Compilation using LLVM

Juan Manuel Martinez Caamaño (@jmmartinez)

Quarkslab

Last course

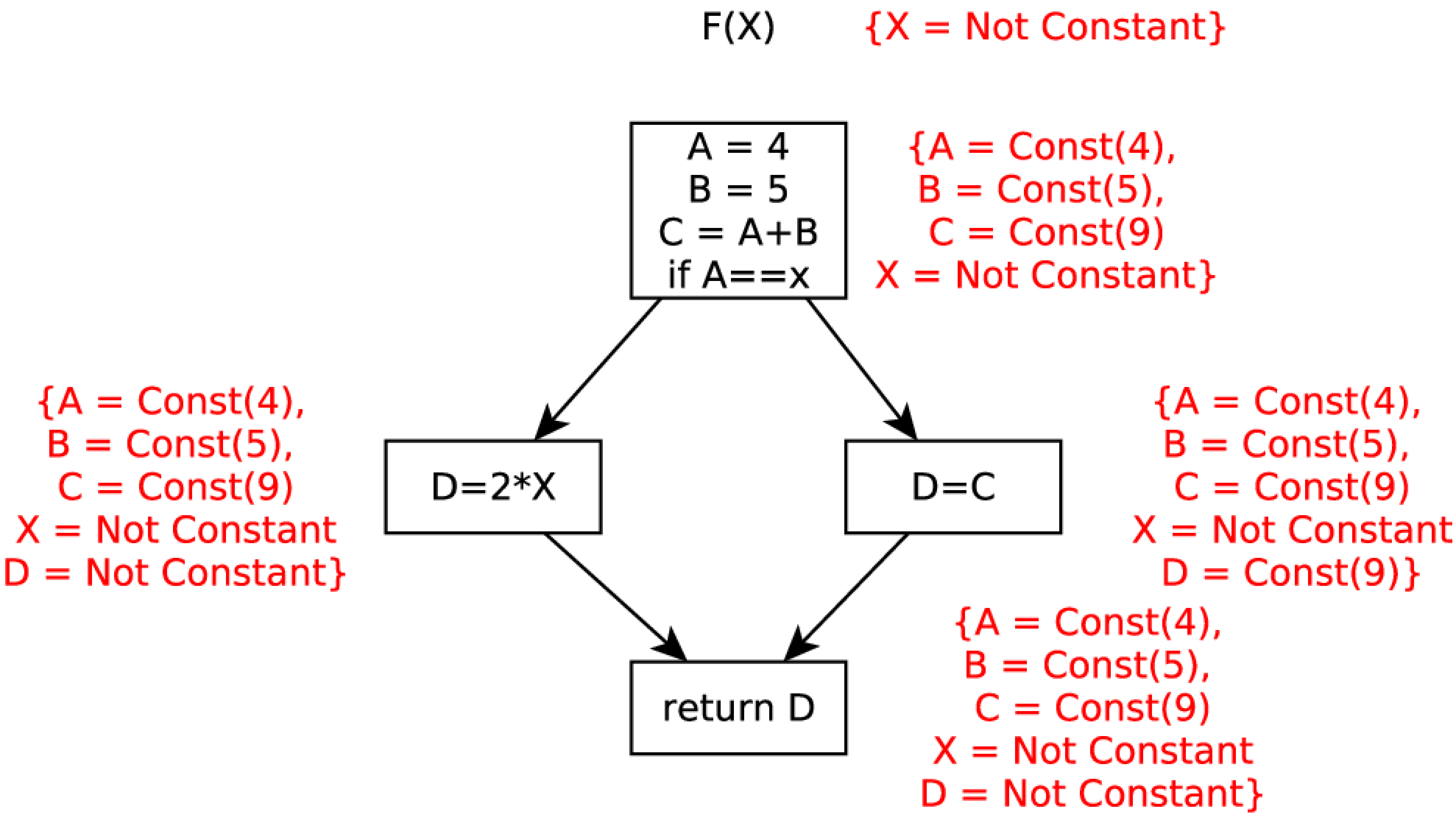
- Overview of the LLVM toolchain
- Start writing the first obfuscations: MBA and OpaqueConstants

Today's objective

- Introduction to static analysis: data-flow analysis
    - Tainting
    - Backwards slice
    - Improving OpaqueConstants

Data-flow Analysis

Constant Propagation



F(X)

