CS 858: Software Security

Offensive and Defensive Approaches

Detection: symbolic execution

Meng Xu *(University of Waterloo)*

Fall 2022

Outline

1. Introduction
2. Conventional symbolic execution
3. Weakest precondition
4. Loop invariant instrumentation
5. Modeling for mutations (memory model)

Motivation

Q: Why research on symbolic execution when we have unit testing or even fuzzing?

Motivation

Q: Why research on symbolic execution when we have unit testing or even fuzzing?

A: A more complete exploration of program states.

Illustration

```
fn foo(x: u64): u64 {
    if (x * 3 == 42) {
        some_hidden_bug();
    }
    if (x * 5 == 42) {
        some_hidden_bug();
    }
    return 2 * x;
}
```

Illustration

Unit Test

```
foo(0);
foo(1);
```

```
fn foo(x: u64): u64 {
    if (x * 3 == 42) {
        some_hidden_bug();
    }
    if (x * 5 == 42) {
        some_hidden_bug();
    }
    return 2 * x;
}
```

Illustration

```
fn foo(x: u64): u64 {
    if (x * 3 == 42) {
        some_hidden_bug();
    }
    if (x * 5 == 42) {
        some_hidden_bug();
    }
    return 2 * x;
}
```

Unit Test

```
foo(0);
foo(1);
```

Fuzzing

```
foo(0);
foo(1);
foo(12);
foo(78);
......
foo(9,223,372,036,854,775,808);
```

Illustration

```
fn foo(x: u64): u64 {
    if (x * 3 == 42) {
        some_hidden_bug();
    }
    if (x * 5 == 42) {
        some_hidden_bug();
    }
    return 2 * x;
}
```

Unit Test

`foo(0);`

`foo(1);`

Fuzzing

`foo(0);`

`foo(1);`

`foo(12);`

`foo(78);`

`......`

`foo(9,223,372,036,854,775,808);`

Symbolic execution

foo(x)

aborts when $x = 14$

returns $2x$ otherwise

Satisfiability Modulo Theories (SMT)

Definition: A procedure that decides whether a mathematical formula is satisfiable.

Example:

- $3x = 42$
- $2x \geq 2^{64}$
- $5x = 42$

Satisfiability Modulo Theories (SMT)

Definition: A procedure that decides whether a mathematical formula is satisfiable.

Example:

- $3x = 42 \longrightarrow$ satisfiable with $x = 14$
- $2x \geq 2^{64} \longrightarrow$ satisfiable with $x \geq 2^{63}$
- $5x = 42 \longrightarrow$ unsatisfiable, cannot find an x

Ask two question whenever you see a symbolic execution work:

- How does it convert code into mathematical formula?
- What does it try to solve for?

Program Modeling Desiderata

- Control-flow graph exploration
- Loop handling
- Memory modeling
- Concurrency

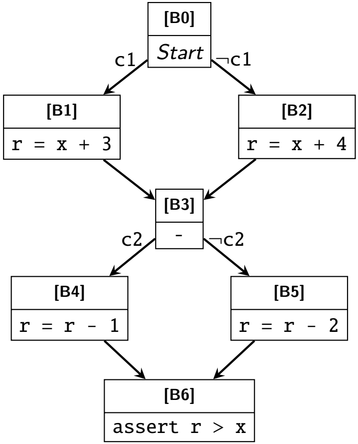
Outline

An example of a pure function

```
fn foo(
    c1: bool, c2: bool,
    x: u64
) -> u64 {
    let r = if (c1) {
        x + 3
    } else {
        x + 4
    };

    let r = if (c2) {
        r - 1
    } else {
        r - 2
    };

    r
}
spec foo {
    ensures r > x;
}
```

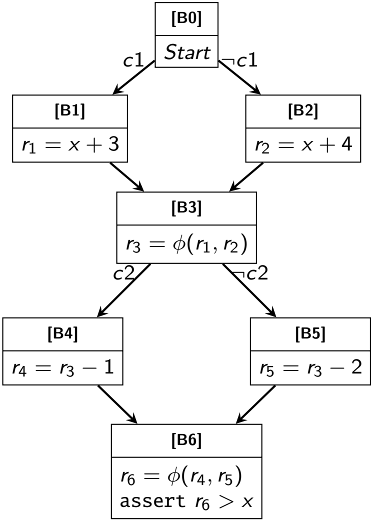
An example of a pure function

```
fn foo(
    c1: bool, c2: bool,
    x: u64
) -> u64 {
    let r = if (c1) {
        x + 3
    } else {
        x + 4
    };

    let r = if (c2) {
        r - 1
    } else {
        r - 2
    };

    r
}
spec foo {
    ensures r > x;
}
```



The example in SSA form

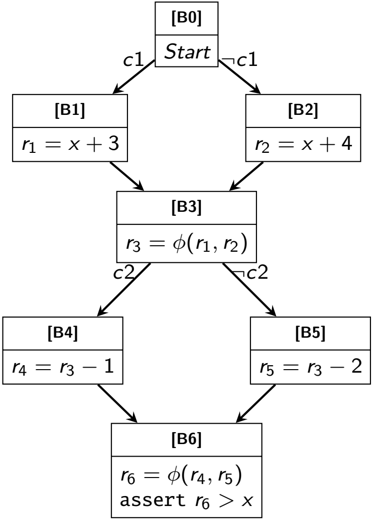
```
fn foo(
    c1: bool, c2: bool,
    x: u64
) -> u64 {
    let r = if (c1) {
        x + 3
    } else {
        x + 4
    };

    let r = if (c2) {
        r - 1
    } else {
        r - 2
    };

    r
}
spec foo {
    ensures r > x;
}
```



Path-based exploration

Vars: $c1$, $c2$, x, r_{1-6}

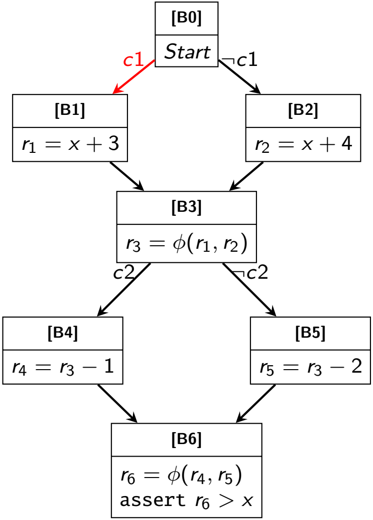
B0	Sym. repr. Path cond.	$\emptyset$ True



