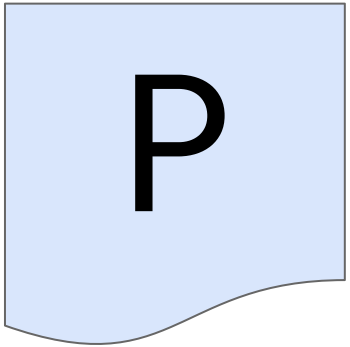
Medusa

Mutant Equivalence Detection Using SAT Analysis

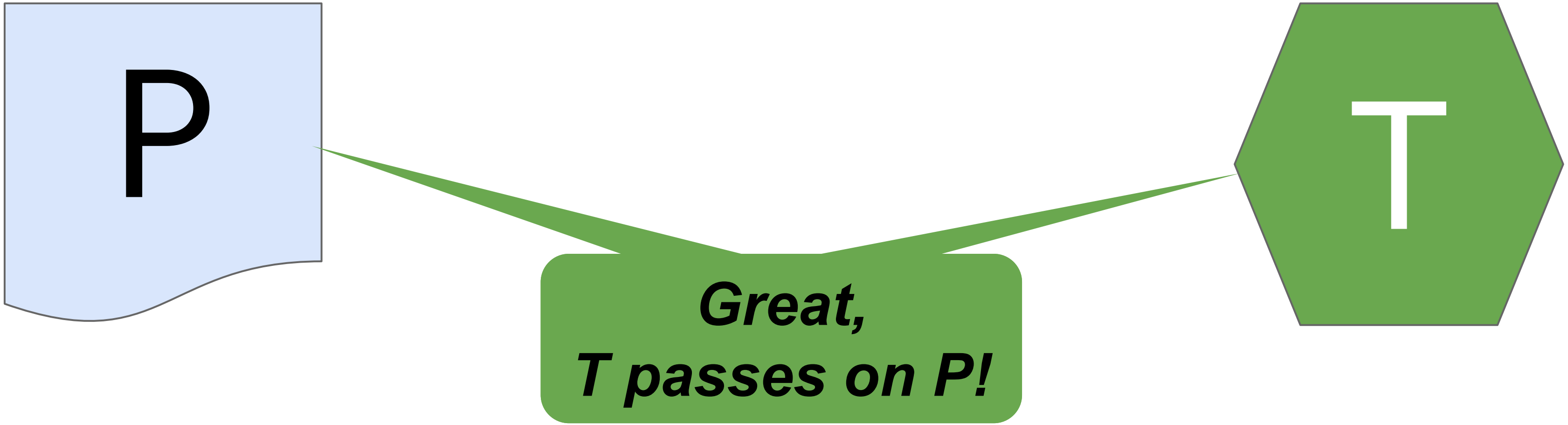
Benjamin Kushigian[1,2], Amit Rawat[1], René Just[2]

[1] UMass Amherst [2] University of Washington

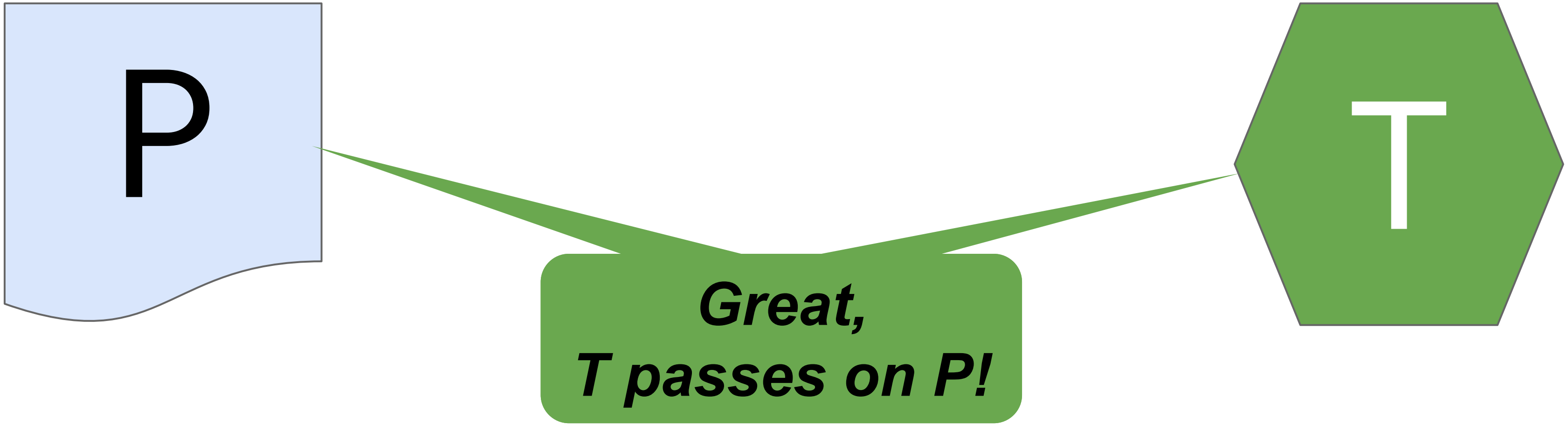
Question: Is our program correct?



Let's test its correctness

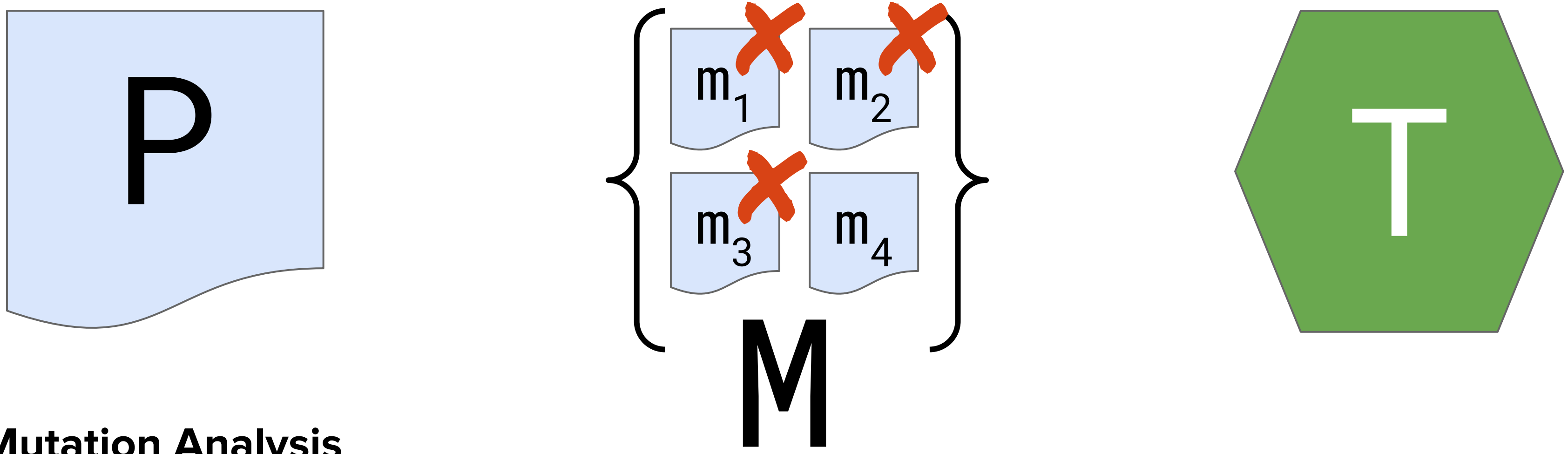


Question: how good is our test suite?



Let's analyze T's effectiveness

Let's analyze T’s effectiveness



Mutation Analysis

- Generate mutant set *M*
- Run *T* on mutants in *M*
- Each mutant that fails a test in *T* is *killed*

Mutation Score

For test suite T and mutant set M, the ***mutation score*** of T is

$$\mu(T) = \frac{\left|\left\{ \text{Killed mutants in } M \right\}\right|}{\left|\left\{ \text{Nonequivalent mutants in } M \right\}\right|}$$

Mutation Score

For test suite T and mutant set M, the ***mutation score*** of T is

$$\mu(T) = \frac{\left|\left\{ \text{Killed mutants in } M \right\}\right|}{\left|\left\{ \text{Nonequivalent mutants in } M \right\}\right|}$$

UNDECIDABLE

A common proxy: Mutation Kill Ratio

For test suite T and mutant set M, the ***mutant kill ratio*** of T is

$$\rho(T) = \frac{\left|\left\{\text{ Killed mutants in } M \right\}\right|}{\left|\left\{\cancel{\text{ Nonequivalent mutants in } M}\right\}\right|}$$

Mutants in M not known to be equivalent

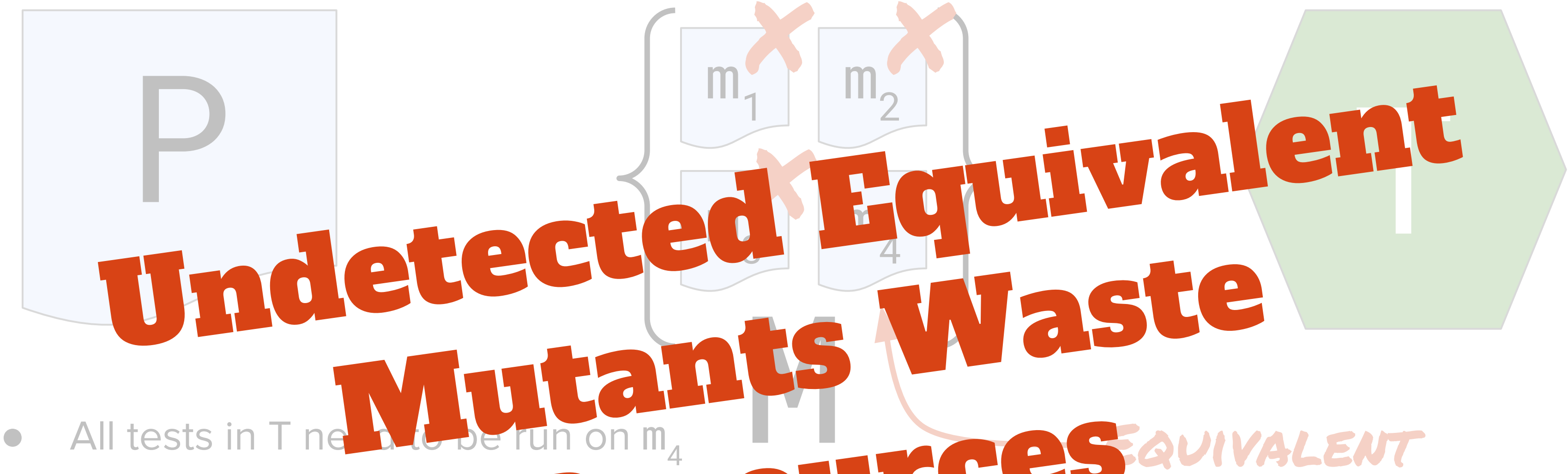
DECIDABLE

Mutation Kill Ratio vs. Mutation Score

- $\rho(T)$ approximates $\mu(T)$
- equivalent mutants are detecte
- undetected equivalent utants, the worse $\rho(T)$ approximates $\mu(T)$

Undetected Equivalent Mutants Skew Analysis Results

Say we have an equivalent mutant



Undetected Equivalent Mutants Waste Resources

- All tests in T need to be run on m_4
  - m_4 is equivalent: it passes all tests
  - This wastes computer resources
- A human developer tries to kill m_4
  - This wastes the developer's time

Detecting Equivalent Mutants

Detecting Equivalent Mutants

There are a number of ways to detect equivalent mutants

1. Hand proof
2. Compiler techniques
3. **Automated reasoning tools**

Detecting Equivalent Mutants

Two Main Challenges

- **Applicability:** What programs can a system reason about?
- **Efficiency**: How many resources does a system need?

Applicability: What can be reasoned about?

Applicability challenges:

- Loops and recursion
  - Loop unrolling can witness non-equivalence but cannot prove equivalence in general

Applicability: What can be reasoned about?