{X = Not Constant}

A = 4
B = 5
C = A+B
if A==x

{A = Const(4),
B = Const(5),
C = Const(9)
X = Not Constant}

D=2*X

{A = Const(4),
B = Const(5),
C = Const(9)
X = Not Constant
D = Not Constant}

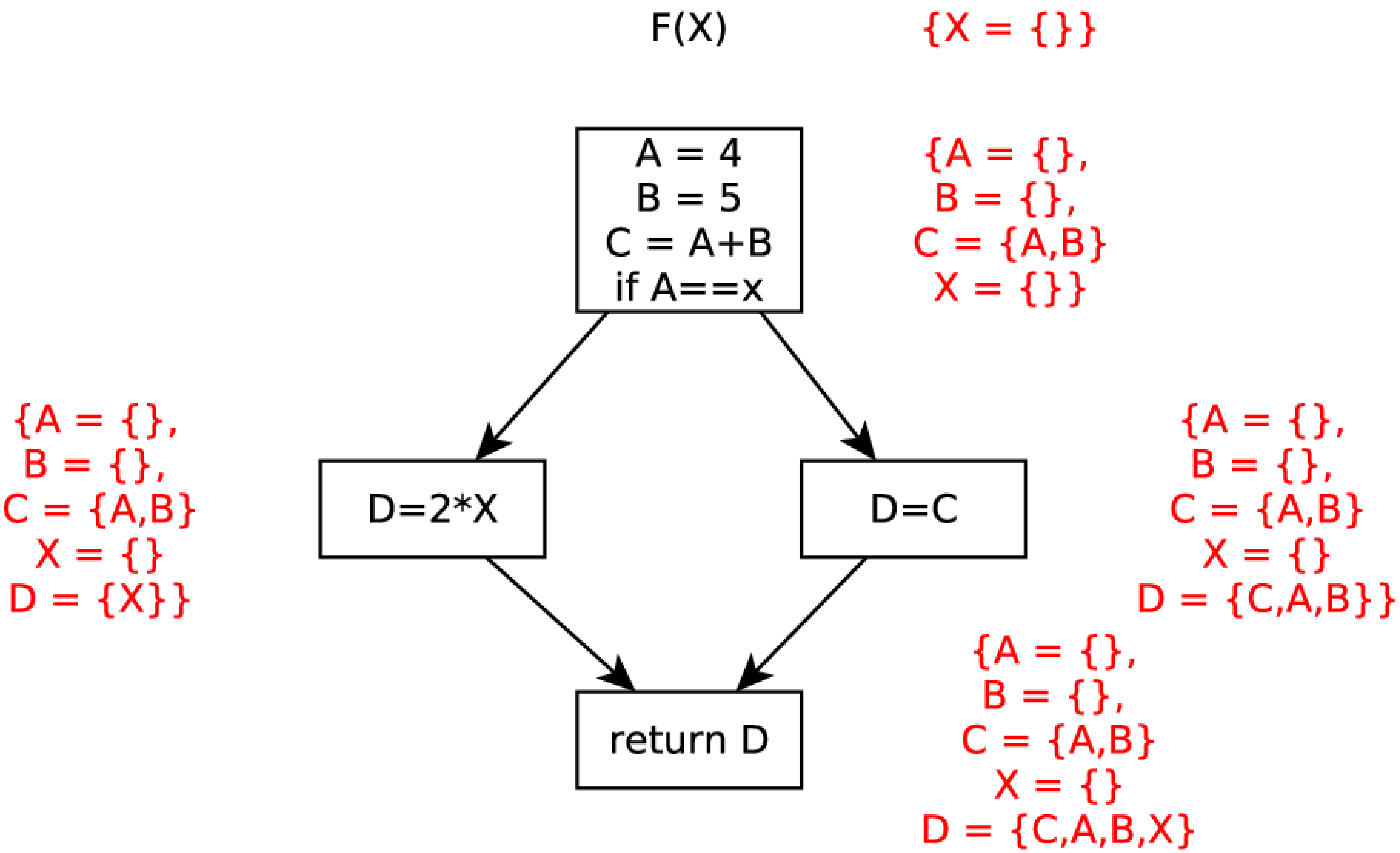
D=C

{A = Const(4),
B = Const(5),
C = Const(9)
X = Not Constant
D = Const(9)}

return D

{A = Const(4),
B = Const(5),
C = Const(9)
X = Not Constant
D = Not Constant}

Tainting & Slicing



Data-flow Analysis

All these analysis form part of a familiy of what's called data-flow analysis.

Techniques that derive information about the flow of data along program execution paths.

Data-flow Analysis

Data-flow Analysis

The execution of a program can be viewed as a transformation on a program state by each instruction.

Consider every possible path in the program.

Data-flow Analysis

Summarize all the program states that may occur at a point in a program.

Do not distinguish among the paths taken to reach a program point.

