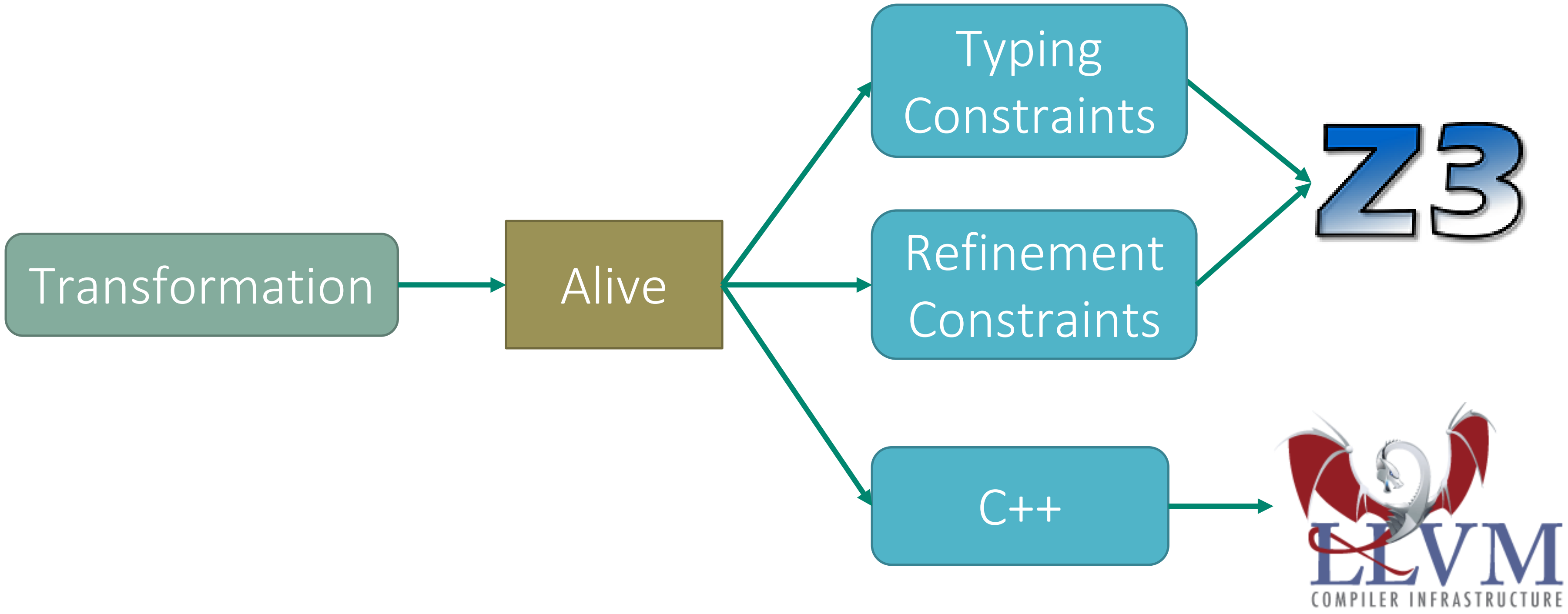
```
Pre: C2 % (1<<C1) == 0
%s = shl nsw %X, C1
%r = sdiv %s, C2
  =>
%r = sdiv %X, C2/(1<<C1)
```

Constants

Generalized from LLVM IR:

- Symbolic constants
- Implicit types

Alive


Transformation
Alive
Typing Constraints
Refinement Constraints
C++
Z3
LLVM
COMPILER INFRASTRUCTURE

Type Inference

- Encode type constraints in SMT
  - Operand have same type
  - Result of trunc has fewer bits than argument

- Find all solutions for the constraint
  - Up to a bounded bitwidth, e.g., 64

Correctness Criteria

1. Target invokes undefined behavior only when the source does
2. Result of target = result of sourc ... source does not invoke undefined behavior
3. Final memory states are equiv

LLVM has 3 types of UB:

- Poison values
- Undef values
- True UB

The story of a new optimization

- A developer wrote a new optimization that improves benchmarks:
  - 3.8% perlbmk (SPEC CPU 2000)
  - 1% perlbench (SPEC CPU 2006)
  - 1.2% perlbench (SPEC CPU 2006) w/ LTO+PGO