Path-based exploration

Vars: $c1$, $c2$, x, r_{1-6}

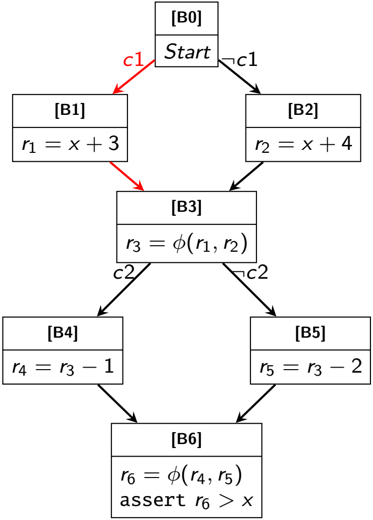
B0	Sym. repr. Path cond.	$\emptyset$ True
B1	Sym. repr. Path cond.	$r_1 = x + 3$ $c1$



Path-based exploration

Vars: $c1$, $c2$, x, r_{1-6}

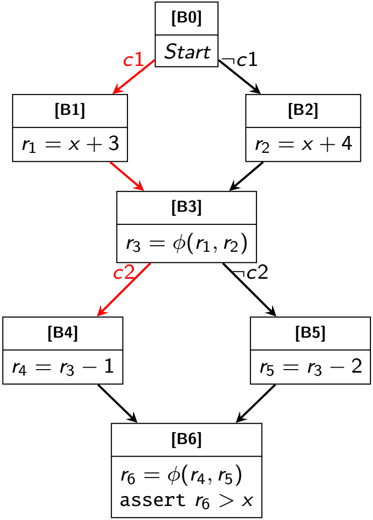
B0	Sym. repr.	$\emptyset$
	Path cond.	True
B1	Sym. repr.	$r_1 = x + 3$
	Path cond.	$c1$
B3	Sym. repr.	$r_1 = x + 3$
		$r_3 = r_1$
	Path cond.	$c1$



Path-based exploration

Vars: $c1$, $c2$, x, r_{1-6}

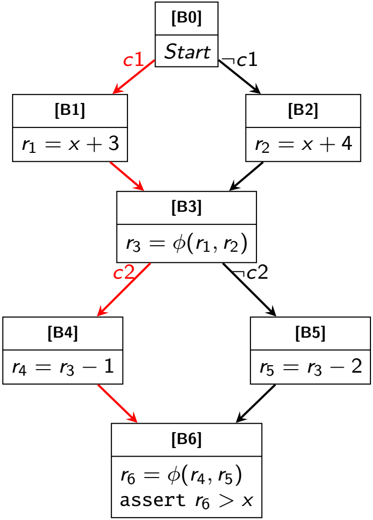
B0	Sym. repr. Path cond.	$\emptyset$ True
B1	Sym. repr. Path cond.	$r_1 = x + 3$ $c1$
B3	Sym. repr. Path cond.	$r_1 = x + 3$ $r_3 = r_1$ $c1$
B4	Sym. repr. Path cond.	$r_1 = x + 3$ $r_3 = r_1$ $r_4 = r_3 - 1$ $c_1 \wedge c_2$



Path-based exploration

Vars: $c1$, $c2$, x, r_{1-6}

B0	Sym. repr.	$\emptyset$
	Path cond.	True
B1	Sym. repr.	$r_1 = x + 3$
	Path cond.	$c1$
B3	Sym. repr.	$r_1 = x + 3$ $r_3 = r_1$
	Path cond.	$c1$
B4	Sym. repr.	$r_1 = x + 3$ $r_3 = r_1$ $r_4 = r_3 - 1$
	Path cond.	$c_1 \wedge c_2$
B6	Sym. repr.	$r_1 = x + 3$ $r_3 = r_1$ $r_4 = r_3 - 1$ $r_6 = r_4$
	Path cond.	$c_1 \wedge c_2$

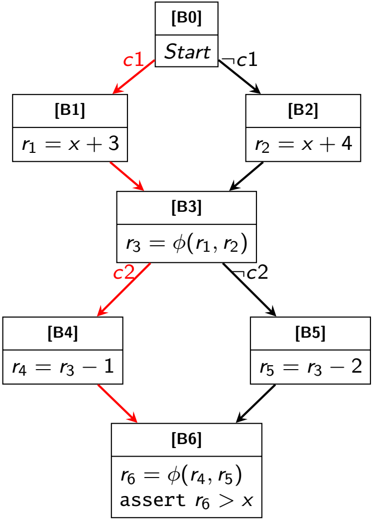


Proving procedure (per path)

Vars: $c1$, $c2$, x, r_{1-6}

B6	Sym. repr.	$r_1 = x + 3$ $r_3 = r_1$ $r_4 = r_3 - 1$ $r_6 = r_4$
	Path cond.	$c_1 \wedge c_2$

$\rightsquigarrow$



Proving procedure (per path)

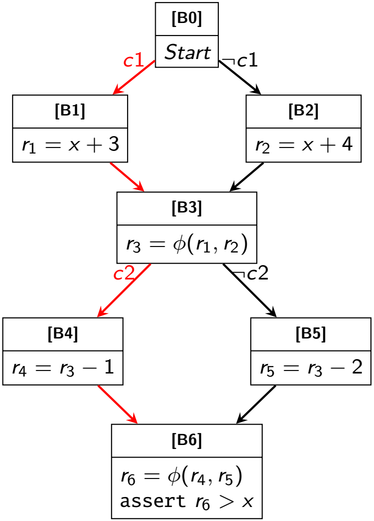
Vars: $c1,\ c2,\ x,\ r_{1-6}$

B6	Sym. repr.	$r_1 = x + 3$
		$r_3 = r_1$
		$r_4 = r_3 - 1$
		$r_6 = r_4$
	Path cond.	$c_1 \wedge c_2$