Data-flow Analysis Schema

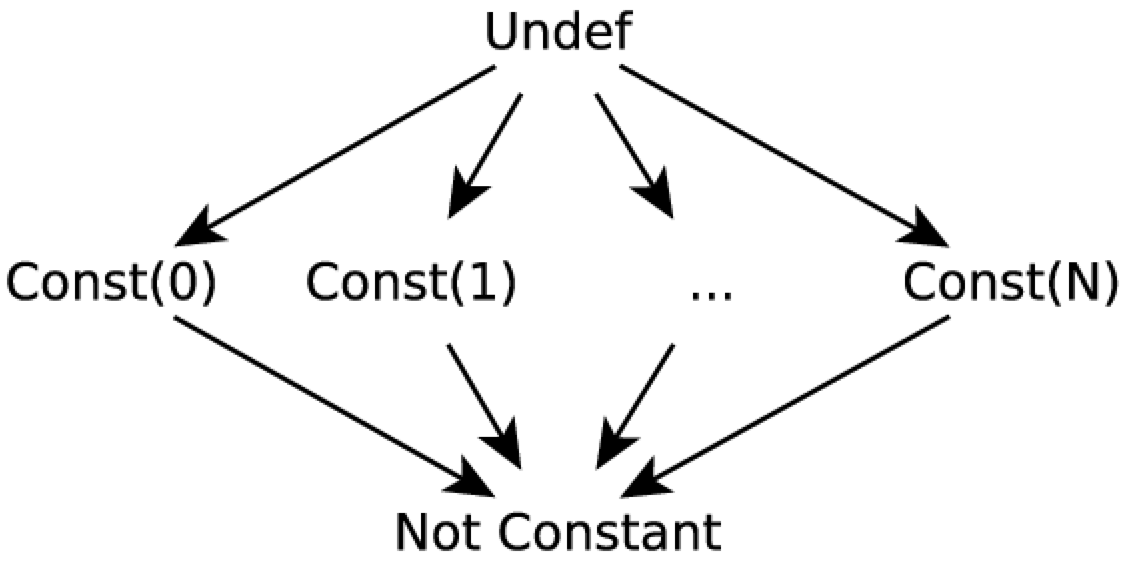
- **Domain**: Associate an abstract value that represent the set of possible program states at each point
- **Meet Operator**: Merges two states. Defines a *partial order*. There is a *Top* element.
- **Transfer Function**: Relate the state of an instruction before and after. Information may be propagated forward or backwards.

When dataflow schemas perform an approximation we must be sure they're "conservative".

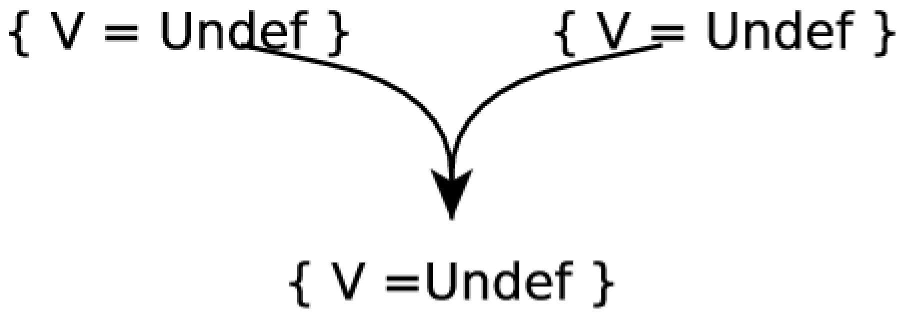
The analysis result is the fixed-point of applying these constraints over a program.