40 lines of code

August 2014

The story of a new optimization

- The first patch was wrong

```
Pre: isPowerOf2(C1 ^ C2)
%x = add %A, C1
%i = icmp ult %x, C3
%y = add %A, C2
%j = icmp ult %y, C3
%r = or %i, %j
  =>
%and = and %A, ~(C1 ^ C2)
%lhs = add %and, umax(C1, C2)
%r = icmp ult %lhs, C3
```

```
ERROR: Mismatch in values of %r

Example:
%A i4 = 0x0 (0)
C1 i4 = 0xA (10, -6)
C3 i4 = 0x5 (5)
C2 i4 = 0x2 (2)
%x i4 = 0xA (10, -6)
%i i1 = 0x0 (0)
%y i4 = 0x2 (2)
%j i1 = 0x1 (1, -1)
%and i4 = 0x0 (0)
%lhs i4 = 0xA (10, -6)
Source value: 0x1 (1, -1)
Target value: 0x0 (0)
```

The story of a new optimization

- The second patch was wrong
- The third patch was correct!
- Still fires on the benchmarks!

```
Pre: C1 u> C3 &&
     C2 u> C3 &&
     isPowerOf2(C1 ^ C2) &&
     isPowerOf2(-C1 ^ -C2) &&
     (-C1 ^ -C2) == ((C3-C1) ^ (C3-C2)) &&
     abs(C1-C2) u> C3
%x = add %A, C1
%i = icmp ult %x, C3
%y = add %A, C2
%j = icmp ult %y, C3
%r = or %i, %j
  =>
%and = and %A, ~(C1^C2)
%lhs = add %and, umax(C1,C2)
%r = icmp ult %lhs, C3
```

Experiments

1. Translated > 300 optimizations from LLVM's InstCombine to Alive. Found 8 bugs; remaining proved correct.
2. Automatic optimal post-condition strengthening
   - Significantly better than developers
3. Replaced InstCombine with automatically generated code

InstCombine: Stats per File

File	# opts.	# translated	# bugs
AddSub	67	49	2
AndOrXor	165	131	0
Calls	80	-	-
Casts	77	-	-
Combining	63	-	-
Compares	245	-	-
LoadStoreAlloca	28	17	0
MulDivRem	65	44	6
PHI	12	-	-
Select	74	52	0
Shifts	43	41	0
SimplifyDemanded	75	-	-
VectorOps	34	-	-
Total	1,028	334	8

14% wrong! (← MulDivRem)

Optimal Attribute Inference

```
Pre: C1 % C2 == 0
%m = mul nsw %X, C1
%r = sdiv %m, C2
  =>
%r = mul nsw %X, C1/C2
```

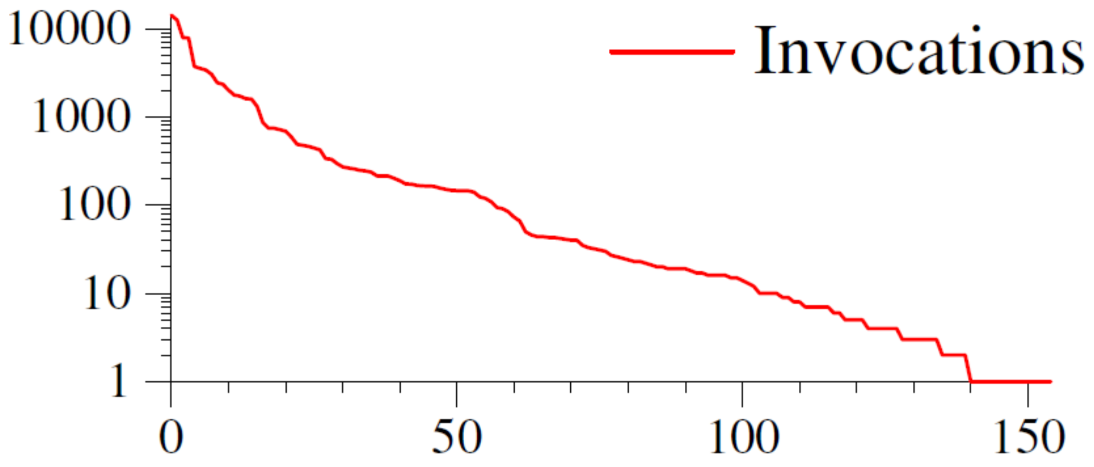
States that the operation will not result in a signed overflow

