TODAY:

(CONCRETE/SYMBOLIC) CONCOLIC EXECUTION

TU Delft

Program

- Theory:
  - *satisfiability and solvers*
- Practice:
  - *from code to formulas*
  - *following program execution*
- Binaries:
  - *from instrumentation and tainting*
  - *to concolic execution*

Satisfiability & Validity

$p \lor q \Rightarrow q \lor p$

$p \lor q \Rightarrow q$

$p \land \neg q \ \land (\neg p \lor q)$

ϕ	A	B	$\neg A$	$A \lor B$	$A \land \neg A$	$A \Rightarrow B$	$A \Rightarrow (B \lor A)$
$\mathcal{M}_1(\phi)$	$\bot$	$\bot$	$\top$	$\bot$	$\bot$	$\top$	$\top$
$\mathcal{M}_2(\phi)$	$\bot$	$\top$	$\top$	$\top$	$\bot$	$\top$	$\top$
$\mathcal{M}_3(\phi)$	$\top$	$\bot$	$\bot$	$\top$	$\bot$	$\bot$	$\top$
$\mathcal{M}_4(\phi)$	$\top$	$\top$	$\bot$	$\top$	$\bot$	$\top$	$\top$

slides from Leonardo de Moura

Satisfiability & Validity

$p \vee q \Rightarrow q \vee p$ VALID

$p \vee q \Rightarrow q$ SATISFIABLE

$p \wedge \neg q \wedge (\neg p \vee q)$ UNSATISFIABLE

ϕ	A	B	$\neg A$	$A \vee B$	$A \wedge \neg A$	$A \Rightarrow B$	$A \Rightarrow (B \vee A)$
$\mathcal{M}_1(\phi)$	$\bot$	$\bot$	$\top$	$\bot$	$\bot$	$\top$	$\top$
$\mathcal{M}_2(\phi)$	$\bot$	$\top$	$\top$	$\top$	$\bot$	$\top$	$\top$
$\mathcal{M}_3(\phi)$	$\top$	$\bot$	$\bot$	$\top$	$\bot$	$\bot$	$\top$
$\mathcal{M}_4(\phi)$	$\top$	$\top$	$\bot$	$\top$	$\bot$	$\top$	$\top$

SAT solvers

- Decades of experience in solving NP-hard problems by translating them to Satisfiability
  - *problem has solution iff formula is satisfiable*
- Yearly competitions pushing the state-of-the-art
  - highly optimized
  - competitive for many problems
  - very general: *every NP-hard problem can be translated to SAT*
  - *(Interested in MSc project? Let me know!)*
- Recently used to solve long-standing open problems:
  - http://www.nature.com/news/two-hundred-terabyte-maths-proof-is-largest-ever-1.19990
- Simply very fast at solving formulas in propositional logic

TU Delft

A CNF formula

$$
\begin{aligned}
\varphi \quad = \quad & (a \vee \neg b \vee d) \wedge (a \vee \neg b \vee e) \wedge \\
& (\neg b \vee \neg d \vee \neg e) \wedge \\
& (a \vee b \vee c \vee d) \wedge (a \vee b \vee c \vee \neg d) \wedge \\
& (a \vee b \vee \neg c \vee e) \wedge (a \vee b \vee \neg c \vee \neg e)
\end{aligned}
$$

A CNF formula

$$
\begin{aligned}
\varphi \quad = \quad & (a \vee \neg b \vee d) \wedge (a \vee \neg b \vee e) \wedge \\
& (\neg b \vee \neg d \vee \neg e) \wedge \\
& (a \vee b \vee c \vee d) \wedge (a \vee b \vee c \vee \neg d) \wedge \\
& (a \vee b \vee \neg c \vee e) \wedge (a \vee b \vee \neg c \vee \neg e)
\end{aligned}
$$

Satisfiable?

SAT solver inner workings --- DPLL

- Standard backtrack search
- DPLL(F) :
  - Apply unit propagation
  - If conflict identified, return UNSAT
  - Apply the pure literal rule
  - If F is satisfied (empty), return SAT
  - Select decision variable x
    - If DPLL($F \wedge x$)=SAT return SAT
    - return DPLL($F \wedge \neg x$)

Simple but effective: Unit Propagation

(Davis–Putnam–Logemann–Loveland)

DPLL $=$ Unit resolution $+$ Split rule.

$$\frac{\Gamma}{\Gamma, p \mid \Gamma, \neg p}\,split \quad p \text{ and } \neg p \text{ are not in } \Gamma.$$

$$\frac{C \vee \bar{l}, l}{C, l}\,unit$$

Used in the most efficient SAT solvers.