Applicability challenges:

- Loops and recursion
  - Loop unrolling can witness non-equivalence but cannot prove equivalence in general
- Heap space

Applicability: What can be reasoned about?

Applicability challenges:

- Loops and recursion
  - Loop unrolling can witness non-equivalence but cannot prove equivalence in general
- Heap space

Approaches to increase applicability (future work described in the paper):

- Middle Out Constraint Generation
- Exception Abstraction
- Foldability: is `(fold f xs acc)` equivalent to `(fold f' xs acc)`?

Efficiency: How many resources are needed?

Efficiency challenge:

- Solving NP-hard problems such as SAT

Medusa improves efficiency with **constraint forking**

- Decreases problem size
- Reuses work done by SMT solver

Proving Program Equivalence

Question: are these equivalent?

2 = 2

```
int squarePos(int x) {
    if (x > 0) x = x * x;
    return x;
}
```

```
int squarePos(int x) {
    if (x <= 0) x = x * x;
    return x;
}
```

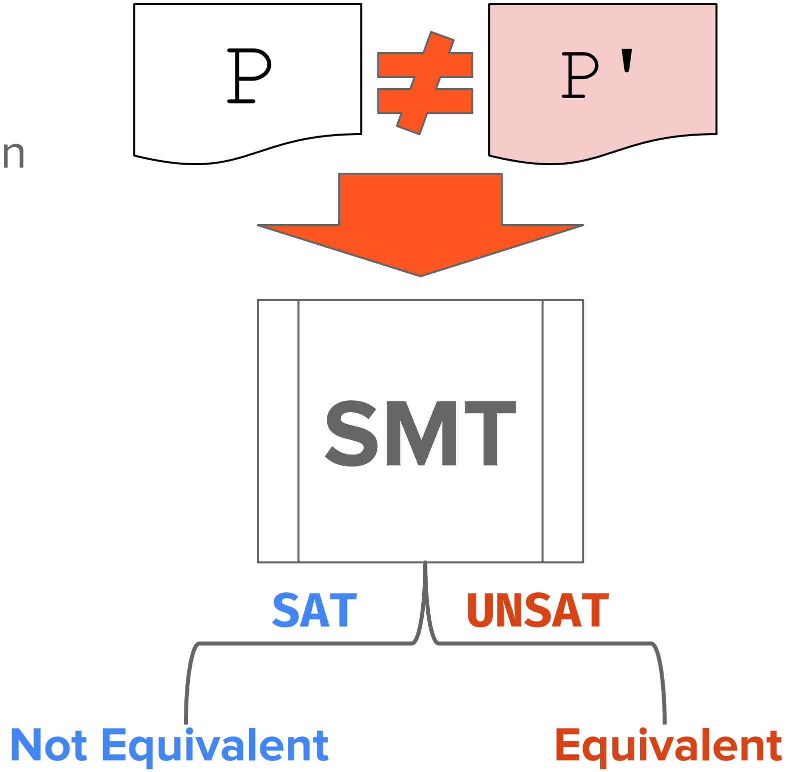
4 ≠ 2

ORIGINAL NOT EQUIVALENT MUTANT

Question: How to (dis)prove equivalence?

- Transform query to SMT *constraints*
  - Constrain both programs' execution
  - Assert inputs are equal
  - Assert outputs are different

- Ask SMT if constraints are satisfiable
  - **`SAT => Not Equivalent`**
  - **`UNSAT => Equivalent`**
  - May also be **`UNKNOWN`**



Question: How to constrain Java?

Observation

- Java is complicated
- No formal semantics
- Semantics defined by Javac/JVM

We Should:

- Compile program
- Constrain bytecode

Question: How to constrain bytecode?

Two types of bytecode instructions

Straight Line

- **Operand Stack Instructions**
  `iadd, imul, push, pop...`
- **Variables Table Instructions**
  `istore, iload, ...`

Branching

- **Control Flow Instructions**
  `ifgt, iflt, if_cmpgt, ...`

Distinction captured by Control Flow Graphs

High Level Overview

Given a program and a mutant:

```
int squarePos(int x) {
    if (x > 0) x = x * x;
    return x;
}
```

```
int squarePos(int x) {
    if (x <= 0) x = x * x;
    return x;
}
```

ORIGINAL

MUTANT

High Level Overview

Given a program and a mutant:

1. Compile to bytecode

```
0: iload_1
1: ifle 8
4: iload_1
5: iload_1
6: imul
7: istore_1
8: iload_1
9: ireturn
```

ORIGINAL

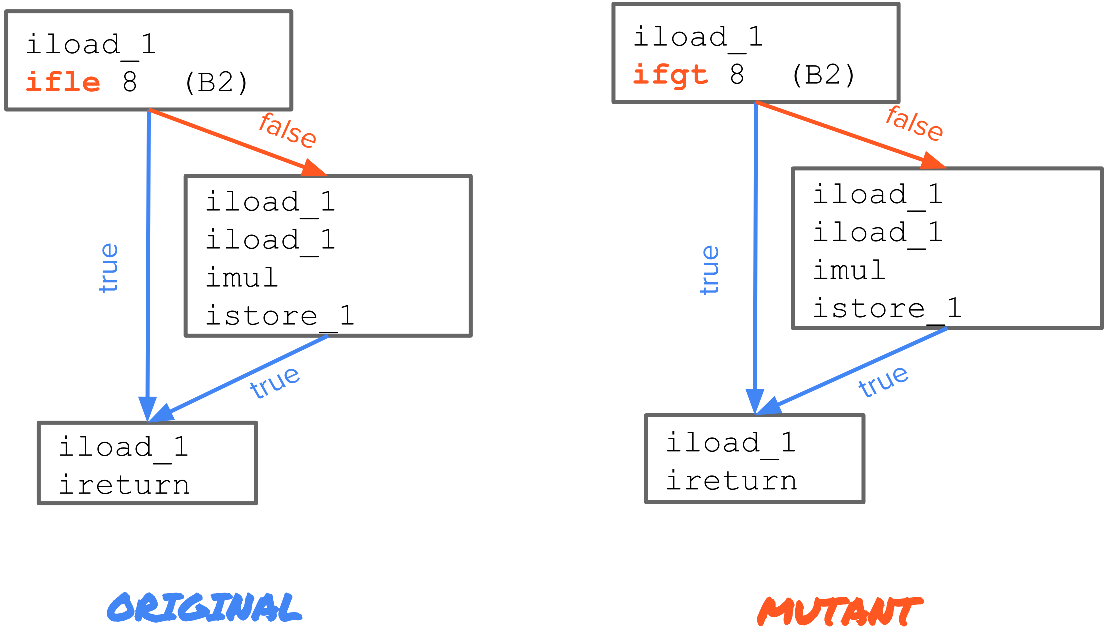
```
0: iload_1
1: ifgt 8
4: iload_1
5: iload_1
6: imul
7: istore_1
8: iload_1
9: ireturn
```

MUTANT

High Level Overview

Given a program and a mutant:

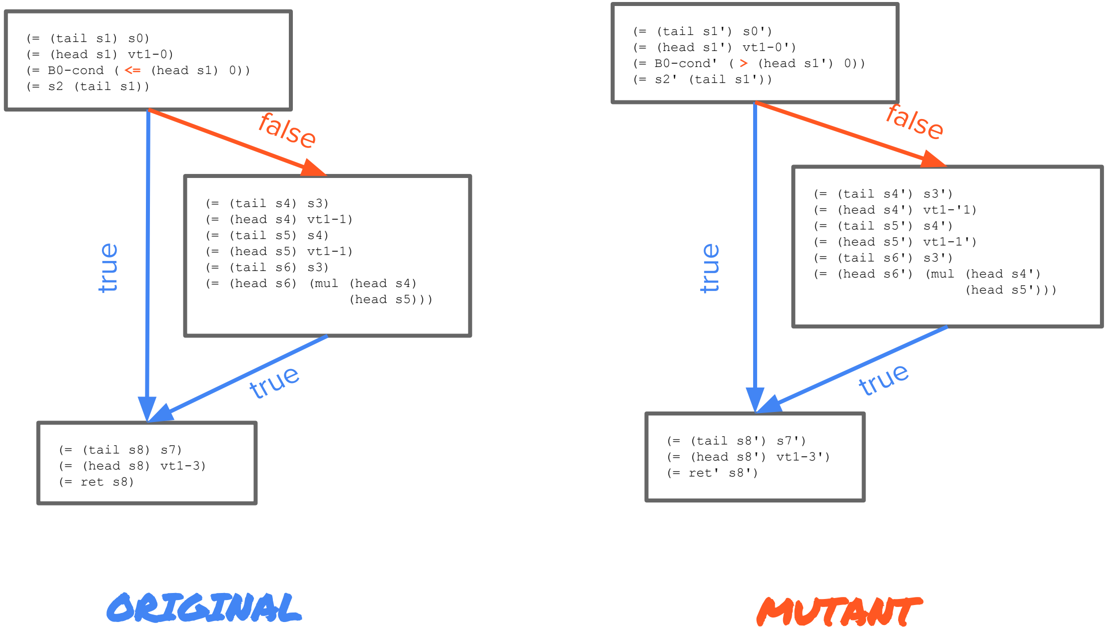
1. Compile to bytecode
2. **Construct CFG**



High Level Overview

Given a program and a mutant:

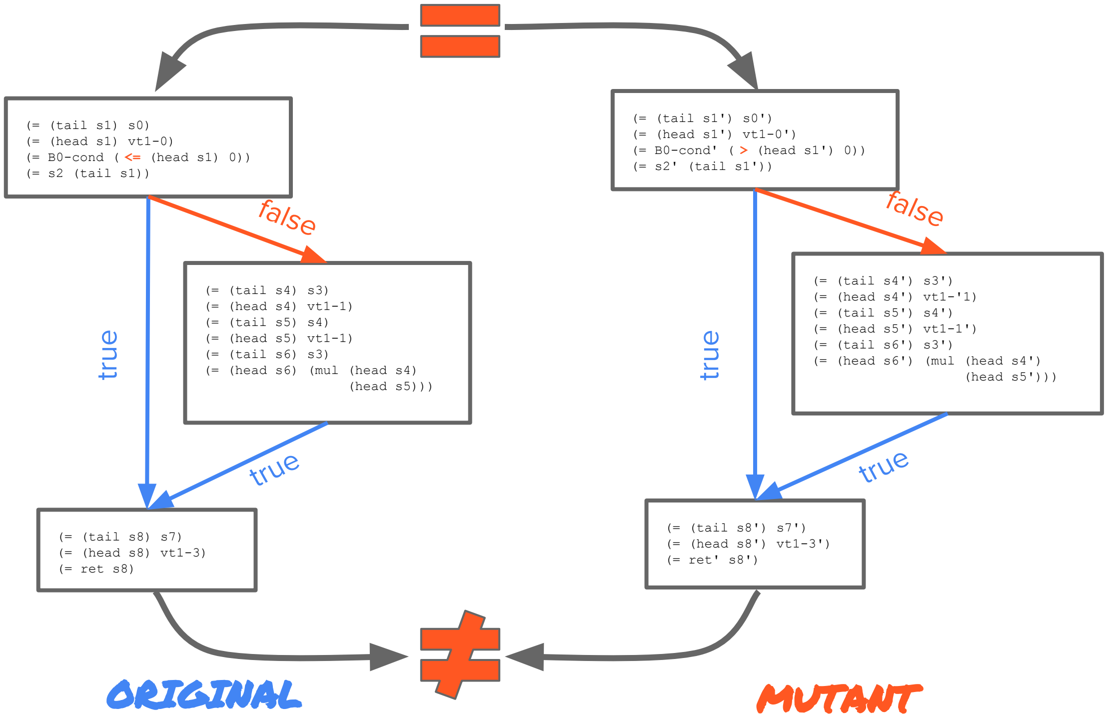
1. Compile to bytecode
2. Construct CFG
3. **Generate constraints**



High Level Overview

Given a program and a mutant:

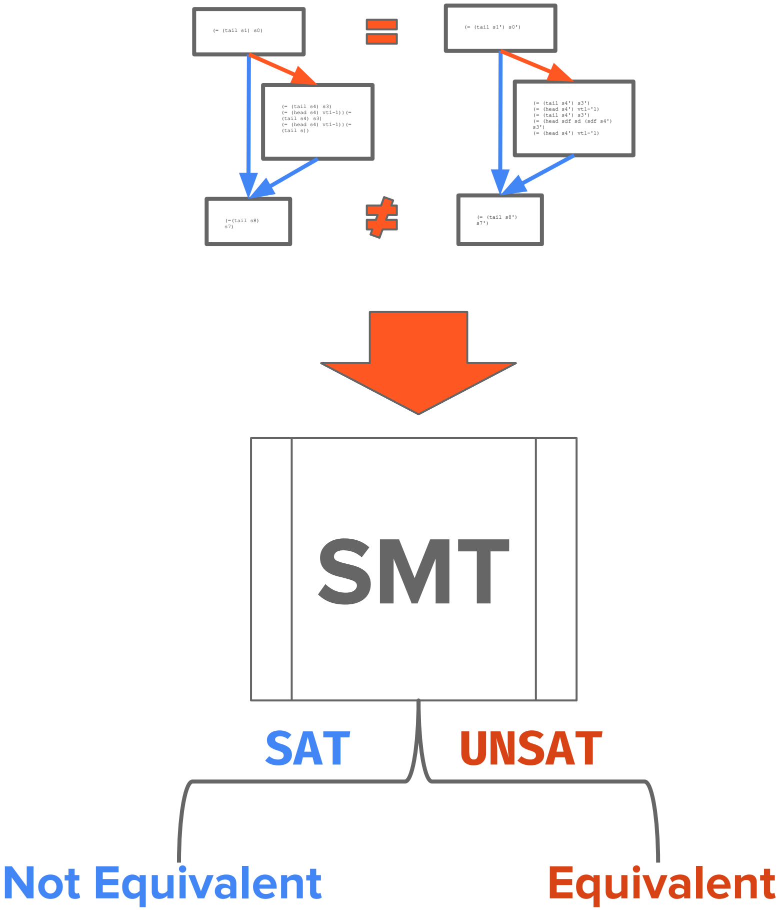
1. Compile to bytecode
2. Construct CFG
3. Generate constraints
4. **Equivalence Query**
    - **Assert inputs are equal**
    - **Assert outputs aren't equal**



High Level Overview

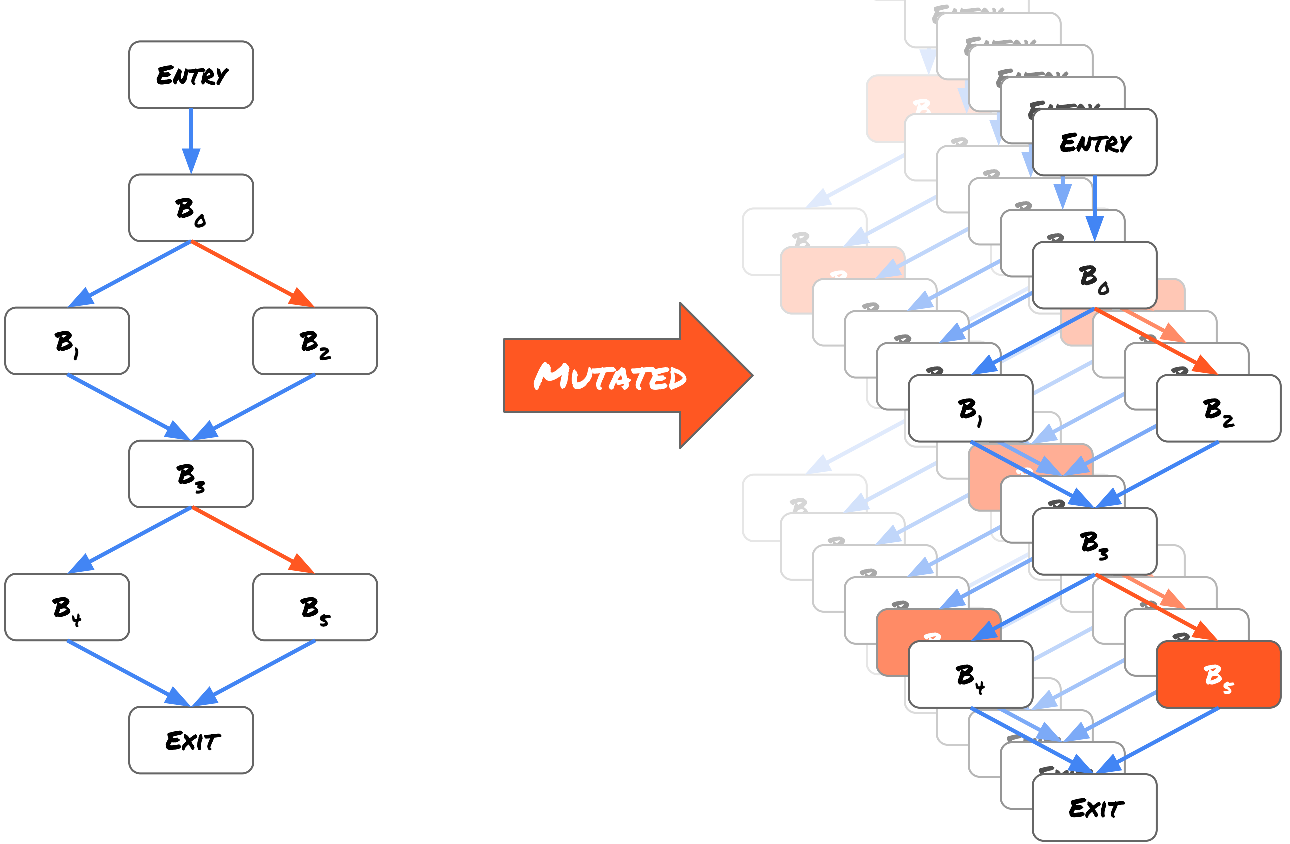
Given a program and a mutant:

1. Compile to bytecode
2. Construct CFG
3. Generate constraints
4. Equivalence Query
    - Assert inputs are equal
    - Assert outputs aren't equal
5. **Ask SMT if equivalence query is satisfiable**

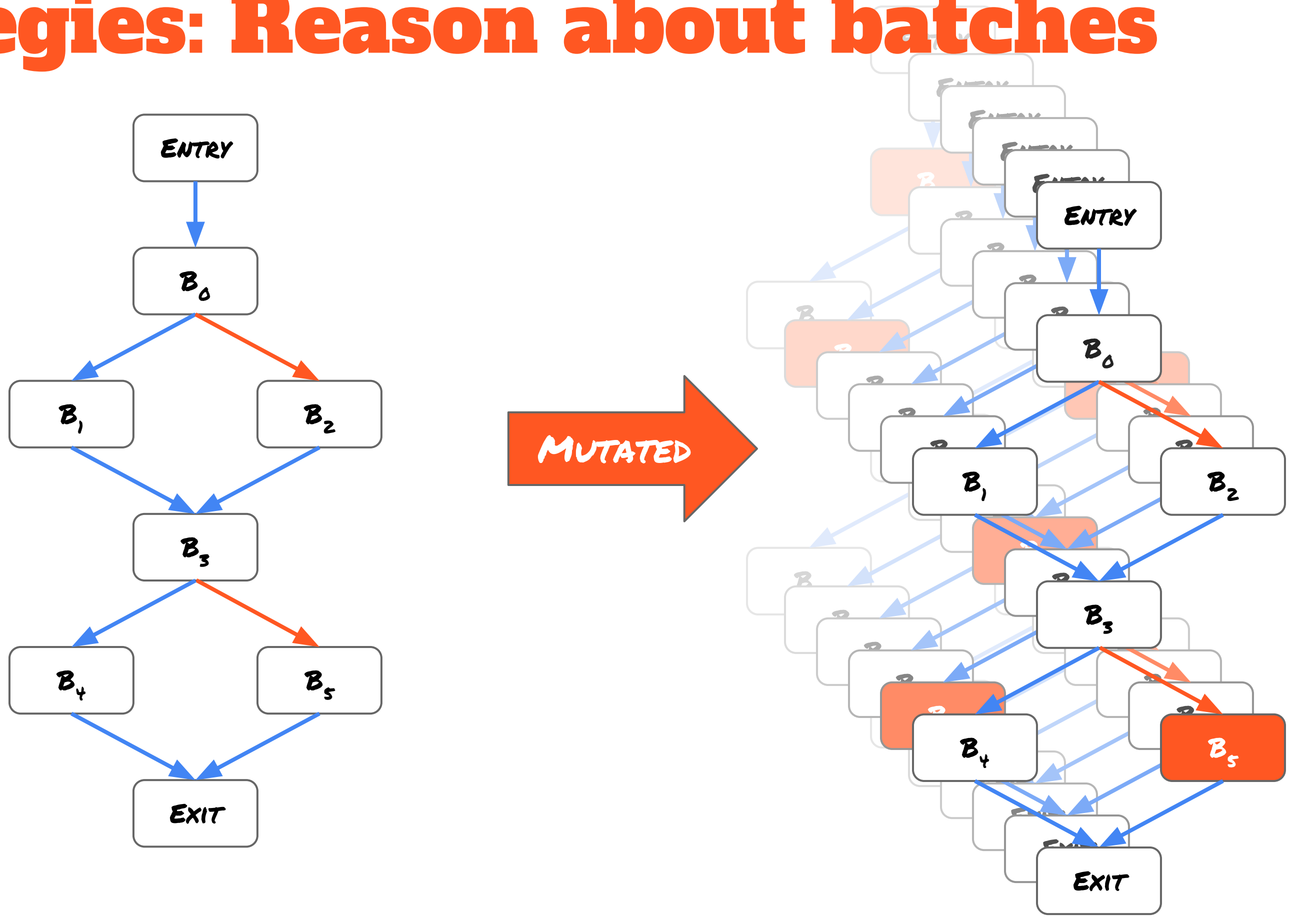


Scaling Up

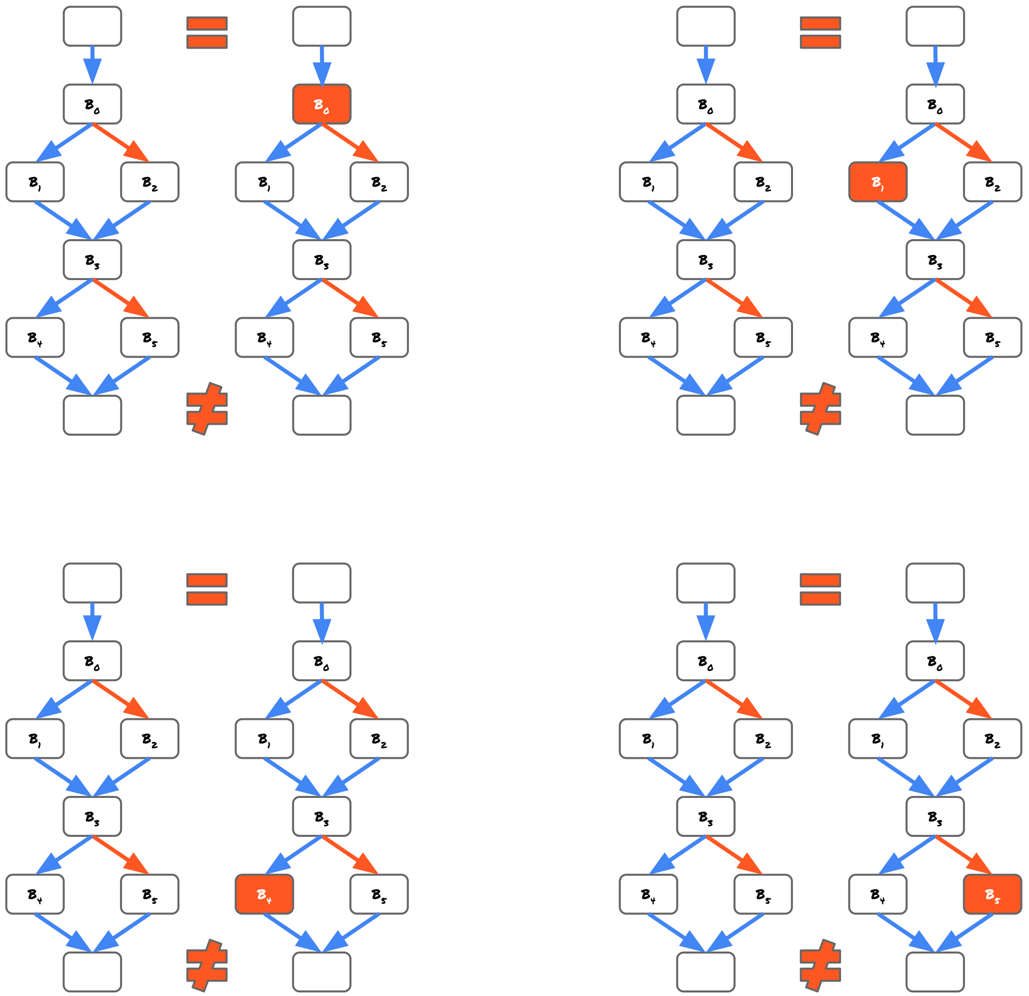
This Can Handle A Single Mutant
How Can We Handle Multiple Mutants?