$\rightsquigarrow$

Prove that $\forall\ c1, c2, x, r_{1-6}$:

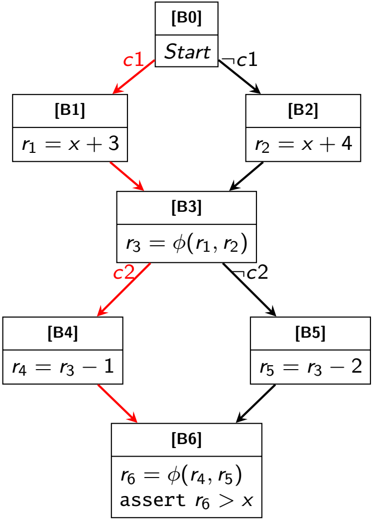
$$
\begin{array}{l}
((c1 \wedge c2) \wedge (\\
\quad (r_1 = x + 3) \\
\quad (r_3 = r_1) \\
\quad (r_4 = r_3 - 1) \\
\quad (r_6 = r_4) \\
)) \Rightarrow (r_6 > x)
\end{array}
$$



Proving procedure (all paths)

Prove that
$\forall\ c1, c2, x, r_{1-6}$:

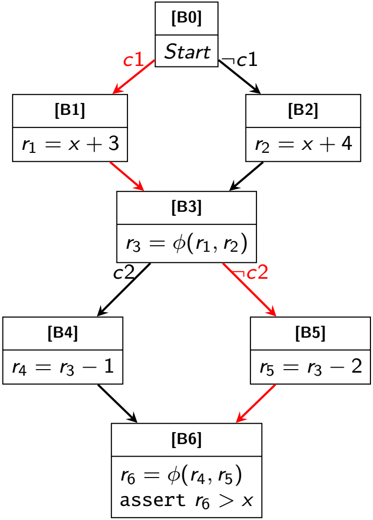
$$((c1 \wedge c2) \wedge ($$
$$(r_1 = x + 3)$$
$$(r_3 = r_1)$$
$$(r_4 = r_3 - 1)$$
$$(r_6 = r_4)$$
$$)) \Rightarrow (r_6 > x)$$



Proving procedure (all paths)

Prove that
$\forall\ c1, c2, x, r_{1-6}$:

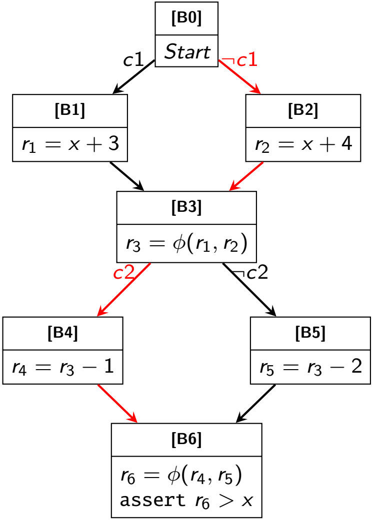
$$((c1 \wedge \neg c2) \wedge ($$
$$(r_1 = x + 3)$$
$$(r_3 = r_1)$$
$$(r_5 = r_3 - 2)$$
$$(r_6 = r_5)$$
$$)) \Rightarrow (r_6 > x)$$



Proving procedure (all paths)

Prove that
$\forall\ c1, c2, x, r_{1-6}$:

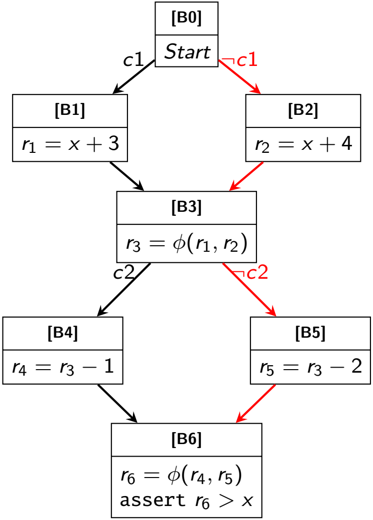
$$((\neg c1 \wedge c2) \wedge ($$
$$(r_2 = x + 4)$$
$$(r_3 = r_2)$$
$$(r_4 = r_3 - 1)$$
$$(r_6 = r_4)$$
$$)) \Rightarrow (r_6 > x)$$



Proving procedure (all paths)

Prove that
$\forall\ c1, c2, x, r_{1-6}$:

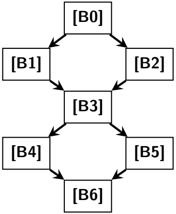
$$((\neg c1 \land \neg c2) \land ($$
$$(r_2 = x + 4)$$
$$(r_3 = r_2)$$
$$(r_5 = r_3 - 2)$$
$$(r_6 = r_5)$$
$$)) \Rightarrow (r_6 > x)$$



Path explosion

Path explosion

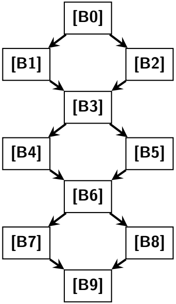
2^{2} paths



Path explosion

2^{2} paths

2^{3} paths



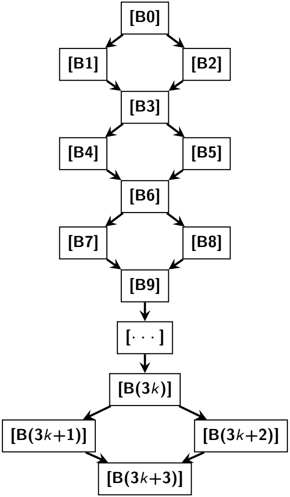
Path explosion

2^{2} paths

2^{3} paths

. . .

2^{k} paths



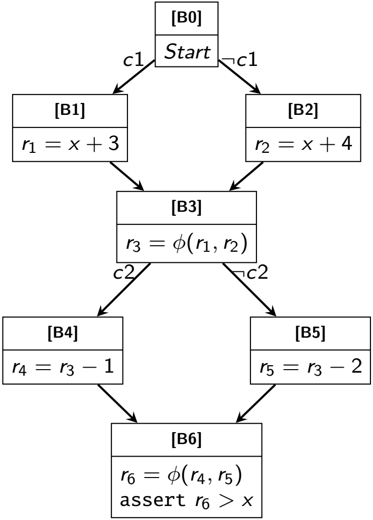
Outline

Weakest precondition calculus

Move prover (Boogie actually) adopts a backward state exploration process, following the weakest precondition calculus.