Optimal Attribute Inference

- Weakened 1 precondition
- Strengthened the postcondition for 70 (21%) optimizations
  - **40% for AddSub, MulDivRem, Shifts**

- Postconditions state, e.g., when an operation will not overflow

Long Tail of Optimizations

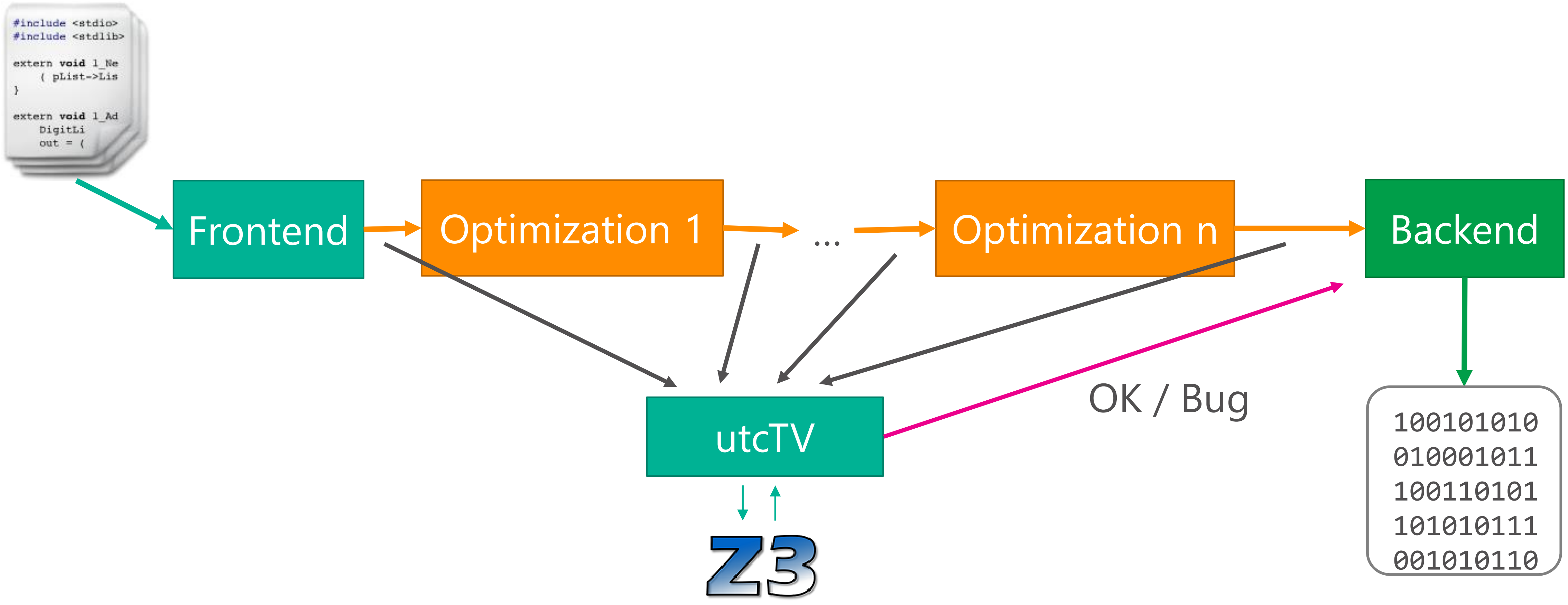


SPEC gives poor code coverage

Alive is Useful!

- Released as open-source in Fall 2014
- In use by developers across 6 companies
- Already caught dozens of bugs in new patches
- Talks about replacing InstCombine

utcTV: Validation for UTC



utcTV: Validation for UTC

- New compiler switch: /d2Verify
- Validates optimizers at the IR/IL level

- No full correctness guarantee (yet) for some cases

utcTV: Early Results

Benchmark	Lines of code	Compile with /d2Verify	Slowdown
bzip2	7k	5 min	106x
gcc	754k	8 hours	186x
gzip	9k	2 min	70x
sqlite3	189k	1 h 20 min	234x
Z3	500k	17 hours	32x

Note: 32-bits, single-threaded compiler


I WANT YOU
TO DO AN
INTERNSHIP
memegenerator.net

Conclusion

- Today's software passes through at least one compiler
- Compilers can introduce bugs and security vulnerabilities in correct programs

- We need reliable compilers
- Alive and utcTV are first steps in this direction

- Can we replicate the success of Static Analysis?

Microsoft Research

Microsoft