Data-flow Analysis: Constant Propagation

Data-flow Analysis: Constant Propagation



Data-flow Analysis: Constant Propagation

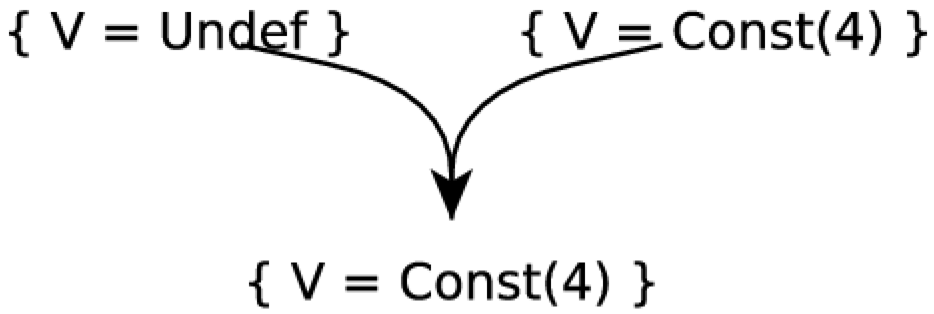


{ V = Undef }

{ V = Undef }

{ V =Undef }

Data-flow Analysis: Constant Propagation

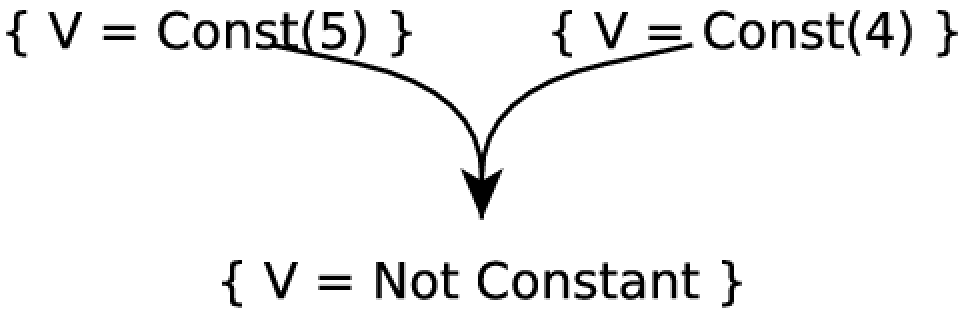


{ V = Undef }

{ V = Const(4) }

{ V = Const(4) }

Data-flow Analysis: Constant Propagation

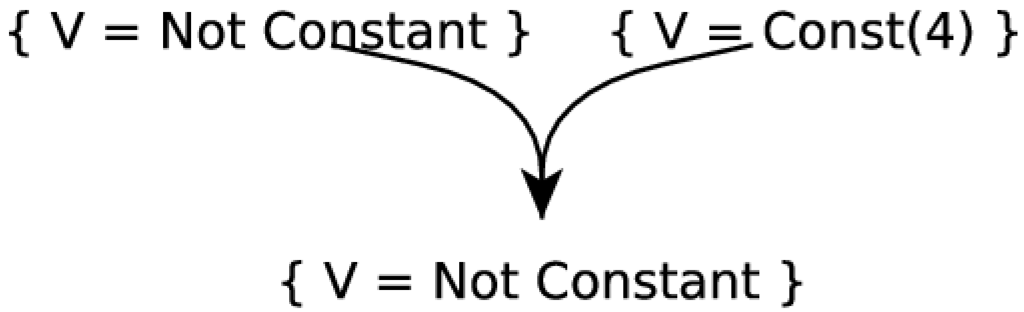


{ V = Const(5) }

{ V = Const(4) }

{ V = Not Constant }

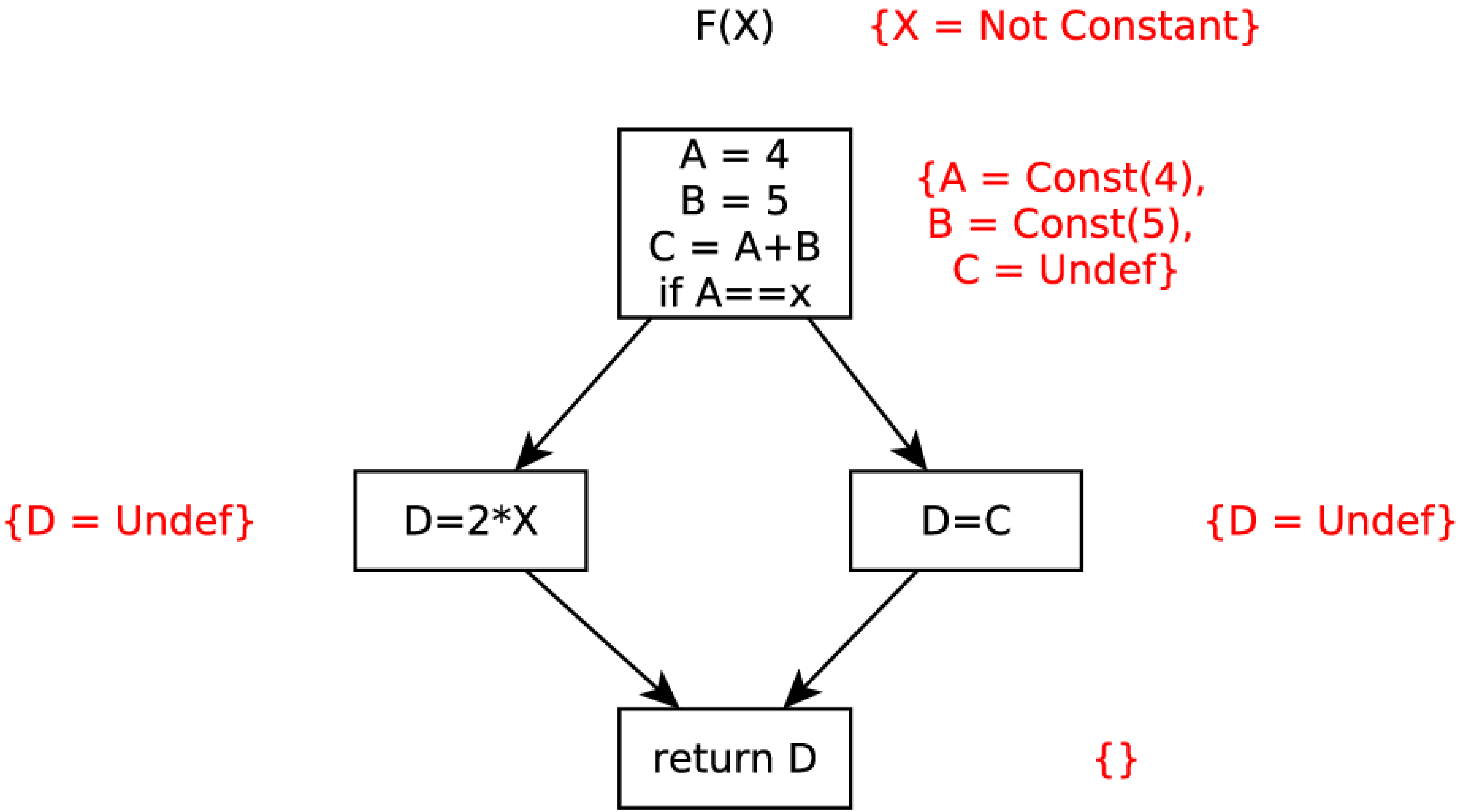
Data-flow Analysis: Constant Propagation