The running example, once again

```
fn foo(
    c1: bool, c2: bool,
    x: u64
) -> u64 {
    let r = if (c1) {
        x + 3
    } else {
        x + 4
    };

    let r = if (c2) {
        r - 1
    } else {
        r - 2
    };

    r
}
spec foo {
    ensures r > x;
}
```



The passification process

Convert the program into a dynamic single assignment (DSA) form.

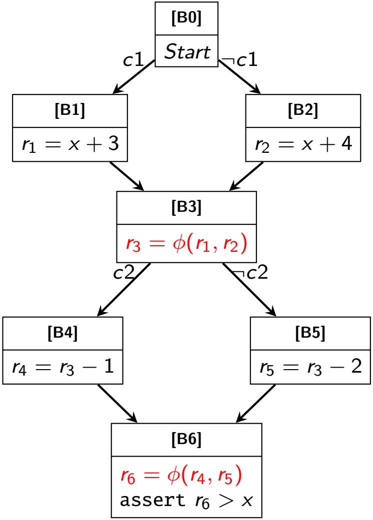
The passification process

Convert the program into a dynamic single assignment (DSA) form.

DSA is extremely similar to static single assignment (SSA) with the ϕ-node eagerly uplifted.

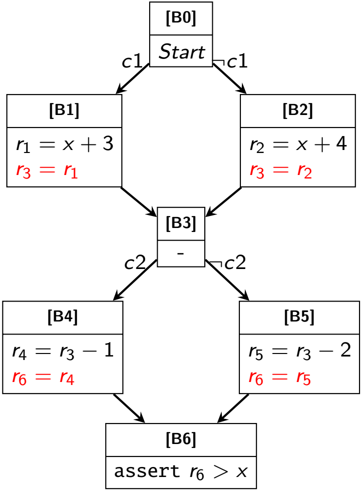
The passification process

```
fn foo(
    c1: bool, c2: bool,
    x: u64
) -> u64 {
    let r = if (c1) {
        x + 3
    } else {
        x + 4
    };

    let r = if (c2) {
        r - 1
    } else {
        r - 2
    };

    r
}
spec foo {
    ensures r > x;
}
```



The passification process

```
fn foo(
    c1: bool, c2: bool,
    x: u64
) -> u64 {
    let r = if (c1) {
        x + 3
    } else {
        x + 4
    };

    let r = if (c2) {
        r - 1
    } else {
        r - 2
    };

    r
}
spec foo {
    ensures r > x;
}
```



The walk-up process

Do a topological sort on the CFG and traverse backward.

The walk-up process

Do a topological sort on the CFG and traverse backward.

This ensures that for each block in the CFG, we visit it *once and only once* (assuming no loops).

The walk-up algorithm

Follow these rules for the intra-block walk-up process:

- $wp(\text{assert } c) = c$
- $wp(\text{assert } c, Q) = c \wedge Q$
- $wp(\text{assign } e, Q) = e \implies Q$
- $wp(s_1; s_2, Q) = wp(s_1, wp(s_2, Q))$

The walk-up algorithm

Follow these rules for the intra-block walk-up process:

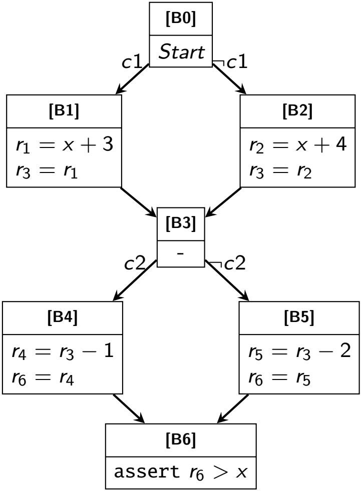
- $wp(\text{assert } c) = c$
- $wp(\text{assert } c, Q) = c \wedge Q$
- $wp(\text{assign } e, Q) = e \implies Q$
- $wp(s_1; s_2, Q) = wp(s_1, wp(s_2, Q))$

The rule for inter-block walk-up is:

$$A \leftarrow wp(s_1; s_2; \ldots; s_n, \bigwedge_{B \in \text{Succ}(A)} B)$$

The walk-up process with an example

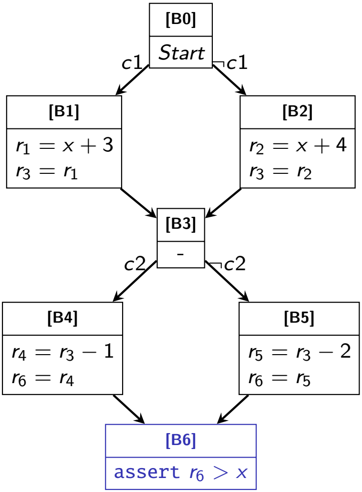
Vars: $c1$, $c2$, x, r_{1-6}, B_{0-6}



The walk-up process with an example

Vars: $c1$, $c2$, x, r_{1-6}, B_{0-6}

$B_6 \leftarrow r_6 > x$

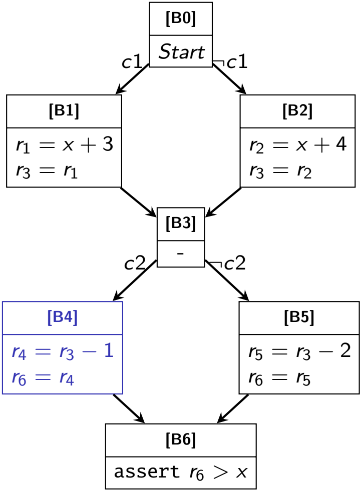


The walk-up process with an example

Vars: $c1$, $c2$, x, r_{1-6}, B_{0-6}

$B_6 \leftarrow r_6 > x$

$B_4 \leftarrow (c2) \Rightarrow ($
$(r_4 = r_3 - 1) \Rightarrow ($
$(r_6 = r_4) \Rightarrow B_6))$