Strategies: Reason about batches



Naive Strategy



Naive Strategy

```
for each mutant m of program P:
    constrain P
    constrain m
    assert inputs are equal
    assert outputs are not equal
    check sat
```

Naive Strategy

Observation: naive recomputes *P*'s constraints every time

Naive Strategy

Scopes to the rescue!

```
for each mutant m of program P:
    constrain P
    constrain m
    assert inputs are equal
    assert outputs are not equal
    check sat
```

SMT Scopes

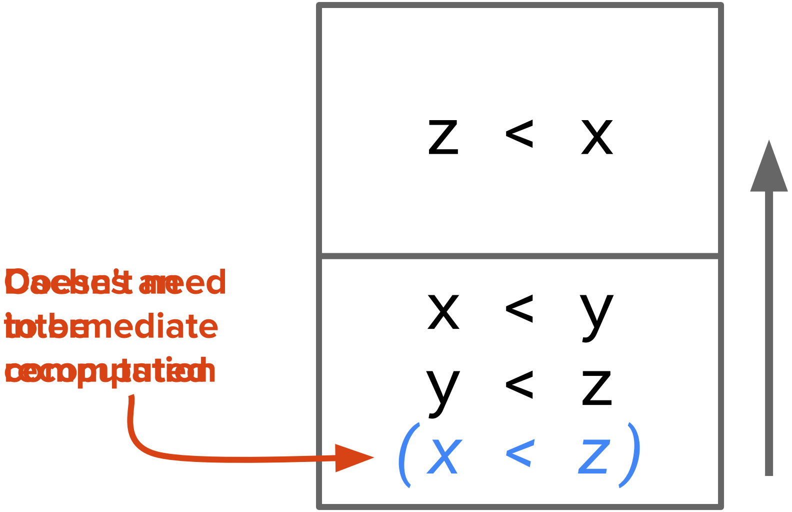
Scopes allow SMT to cache computations

- Pushed and popped to a stack

```
(assert (< x y))
(assert (< y z))
(check-sat)          ; SAT

(push)               ; New scope
(assert (< z x))
(check-sat)          ; UNSAT

(pop)
(check-sat)          ; SAT
```



scope stack

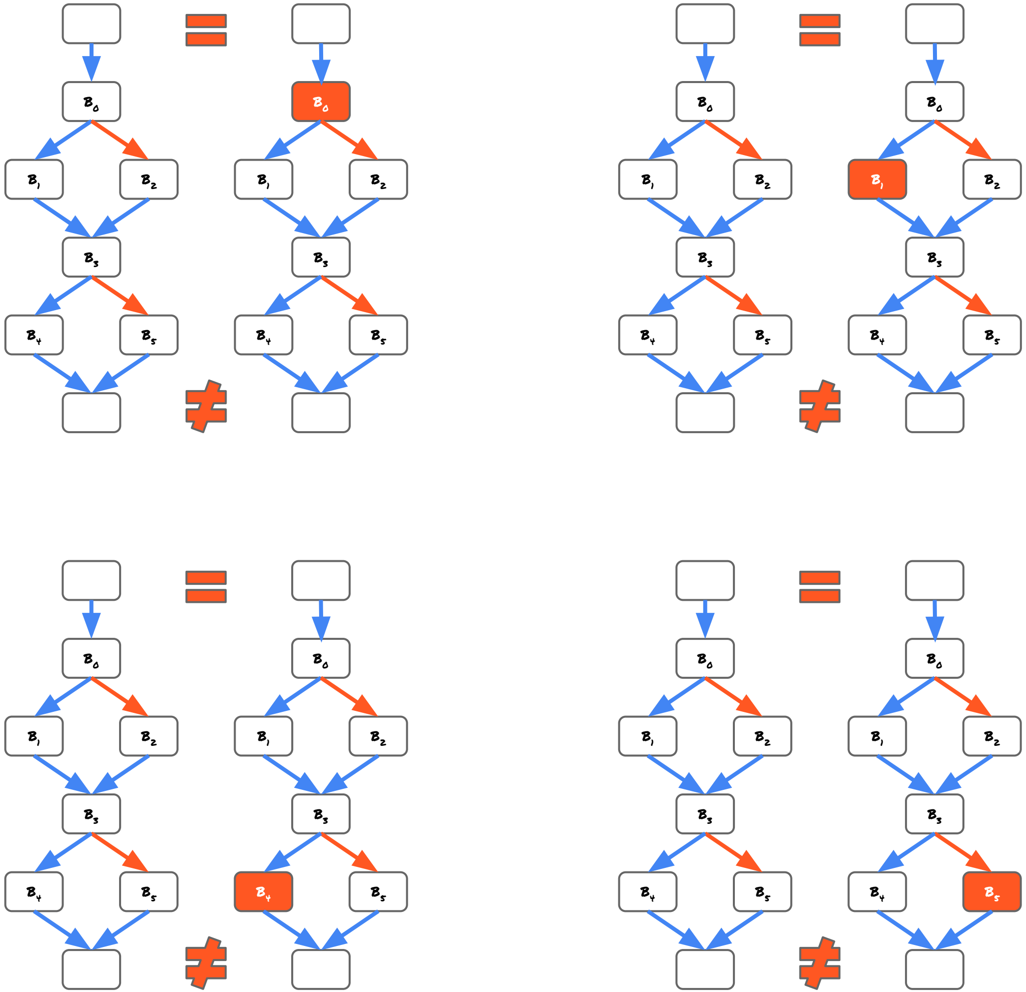
Naive Strategy

Observation: naive recomputes *P*'s constraints every iteration

Solution: use scopes!

Naive Strategy

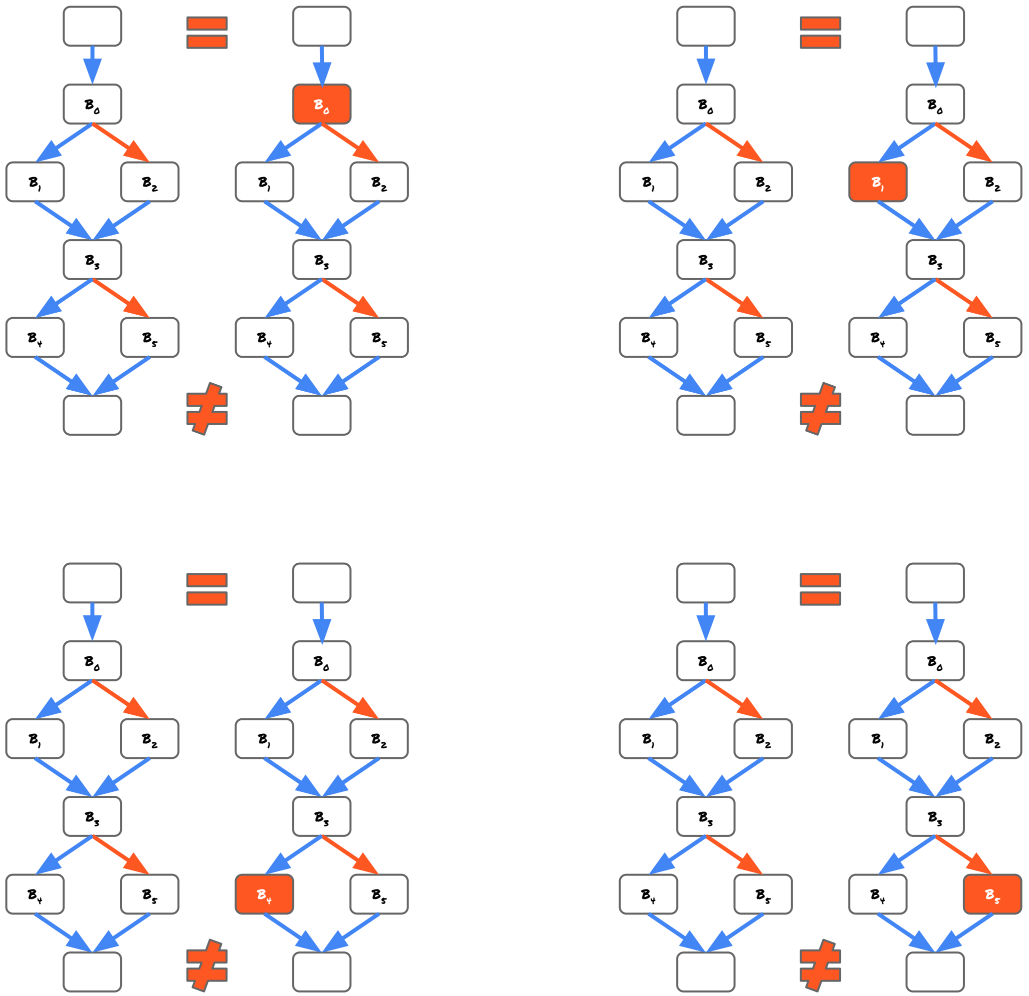
```
for each mutant m of program P:
    constrain P
    constrain m
    assert inputs are equal
    assert outputs are not equal
    check sat
```



Cache Strategy

Cache Strategy

```
constrain P
for each mutant m of program P:
    push scope
    constrain m
    assert inputs are equal
    assert outputs are not equal
    check sat
    pop scope
```