{ V = Not Constant }   { V = Const(4) }

{ V = Not Constant }

Data-flow Analysis: Constant Propagation



F(X) {X = Not Constant}

A = 4
B = 5
C = A+B
if A==x

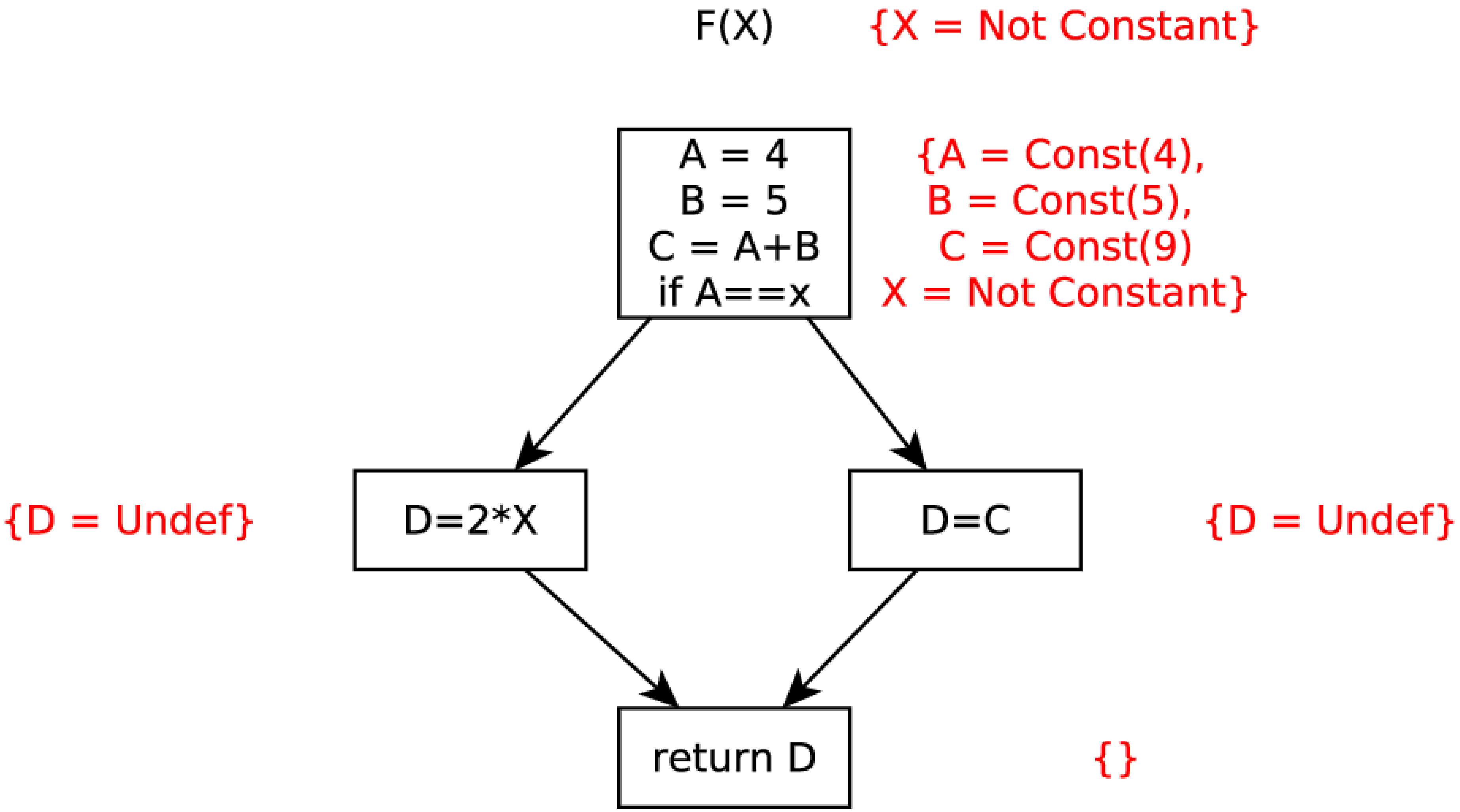
{A = Const(4),
B = Const(5),
C = Undef}

{D = Undef} D=2*X

D=C {D = Undef}

return D {}

Data-flow Analysis: Constant Propagation



F(X)