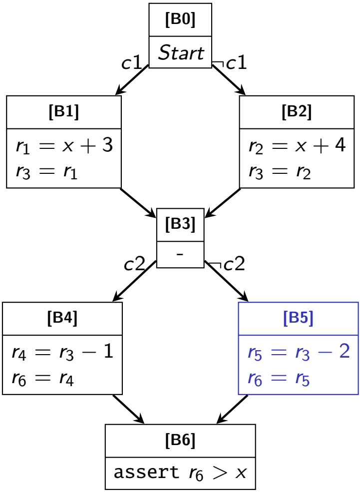
The walk-up process with an example

Vars: $c1,\ c2,\ x,\ r_{1-6},\ B_{0-6}$

$B_6 \leftarrow r_6 > x$

$$B_4 \leftarrow (c2) \Rightarrow ((r_4 = r_3 - 1) \Rightarrow ((r_6 = r_4) \Rightarrow B_6))$$

$$B_5 \leftarrow (\neg c2) \Rightarrow ((r_5 = r_3 - 2) \Rightarrow ((r_6 = r_5) \Rightarrow B_6))$$



The walk-up process with an example

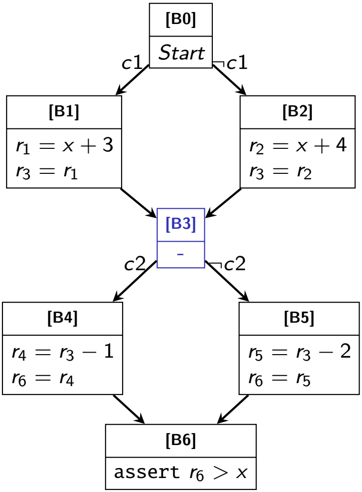
Vars: $c1$, $c2$, x, r_{1-6}, B_{0-6}

$B_6 \leftarrow r_6 > x$

$$B_4 \leftarrow (c2) \Rightarrow ((r_4 = r_3 - 1) \Rightarrow ((r_6 = r_4) \Rightarrow B_6))$$

$$B_5 \leftarrow (\neg c2) \Rightarrow ((r_5 = r_3 - 2) \Rightarrow ((r_6 = r_5) \Rightarrow B_6))$$

$B_3 \leftarrow B_4 \wedge B_5$



The walk-up process with an example

Vars: $c1,\ c2,\ x,\ r_{1-6},\ B_{0-6}$

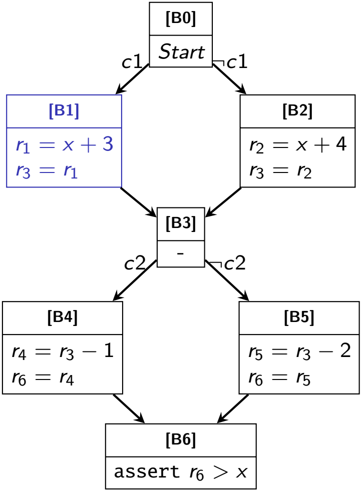
$B_6 \leftarrow r_6 > x$

$$B_4 \leftarrow (c2) \Rightarrow ((r_4 = r_3 - 1) \Rightarrow ((r_6 = r_4) \Rightarrow B_6))$$

$$B_5 \leftarrow (\neg c2) \Rightarrow ((r_5 = r_3 - 2) \Rightarrow ((r_6 = r_5) \Rightarrow B_6))$$

$B_3 \leftarrow B_4 \wedge B_5$

$$B_1 \leftarrow (c1) \Rightarrow ((r_1 = x + 3) \Rightarrow ((r_3 = r_1) \Rightarrow B_3))$$



The walk-up process with an example

Vars: $c1,\ c2,\ x,\ r_{1-6},\ B_{0-6}$

$B_6 \leftarrow r_6 > x$

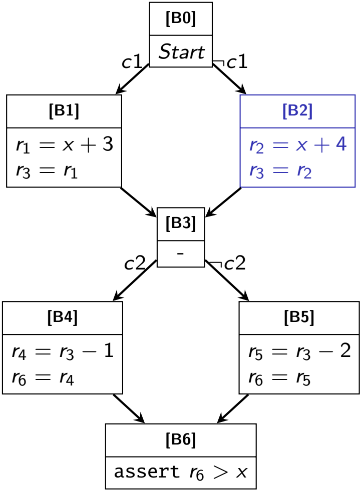
$B_4 \leftarrow (c2) \Rightarrow ($
$\quad (r_4 = r_3 - 1) \Rightarrow ($
$\quad\quad (r_6 = r_4) \Rightarrow B_6))$

$B_5 \leftarrow (\neg c2) \Rightarrow ($
$\quad (r_5 = r_3 - 2) \Rightarrow ($
$\quad\quad (r_6 = r_5) \Rightarrow B_6))$

$B_3 \leftarrow B_4 \wedge B_5$

$B_1 \leftarrow (c1) \Rightarrow ($
$\quad (r_1 = x + 3) \Rightarrow ($
$\quad\quad (r_3 = r_1) \Rightarrow B_3))$

$B_2 \leftarrow (\neg c1) \Rightarrow ($
$\quad (r_2 = x + 4) \Rightarrow ($
$\quad\quad (r_3 = r_2) \Rightarrow B_3))$



The walk-up process with an example

Vars: $c1$, $c2$, x, r_{1-6}, B_{0-6}

$B_6 \leftarrow r_6 > x$

$$B_4 \leftarrow (c2) \Rightarrow ((r_4 = r_3 - 1) \Rightarrow ((r_6 = r_4) \Rightarrow B_6))$$