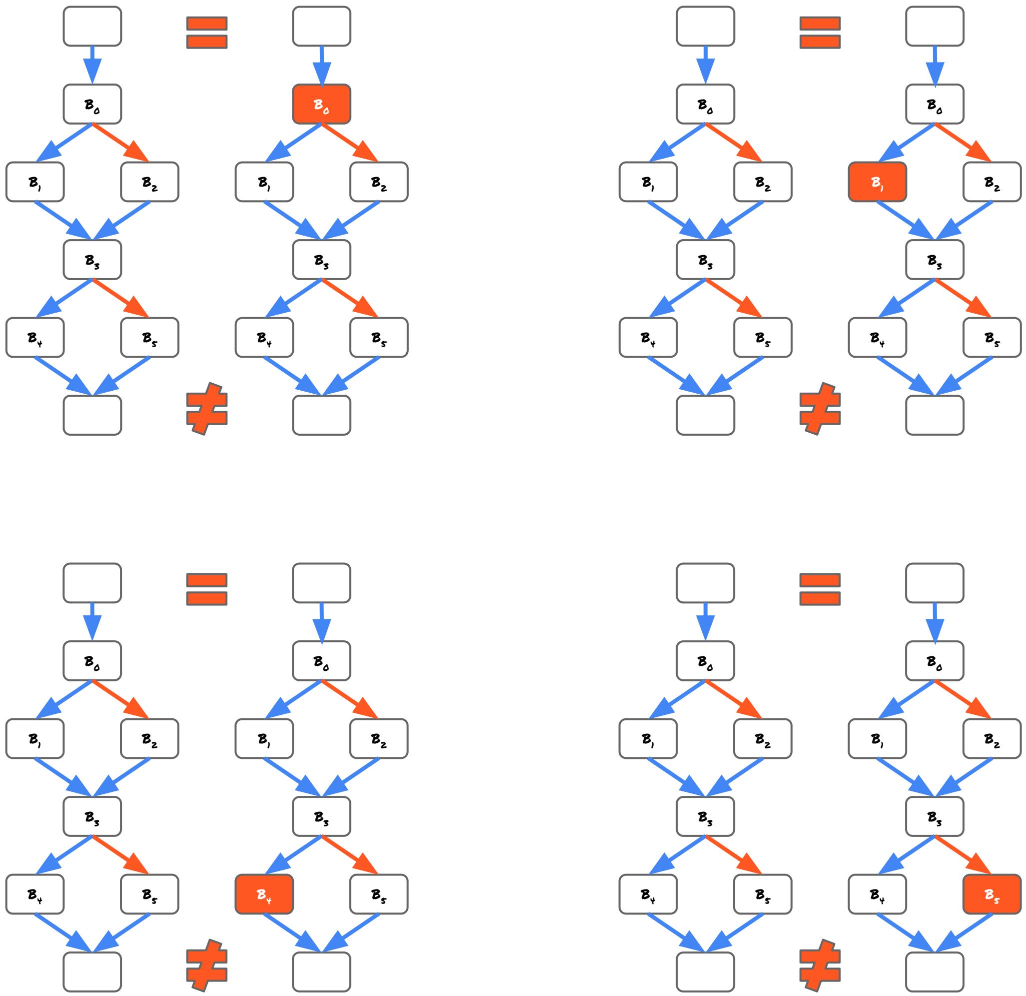


Cache Strategy

Observation: mutant's basic blocks are reconstrained each iteration

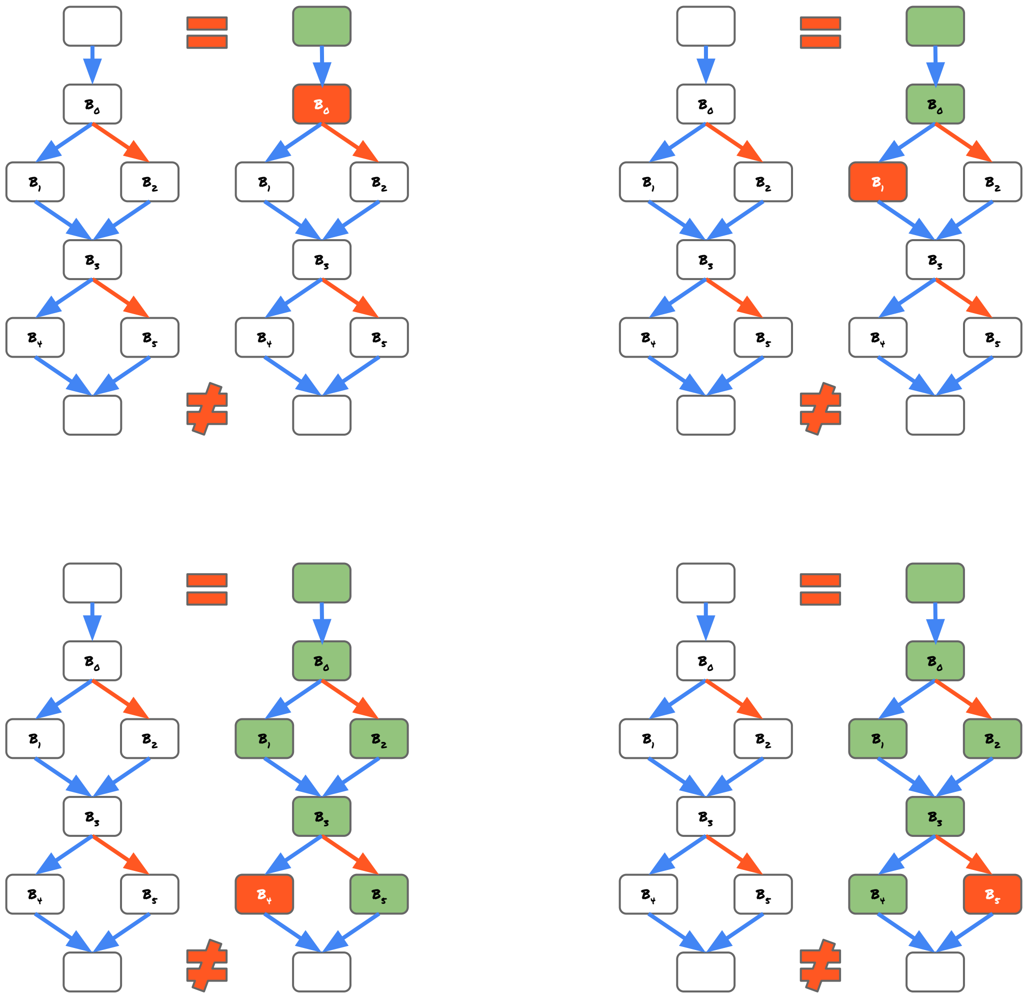
Cache Strategy

```
constrain P
for each mutant m of program P:
    push scope
    constrain m
    assert inputs are equal
    assert outputs are not equal
    check sat
    pop scope
```



Cache Strategy

Observation: mutant's basic blocks are reconstrained each iteration



Cache Strategy

```
constrain P
for each mutant m of program P:
    push scope
    constrain m
    assert inputs are equal
    assert outputs are not equal
    check sat
    pop scope
```

Atomic CFG Mutants

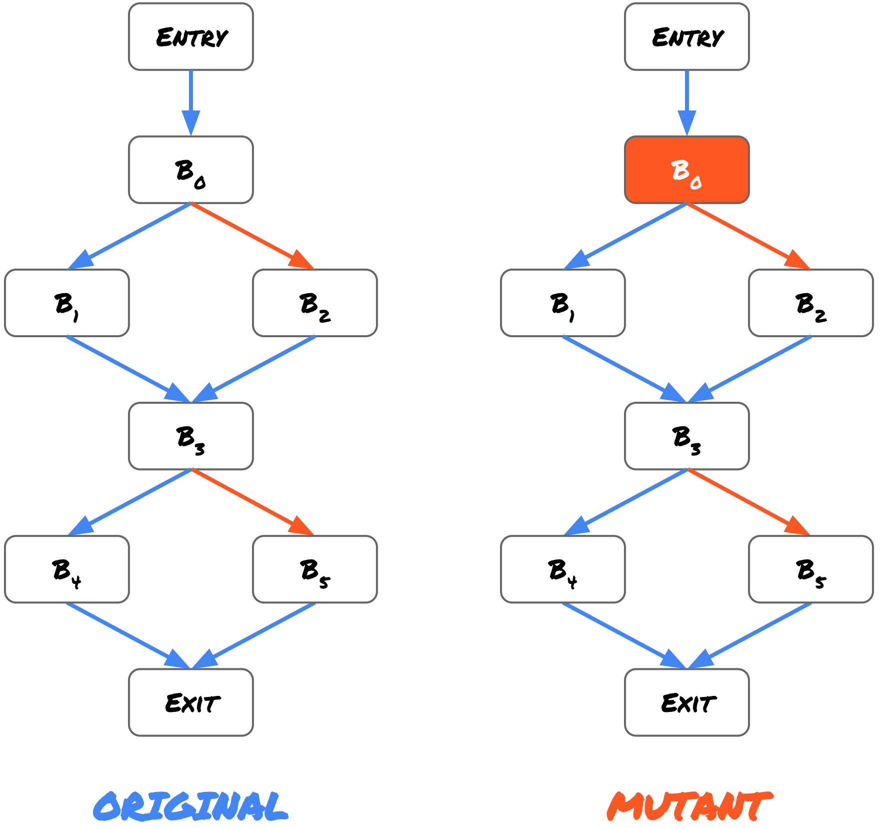
A Special Case

Atomic CFG Mutants are mutants that

1. have the same CFG structure as the original program
2. mutate at most one basic block

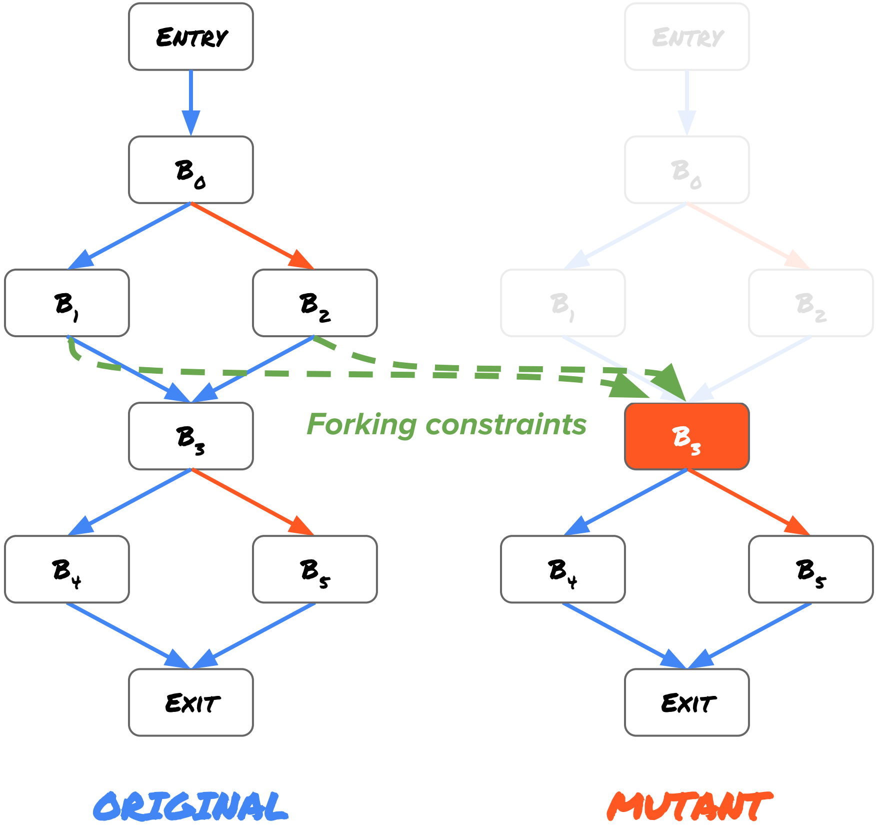
Medusa uses this structural information to perform **constraint forking**