{X = Not Constant}

A = 4
B = 5
C = A+B
if A==x

{A = Const(4),
B = Const(5),
C = Const(9)
X = Not Constant}

{D = Undef}

D=2*X

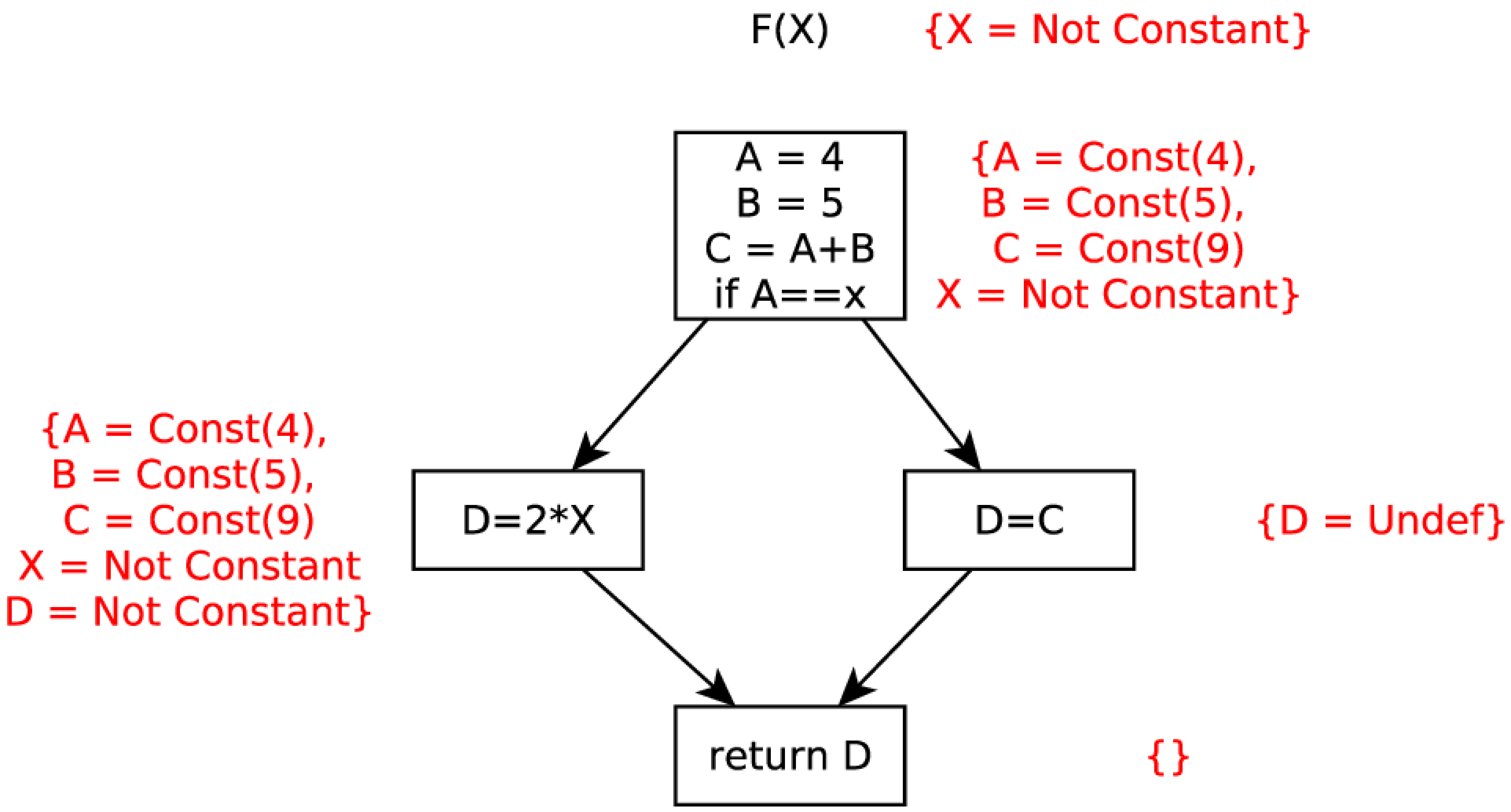
D=C

{D = Undef}

return D

{}

Data-flow Analysis: Constant Propagation



F(X)