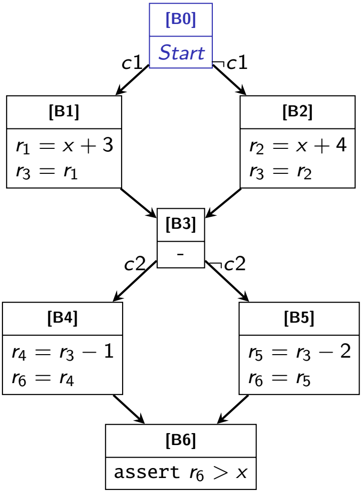
$$B_5 \leftarrow (\neg c2) \Rightarrow ((r_5 = r_3 - 2) \Rightarrow ((r_6 = r_5) \Rightarrow B_6))$$

$B_3 \leftarrow B_4 \wedge B_5$

$$B_1 \leftarrow (c1) \Rightarrow ((r_1 = x + 3) \Rightarrow ((r_3 = r_1) \Rightarrow B_3))$$

$$B_2 \leftarrow (\neg c1) \Rightarrow ((r_2 = x + 4) \Rightarrow ((r_3 = r_2) \Rightarrow B_3))$$

$B_0 \leftarrow B_1 \wedge B_2$

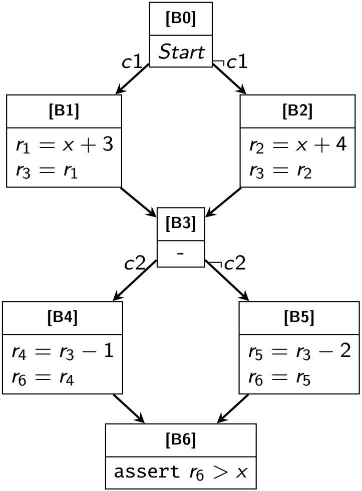


Proving procedure

Prove that
$\forall\ c1, c2, x, r_{1-6}, B_{0-6}$:

$B_6 \leftarrow r_6 > x$
$B_4 \leftarrow (c2) \Rightarrow ($
$\quad (r_4 = r_3 - 1) \Rightarrow ($
$\quad\quad (r_6 = r_4) \Rightarrow B_6))$
$B_5 \leftarrow (\neg c2) \Rightarrow ($
$\quad (r_5 = r_3 - 2) \Rightarrow ($
$\quad\quad (r_6 = r_5) \Rightarrow B_6))$
$B_3 \leftarrow B_4 \wedge B_5$
$B_1 \leftarrow (c1) \Rightarrow ($
$\quad (r_1 = x + 3) \Rightarrow ($
$\quad\quad (r_3 = r_1) \Rightarrow B_3))$
$B_2 \leftarrow (\neg c1) \Rightarrow ($
$\quad (r_2 = x + 4) \Rightarrow ($
$\quad\quad (r_3 = r_2) \Rightarrow B_3))$
$B_0 \leftarrow B_1 \wedge B_2$

$B_0 = \text{True}$



Comparison of forward and backward symbolic execution

Prove that $\forall\ c1, c2, x, r_{1-6}$:

$$((c1 \wedge c2) \wedge ((r_1 = x + 3) (r_3 = r_1) (r_4 = r_3 - 1) (r_6 = r_4))) \Rightarrow (r_6 > x)$$

However, need to repeat this process multiple (worst case exponential) times.

Prove that
$\forall\ c1, c2, x, r_{1-6}, B_{0-6}$:

$$\begin{aligned}
&B_6 \leftarrow r_6 > x \\
&B_4 \leftarrow (c2) \Rightarrow (\\
&\quad (r_4 = r_3 - 1) \Rightarrow (\\
&\quad\quad (r_6 = r_4) \Rightarrow B_6)) \\
&B_5 \leftarrow (\neg c2) \Rightarrow (\\
&\quad (r_5 = r_3 - 2) \Rightarrow (\\
&\quad\quad (r_6 = r_5) \Rightarrow B_6)) \\
&B_3 \leftarrow B_4 \wedge B_5 \\
&B_1 \leftarrow (c1) \Rightarrow (\\
&\quad (r_1 = x + 3) \Rightarrow (\\
&\quad\quad (r_3 = r_1) \Rightarrow B_3)) \\
&B_2 \leftarrow (\neg c1) \Rightarrow (\\
&\quad (r_2 = x + 4) \Rightarrow (\\
&\quad\quad (r_3 = r_2) \Rightarrow B_3)) \\
&B_0 \leftarrow B_1 \wedge B_2
\end{aligned}$$

$B_0 = \text{True}$

Outline

Breaking cycles in the CFG

Loop invariants are keys to break cycles in the CFG

Breaking cycles in the CFG

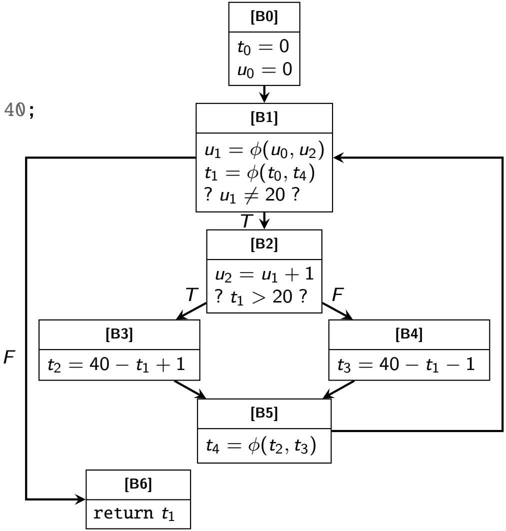
Loop invariants are keys to break cycles in the CFG

A loop invariant is transformed into statements that:

- Assert the invariant at the beginning of the loop
- Havoc (i.e., re-symbolize) the loop induction variables
- Assume the invariant to re-establish relations among the induction variables being havoc-ed
- Assert the invariant at the end of the loop body