252 of 488 (52%) of studied mutants were atomic CFG mutants



Fork Strategy

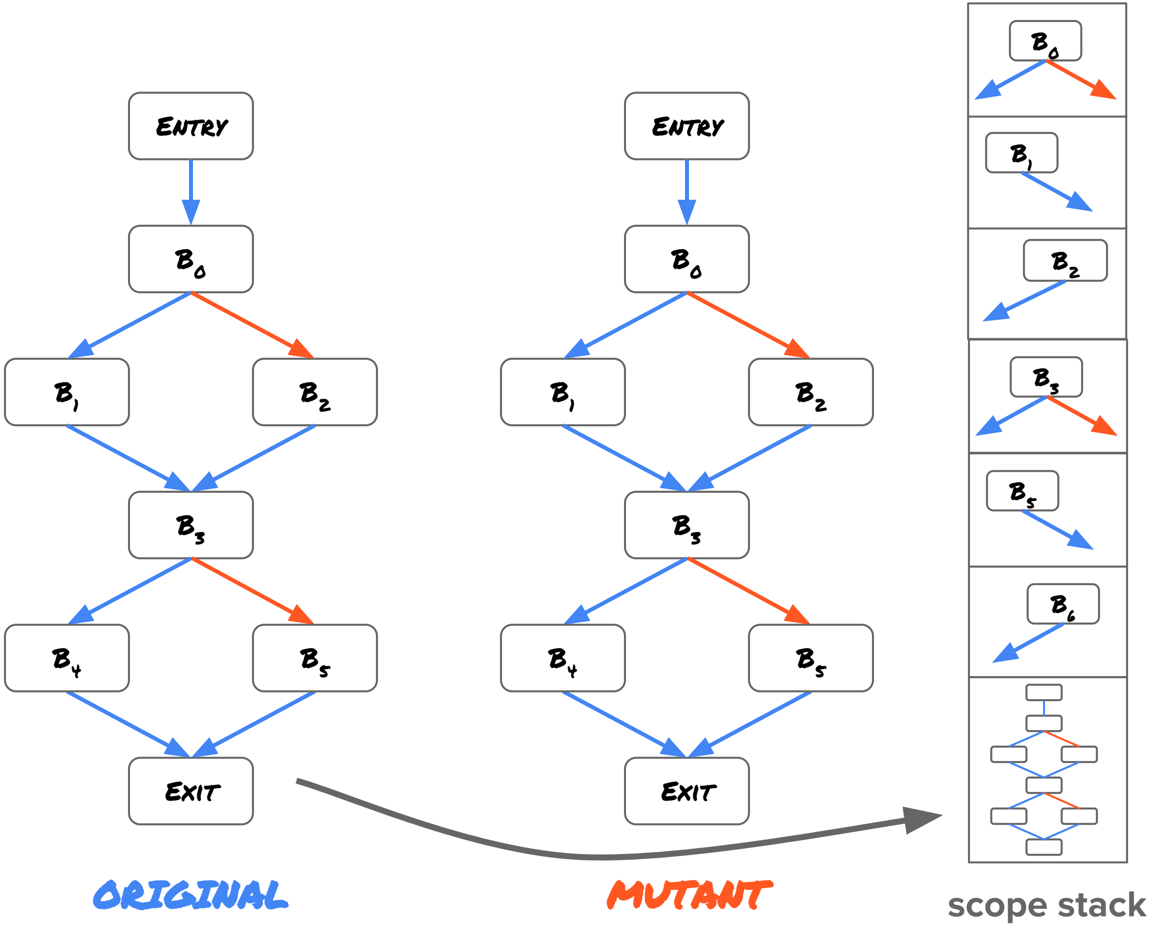
- Execution starts the same
- Use original program's constraints until mutated block is reached
- This
  - Reuses P's constraints for the beginning of execution
  - Makes formulas smaller



Fork Strategy

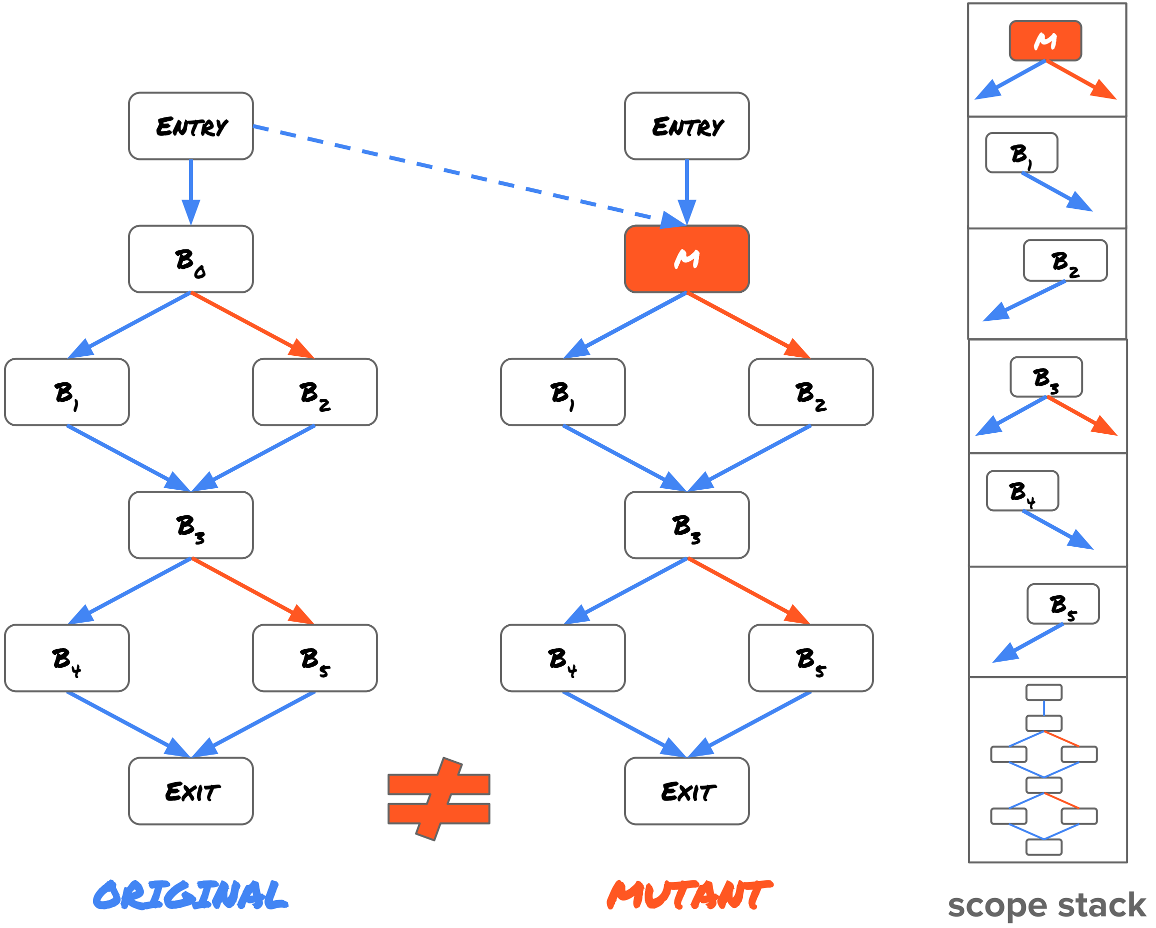
Setup

1. Order basic blocks B_0, B_1, …, B_n: each block comes before its children
2. Make two copies of the original constraints
3. Push a scope, and assert all of the first copy of the constraints
4. Bottom up, assert each block constraint and its outgoing edges into new scopes



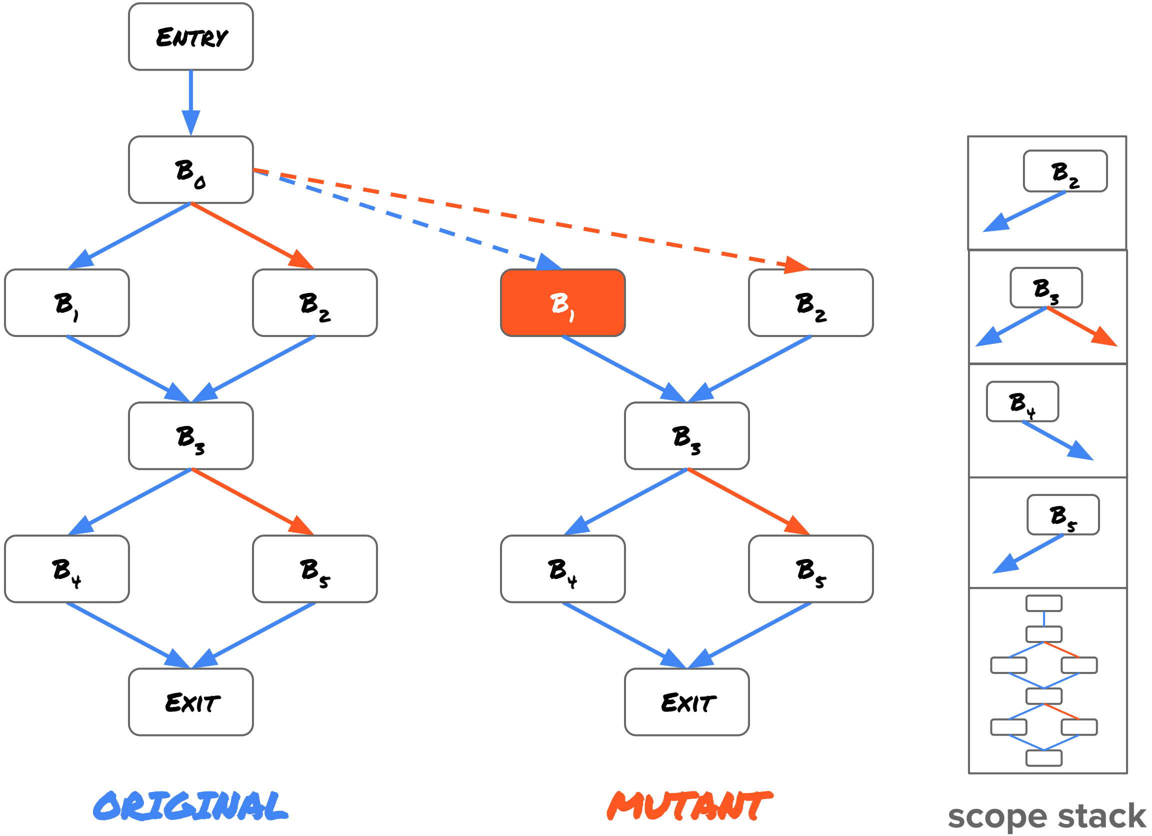
Fork Strategy

- Pop top scope,, clearing B_0's constraints.
- For each mutated version M of B_0:
  - Push a new scope; assert M's block/edge constraints
  - Fork constraints
  - Check satisfiability
  - Pop the scope



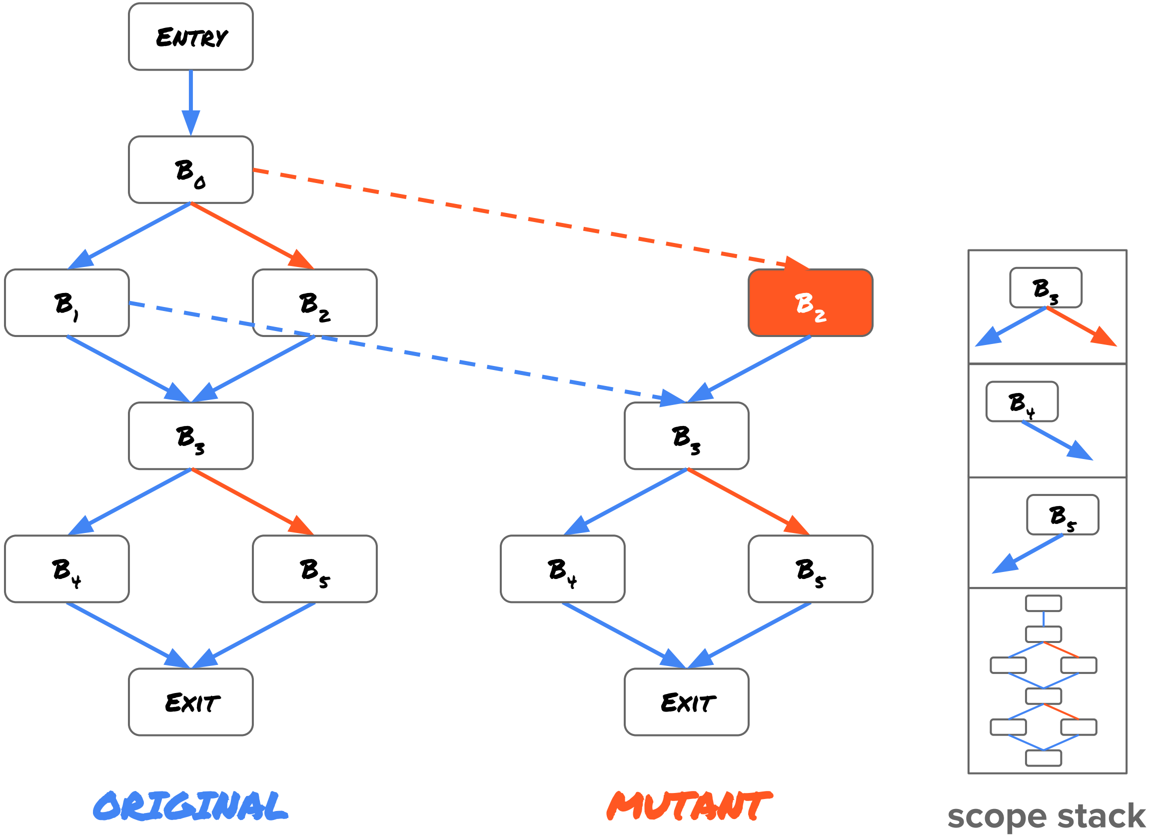
Fork Strategy

Continue until all the blocks have been processed...