{X = Not Constant}

A = 4
B = 5
C = A+B
if A==x

{A = Const(4),
B = Const(5),
C = Const(9)
X = Not Constant}

D=2*X

{A = Const(4),
B = Const(5),
C = Const(9)
X = Not Constant
D = Not Constant}

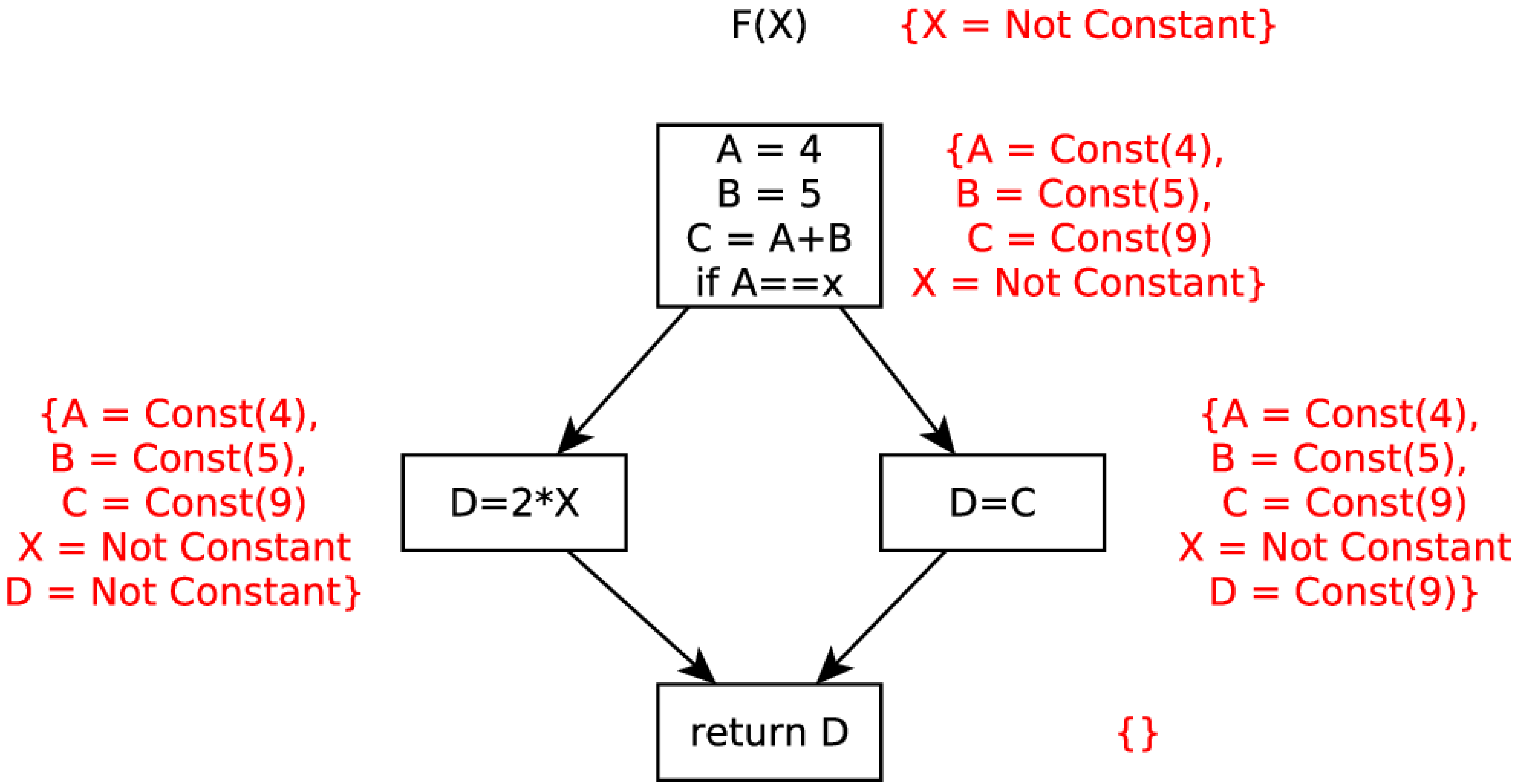
D=C

{D = Undef}

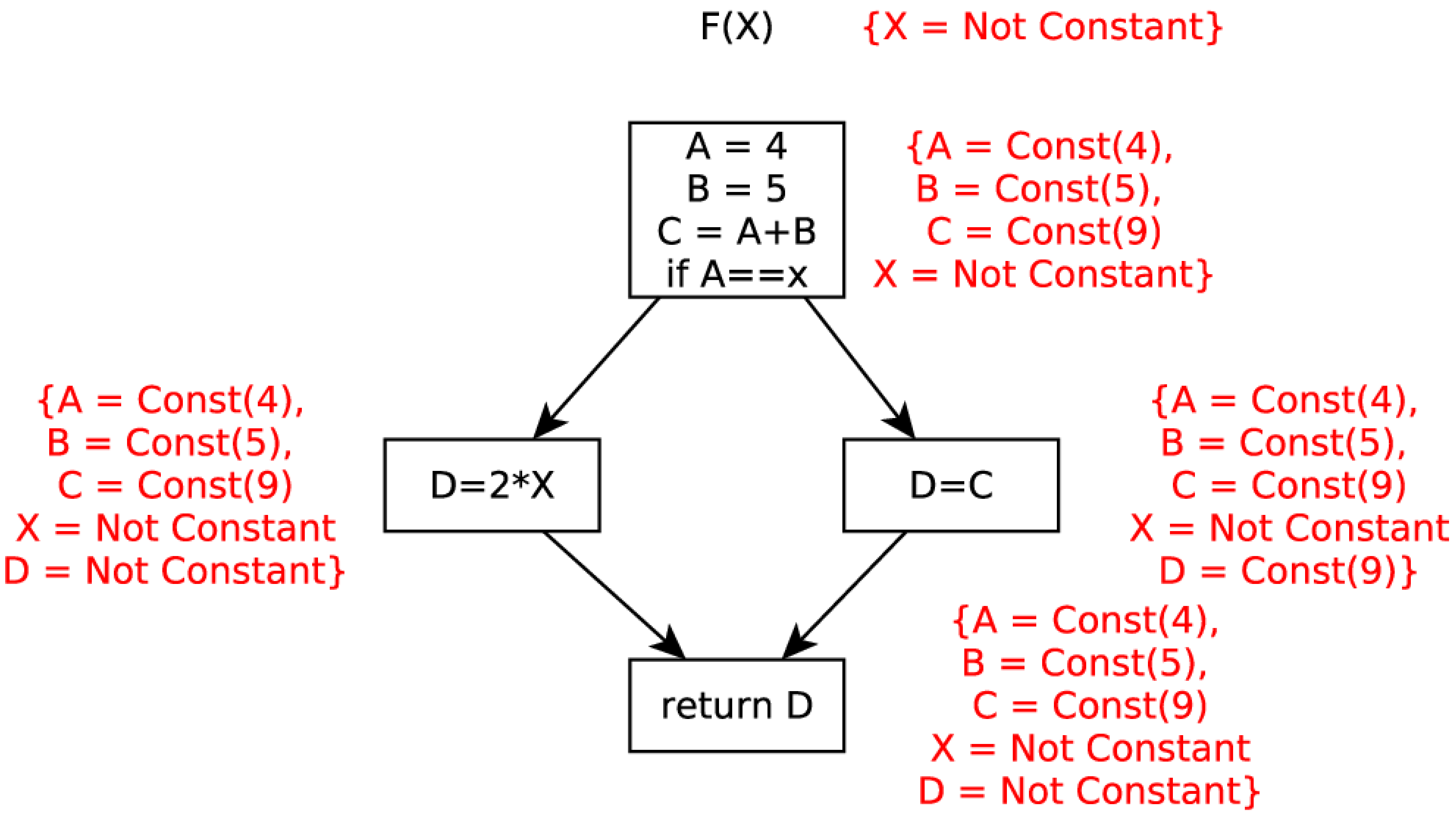
return D

{}

Data-flow Analysis: Constant Propagation



Data-flow Analysis: Constant Propagation



Improving OpaqueConstants

When choosing a live-variable, pick one that is not in the slice of the instruction.

Conclusions

- Data-flow analysis drive conlcusions about all executions from following the data-flow between operations
- Attackers rely on taint-analysis/slicing to reduce the scope of what they're tring to understand
- Use the same analysis to select a transformation that harm their analysis