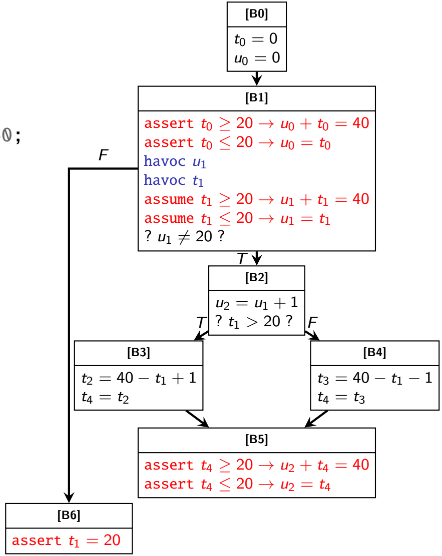
A running example

```
fn bar(): u64 {
    t: u64 = 0;
    u: u64 = 0;
    while ({
spec {
    invariant t >= 20 ==> u + t == 40;
    invariant t <= 20 ==> u == t;
}
        (u != 20)
    }) {
        u = u + 1;
        if (t > 20) {
            t = 40 - t + 1;
        } else {
            t = 40 - t - 1;
        }
    }
    t
}
spec bar {
    ensures result == 20;
}
```



A running example

```
fn bar(): u64 {
    t: u64 = 0;
    u: u64 = 0;
    while ({
spec {
    invariant t >= 20 ==> u + t == 40;
    invariant t <= 20 ==> u == t;
}
        (u != 20)
    }) {
        u = u + 1;
        if (t > 20) {
            t = 40 - t + 1;
        } else {
            t = 40 - t - 1;
        }
    }
    t
}
spec bar {
    ensures result == 20;
}
```



A running example

$$B_6 \leftarrow (u_1 = 20) \Rightarrow (\\ (t_1 = 20))$$

$$B_5 \leftarrow (\\ (t_4 \leq 20 \rightarrow u_2 = t_4) \wedge (\\ t_4 \geq 20 \rightarrow u_2 + t_4 = 40))$$

$$B_4 \leftarrow (t1 \leq 20) \Rightarrow (\\ (t_3 = 40 - t_1 - 1) \Rightarrow (\\ (t_4 = t_3) \Rightarrow B_5))$$

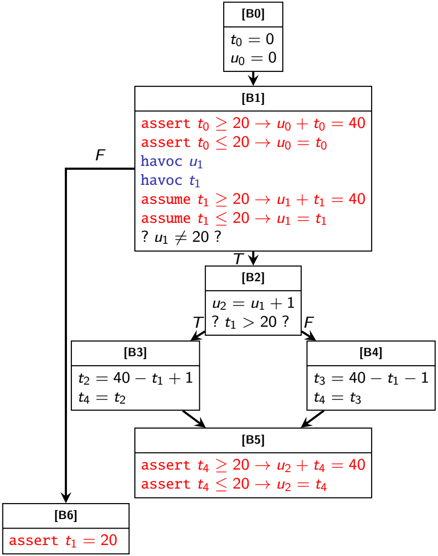
$$B_3 \leftarrow (t1 > 20) \Rightarrow (\\ (t_2 = 40 - t_1 + 1) \Rightarrow (\\ (t_4 = t_2) \Rightarrow B_5))$$

$$B_2 \leftarrow (u_1 \neq 20) \Rightarrow (\\ (u_2 = u_1 + 1) \Rightarrow (\\ B_3 \wedge B4))$$

$$B_1 \leftarrow (\\ (t_0 \geq 20 \rightarrow u_0 + t_0 = 40) \wedge (\\ (t_0 \leq 20 \rightarrow u_0 = t_0) \wedge (\\ (t_1 \geq 20 \rightarrow u_1 + t_1 = 40) \Rightarrow (\\ (t_1 \leq 20 \rightarrow u_1 = t_1) \Rightarrow (\\ B_2 \wedge B6)))))$$

$$B_0 \leftarrow (\\ (t_0 = 0) \Rightarrow (\\ (u_0 = 0) \Rightarrow B_1))$$

Prove that: $B_0 = \text{True}$



Outline

Essence of the borrow semantics

```
fn foo(a: &mut u64, b: &u64) { ... }
```

Essence of the borrow semantics

```
fn foo(a: &mut u64, b: &u64) { ... }
```

- The type system guarantees that `a` and `b` can never alias.
- Regardless of where `a` or `b` is borrowed from, their parents can never change before the lifetime of `a` and `b` ends.

Essence of the borrow semantics

```
fn foo(a: &mut u64, b: &u64) { ... }
```

- The type system guarantees that `a` and `b` can never alias.
- Regardless of where `a` or `b` is borrowed from, their parents can never change before the lifetime of `a` and `b` ends.

The borrow semantics allows Move Prover to eliminate references all together

Mutations under borrow semantics

```
enum Root {
    Param(usize),
    Local(usize),
}

enum Path {
    Field(usize),
    Index(usize),
}

struct Mutation<T> {
    root: Root,
    paths: Vec<Path>,
    value: T,
}
```

Mutations under borrow semantics

```rust
enum Root {
    Param(usize),
    Local(usize),
}

enum Path {
    Field(usize),
    Index(usize),
}

struct Mutation<T> {
    root: Root,
    paths: Vec<Path>,
    value: T,
}
```

```rust
struct S {
    f1: u64,
    f2: u64,
}

fn foo(x: &mut S) {
    let p = &mut x.f1;
    *p = 1;
}
```

```rust
fn _foo_(x: Mutation<S>) -> Mutation<S> {
    Mutation<S> {
        root: x.root, // Root::Param(0)
        paths: x.paths, // vec[]
        value: S {
            f1: 1,
            f2: x.value.f2,
        }
    }
}
```

Simple borrow

```
struct S {
    f1: u64,
    f2: u64,
}

fn foo(x: &mut S) {
    let p = &mut x.f1;
    *p = 1;
}
```

Simple borrow

```
struct S {
    f1: u64,
    f2: u64,
}

fn foo(x: &mut S) {
    let p = &mut x.f1;
    *p = 1;
}
```

```
fn _foo_(x: Mutation<S>) -> Mutation<S> {
    // p := borrow_field<S>.f1(x);
    let p = Mutation<u64> {
        root: x.root, // Param(0),
        paths: concat!(x.paths, Field(0)),
        value: x.value.f1,
    };

    // p2 := write_ref(p, 1);
    let p2 = update!(p, @value = 1);

    // x2 := write_back[x.f1](p2);
    let v = update!(x.value, @f1 = p2.value)
    let x2 = update!(x, @value = v)

    // return x2;
    x2
}
```