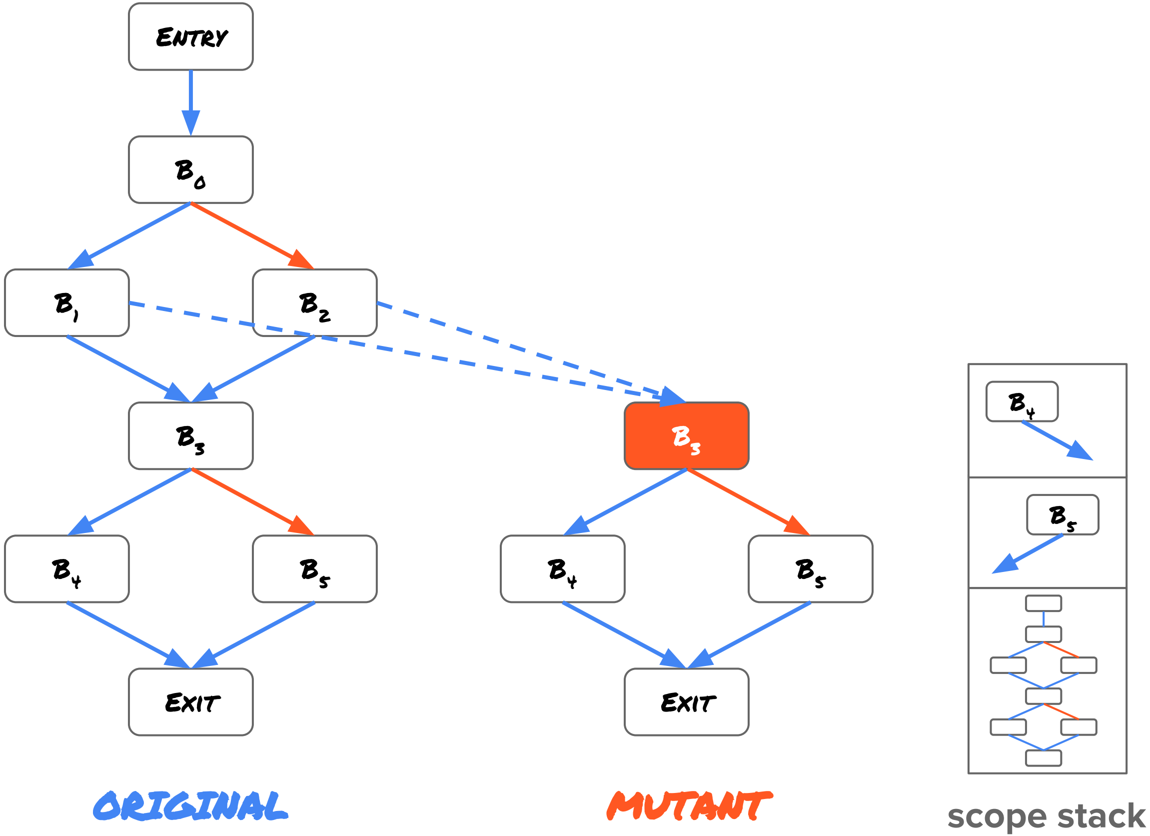
Fork Strategy

Continue until all the blocks have been processed…



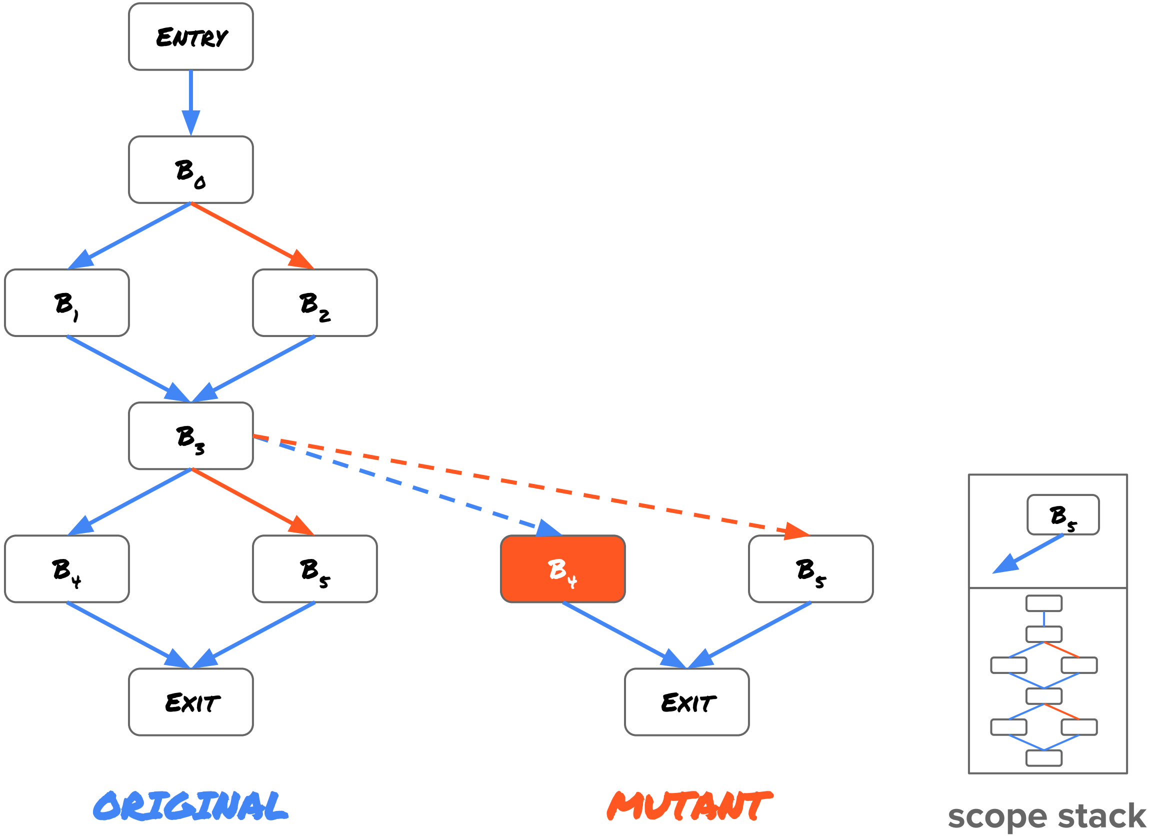
Fork Strategy

Continue until all the blocks have been processed...



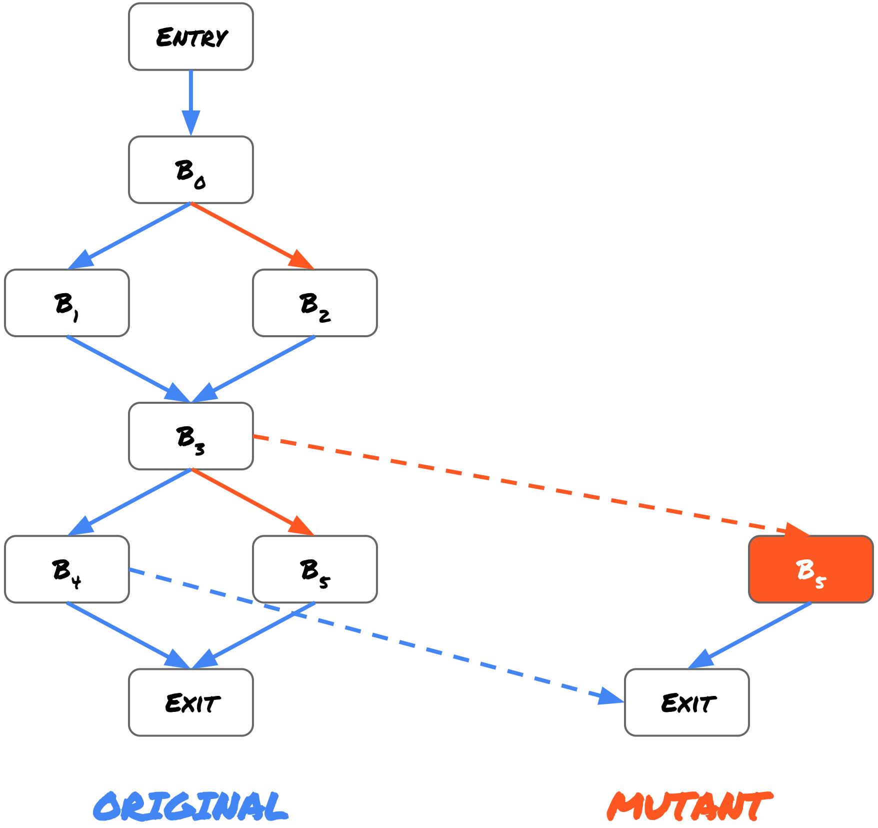
Fork Strategy

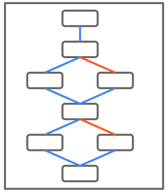
Continue until all the blocks have been processed...



Fork Strategy

Continue until all the blocks have been processed...

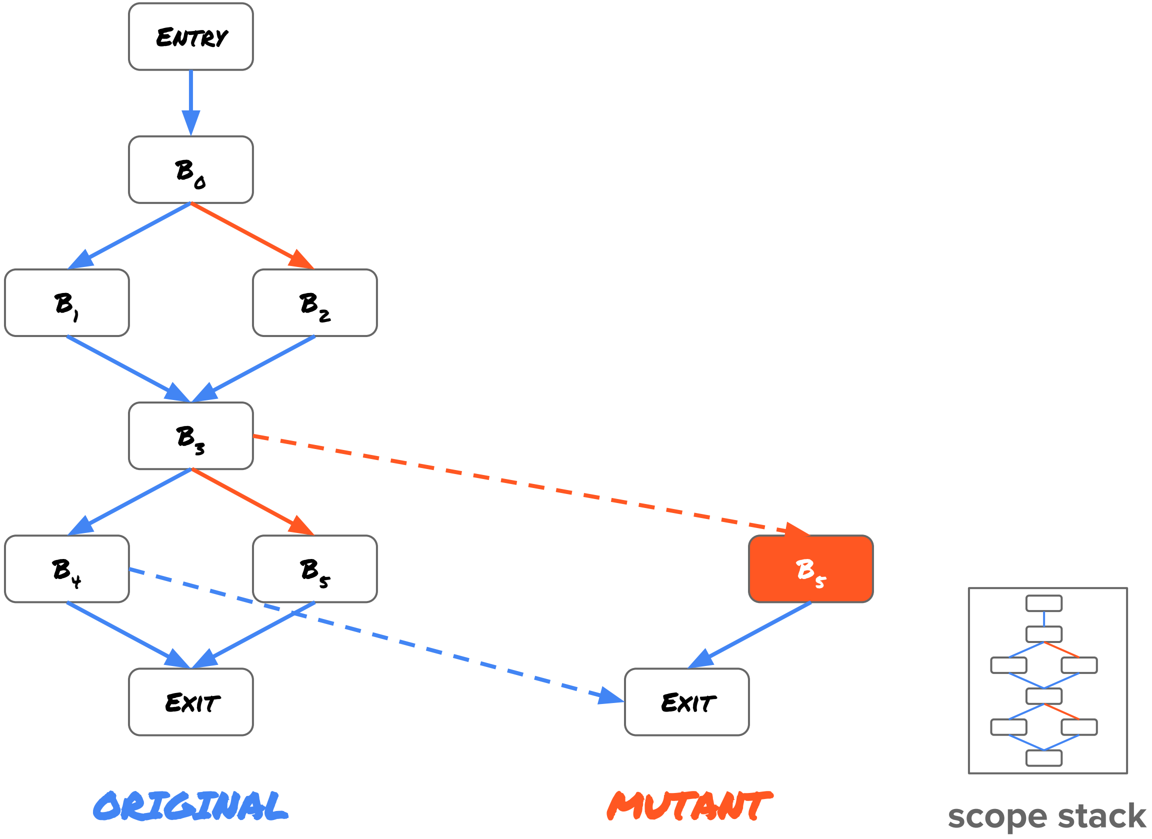




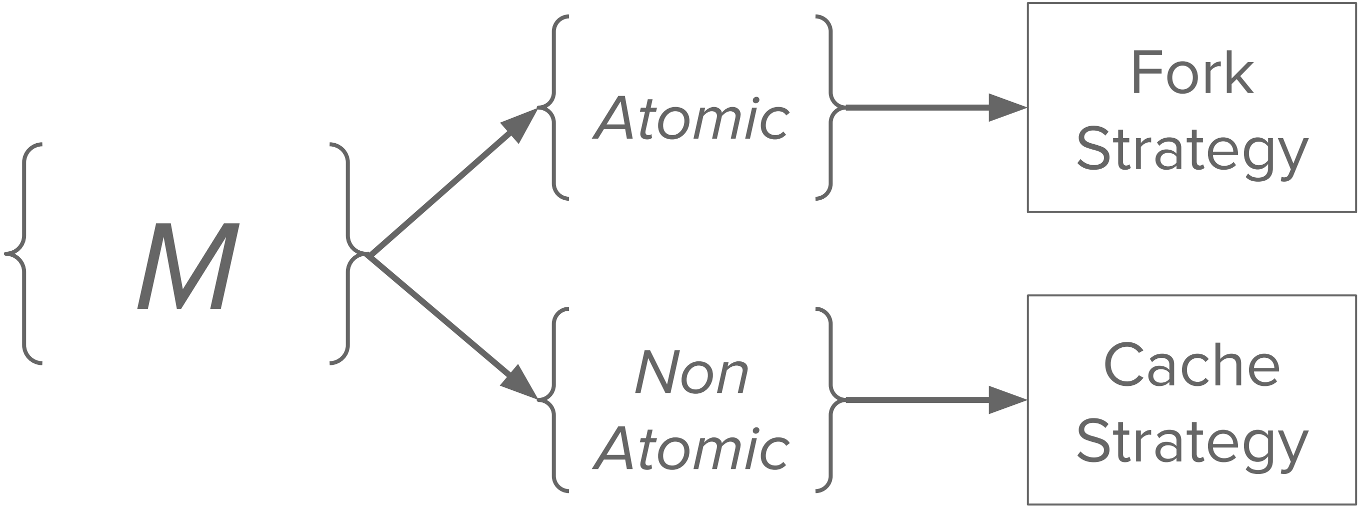
scope stack

Fork Strategy

- Fork Strategy is only applicable to atomic CFG mutants
- The strategy can be generalized (future work)
- Medusa currently uses a **hybrid strategy**



Hybrid Strategy



Preliminary Results

Three Test Subjects

1. `Tax` — computes single-payer tax amount for a given income.
2. `TicTacToe` — checks the win condition for the game of Tic Tac Toe, including bounds checking on inputs.
3. `Triangle` — classifies a triangle as equilateral, isosceles, scalene, or invalid

Mutants Generated by MAJOR

MAJOR generated **488 mutants**

- Ground truth calculated by hand
- 37 equivalent (7.6%)
- Medusa correctly classified all mutants
- 252 atomic (52%)
    - Atomic mutants are common

Subjects		All	Atomic
Tax	total	99	84
	equiv	10	10
TicTacToe	total	267	98
	equiv	23	19
Triangle	total	122	70
	equiv	4	3
All	total	488	252
	equiv	37	32

Mutants Generated by MAJOR

MAJOR generated **488 mutants**

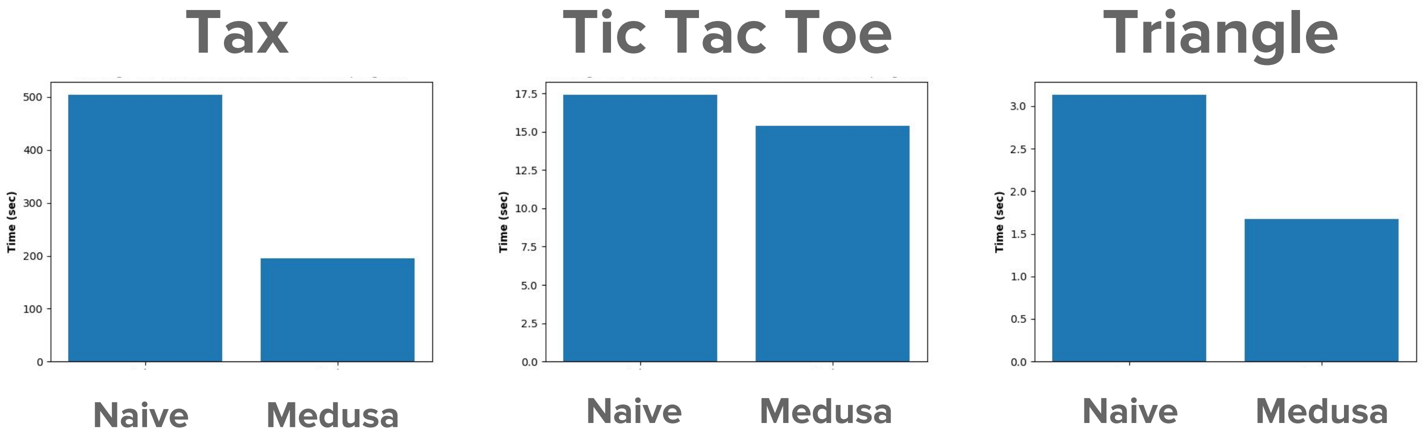
- Ground truth calculated by hand
- 37 equivalent (7.6%)
- Medusa correctly classified all mutants
- 252 atomic (52%)
  - Atomic mutants are common

TicTacToe has lower ratio of atomic mutants (large number of short circuiting boolean operators from bounds checking)

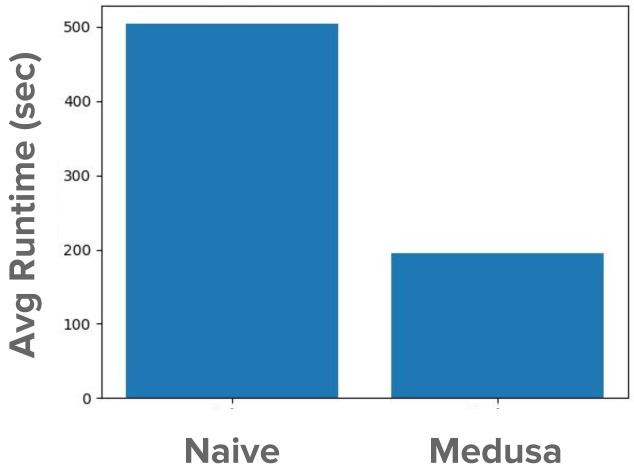
Subjects		All	Atomic
Tax	total	99	84
	equiv	10	10
TicTacToe	total	267	98
	equiv	23	19
Triangle	total	122	70
	equiv	4	3
All	total	488	252
	equiv	37	32

Summary of Medusa's performance

- **Medusa** was run 5 times on each subject and mutant (run times are averaged)
- **Tax** was long-running due to floating point computations.



Runtime results: Tax



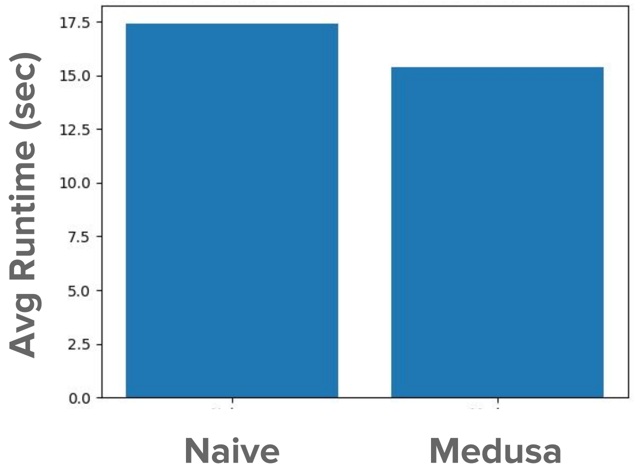
	#	Naive	Cache (Improve)	Fork (Improve)
Atomic	84	521	258 (50.5%)	197 (62.2%)
Non Atomic	15	410	185 (54.9%)	N/A
All	99	504	247 (51.0%)	195 (61.3%)

HYBRID

Results

- **Atomic** mutants tend to be **longer running than non atomic** mutants
- **Cache** has a **51% improvement** over naive
- **Hybrid** has a **61% improvement** over naive

Runtime results: Tic Tac Toe



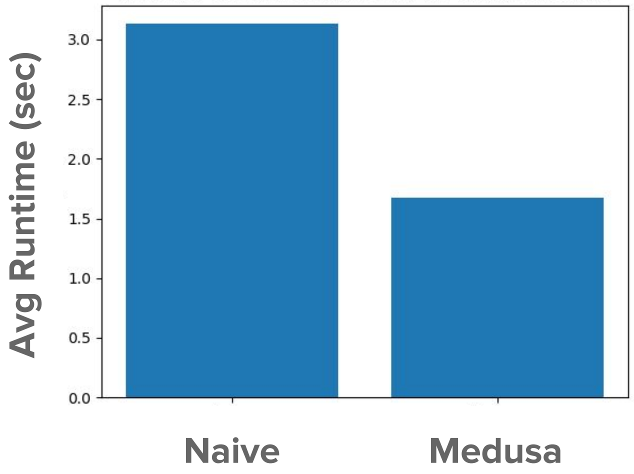
	#	Naive	Cache (Improve)	Fork (Improve)
Atomic	98	17.9	16.5 **(7.8%)**	13.9 **(22.5%)**
Non Atomic	169	17.0	16.3 **(4.1%)**	N/A
All	267	17.4	16.4 **(6.3%)**	15.4 **(11.5%)**

HYBRID

Results

- **Atomic** mutants tend to be **longer running than non atomic** mutants
- **Cache** has a **6% improvement** over naive
- **Hybrid** has a **12% improvement** over naive

Runtime results: Triangle



	#	Naive	Cache (Improve)	Fork (Improve)
Atomic	70	3.72	3.26 **(12.4%)**	1.3 **(65.1%)**
Non Atomic	52	2.34	2.16 **(7.7%)**	N/A
All	122	3.13	2.79 **(10.9%)**	1.67 **(46.6%)**

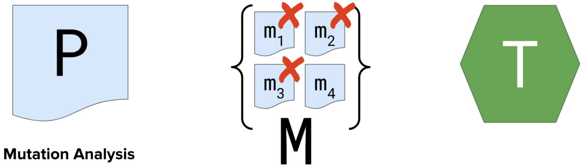
HYBRID

Results

- **Atomic** mutants tend to be **longer running than non atomic** mutants
- **Cache** has a **11% improvement** over naive
- **Hybrid** has a **47% improvement** over naive

Medusa: Mutant Equivalence Detection Using SAT Analysis

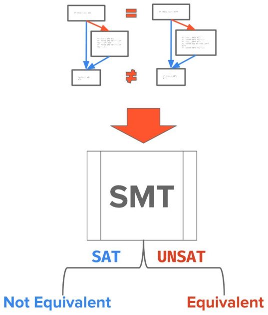
Let's analyze T's effectiveness



Mutation Analysis

- Generate mutant set *M*
- Run *T* on mutants in *M*
- Each mutant that fails a test in *T* is *killed*

High Level Overview

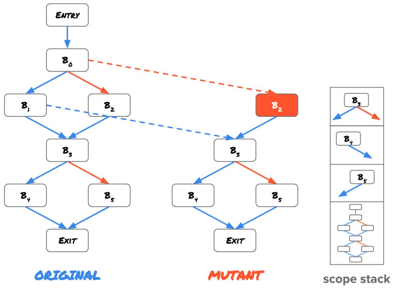


Given a program and a mutant:

1. Compile to bytecode
2. Construct CFG
3. Generate constraints
4. Equivalence Query
   - Assert inputs are equal
   - Assert outputs aren't equal
5. **Ask SMT if equivalence query is satisfiable**

Fork Strategy

Continue until all the blocks have been processed...



Summary of Medusa's performance

- **Medusa** was run 5 times on each subject and mutant (run times are averaged)
- **Tax** was long-running due to floating point computations.