Conditional borrow

```
struct S {
    f1: u64,
    f2: u64,
}

fn foo(b: bool, x: &mut S) {
    let p = if b {
        &mut x.f1
    } else {
        &mut x.f2
    };

    *p = 1;
}
```

Conditional borrow

```
struct S {
    f1: u64,
    f2: u64,
}

fn foo(b: bool, x: &mut S) {
    let p = if b {
        &mut x.f1
    } else {
        &mut x.f2
    };

    *p = 1;
}
```

```
fn _foo_(b: bool, x: Mutation<S>) -> Mutation<S> {
    let p = if b {
        // p := borrow_field<S>.f1(x);
        Mutation<u64> {
            root: x.root,
            paths: concat!(x.paths, Field(0)),
            value: x.value.f1,
        }
    } else {
        // p := borrow_field<S>.f2(x);
        Mutation<u64> {
            root: x.root,
            paths: concat!(x.paths, Field(1)),
            value: x.value.f2,
        }
    };

    // p2 := write_ref(p, 1);
    let p2 = update!(p, @value = 1);

    // to be continued
    // ......
```

Conditional borrow

```
struct S {
    f1: u64,
    f2: u64,
}

fn foo(b: bool, x: &mut S) {
    let p = if b {
        &mut x.f1
    } else {
        &mut x.f2
    };

    *p = 1;
}
```

```
fn _foo_(b: bool, x: Mutation<S>) -> Mutation<S> {
    // ......
    // continued from above

    // is_parent(x.f1, p2)
    if p2.root == x.root &&
        p2.paths == concat!(x.paths, Field(0)) {
        // x2 := write_back[x.f1](p2);
        let v = update!(x.value, @f1 = p2.value)
        let x2 = update!(x, @value = v)
    }

    // is_parent(x.f2, p2)
    if p2.root == x.root &&
        p2.paths == concat!(x.paths, Field(1)) {
        // x2 := write_back[x.f1](p2);
        let v = update!(x.value, @f2 = p2.value)
        let x2 = update!(x, @value = v)
    }

    // return x2;
    x2
}
```

Conditional borrow (multiple)

```
struct S {f1: u64, f2: u64}
struct R {s1: S, s2: S}

fn foo(
    a: bool, b: bool,
    x: &mut R,
) {
    let p = if a {
        &mut x.s1
    } else {
        &mut x.s2
    };

    let q = if b {
        &mut p.f1
    } else {
        &mut p.f2
    };

    *q = 1;
}
```

Conditional borrow (multiple)

```
struct S {f1: u64, f2: u64}
struct R {s1: S, s2: S}

fn foo(
    a: bool, b: bool,
    x: &mut R,
) {
    let p = if a {
        &mut x.s1
    } else {
        &mut x.s2
    };

    let q = if b {
        &mut p.f1
    } else {
        &mut p.f2
    };

    *q = 1;
}
```

```
fn _foo_(a: bool, b: bool, x: Mutation<R>)
        -> Mutation<R> {
    let p = if a {
        // borrow_field<R>.s1(x);
    } else {
        // borrow_field<R>.s2(x);
    };

    let q = if b {
        // borrow_field<S>.f1(p);
    } else {
        // borrow_field<S>.f2(p);
    };

    // q2 = write_ref(q, 1);

    // to be continued
    // ......
```

Conditional borrow (multiple)

```
struct S {f1: u64, f2: u64}
struct R {s1: S, s2: S}

fn foo(
    a: bool, b: bool,
    x: &mut R,
) {
    let p = if a {
        &mut x.s1
    } else {
        &mut x.s2
    };

    let q = if b {
        &mut p.f1
    } else {
        &mut p.f2
    };

    *q = 1;
}
```

```
fn _foo_(a: bool, b: bool, x: Mutation<R>)
        -> Mutation<R> {
    // is_parent(p.f1, q2);
    if q2.root == p.root &&
        q2.paths == concat!(p.paths, Field::(0)) {
        // p2 = write_back[p.f1](q2);
    }
    // is_parent(p.f2, q2);
    if q2.root == p.root &&
        q2.paths == concat!(p.paths, Field::(1)) {
        // p2 = write_back[p.f2](q2);
    }
    // is_parent(x.s1, p2);
    if p2.root == x.root &&
        p2.paths == concat!(x.paths, Field::(0)) {
        // x2 = write_back[x.s1](p2);
    }
    // is_parent(p.s2, q2);
    if p2.root == x.root &&
        p2.paths == concat!(x.paths, Field::(1)) {
        // x2 = write_back[x.s2](p2);
    }
    // return x2
}
```

Borrow through function calls

```
struct S {
    f1: u64,
    f2: u64,
}

fn bar(b: bool, x: &mut S)
        -> &mut u64 {
    if b {
        &mut x.f1
    } else {
        &mut x.f2
    }
}

fn foo(b: bool, x: &mut S) {
    let p = bar(b, x);
    *p = 1;
}
```

Borrow through function calls

```
struct S {
    f1: u64,
    f2: u64,
}

fn bar(b: bool, x: &mut S)
        -> &mut u64 {
    if b {
        &mut x.f1
    } else {
        &mut x.f2
    }
}

fn foo(b: bool, x: &mut S) {
    let p = bar(b, x);
    *p = 1;
}
```

```
fn _foo_(b: bool, x: Mutation<S>) -> Mutation<S> {
    let p = _bar_(b, x);
    // p2 := write_ref(p, 1);
    let p2 = update!(p, @value = 1);

    // is_parent(x.f1, p2)
    if p2.root == x.root &&
        p2.paths == concat!(x.paths, Field(0)) {
        // x2 := write_back[x.f1](p2);
        let v = update!(x.value, @f1 = p2.value)
        let x2 = update!(x, @value = v)
    }
    // is_parent(x.f2, p2)
    if p2.root == x.root &&
        p2.paths == concat!(x.paths, Field(1)) {
        // x2 := write_back[x.f1](p2);
        let v = update!(x.value, @f2 = p2.value)
        let x2 = update!(x, @value = v)
    }

    // return x2;
    x2
}
```

$\langle$ **End** $\rangle$