Simple but effective: Pure Literals

A literal is pure if only occurs positively or negatively.

Example :

$\varphi = (\neg x_1 \vee x_2) \wedge (x_3 \vee \neg x_2) \wedge (x_4 \vee \neg x_5) \wedge (x_5 \vee \neg x_4)$

$\neg x_1$ and x_3 are pure literals

Pure literal rule :

Clauses containing pure literals can be removed from the formula (i.e. just satisfy those pure literals)

$$\varphi_{\neg x_1, x_3} = (x_4 \vee \neg x_5) \wedge (x_5 \vee \neg x_4)$$

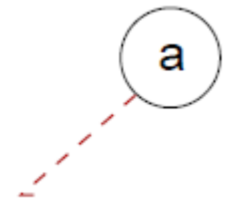
Preserve satisfiability, not logical equivalency !

TU Delft

DPLL (example)

Guess

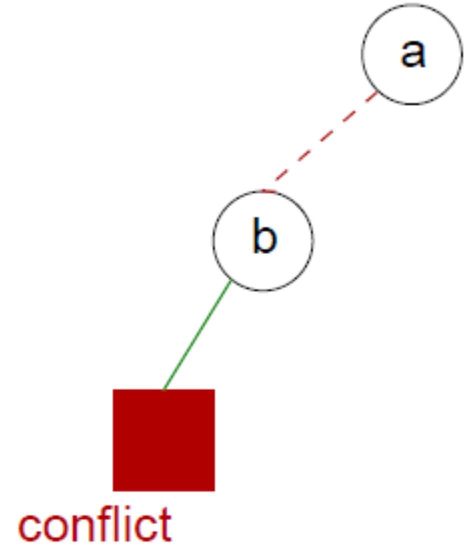
$$
\begin{aligned}
\varphi \;=\; & (a \vee \neg b \vee d) \wedge (a \vee \neg b \vee e) \wedge \\
& (\neg b \vee \neg d \vee \neg e) \wedge \\
& (a \vee b \vee c \vee d) \wedge (a \vee b \vee c \vee \neg d) \wedge \\
& (a \vee b \vee \neg c \vee e) \wedge (a \vee b \vee \neg c \vee \neg e)
\end{aligned}
$$



DPLL (example)

Deduce

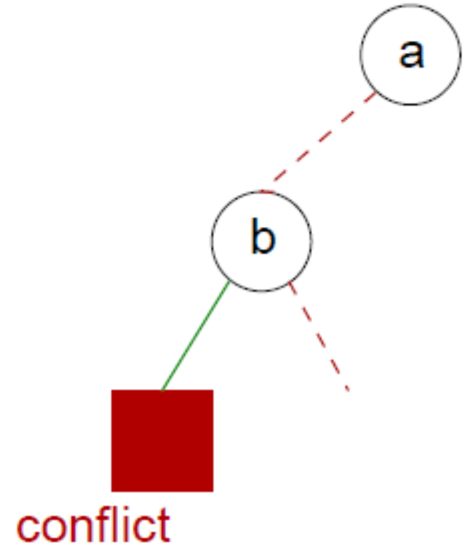
$$
\begin{aligned}
\varphi \quad = \quad & (a \vee \neg b \vee d) \wedge (a \vee \neg b \vee e) \wedge \\
& (\neg b \vee \neg d \vee \neg e) \wedge \\
& (a \vee b \vee c \vee d) \wedge (a \vee b \vee c \vee \neg d) \wedge \\
& (a \vee b \vee \neg c \vee e) \wedge (a \vee b \vee \neg c \vee \neg e)
\end{aligned}
$$



DPLL (example)

Backtrack

$$\begin{aligned}\varphi \;=\; & (a \vee \neg b \vee d) \wedge (a \vee \neg b \vee e) \wedge \\ & (\neg b \vee \neg d \vee \neg e) \wedge \\ & (a \vee b \vee c \vee d) \wedge (a \vee b \vee c \vee \neg d) \wedge \\ & (a \vee b \vee \neg c \vee e) \wedge (a \vee b \vee \neg c \vee \neg e)\end{aligned}$$

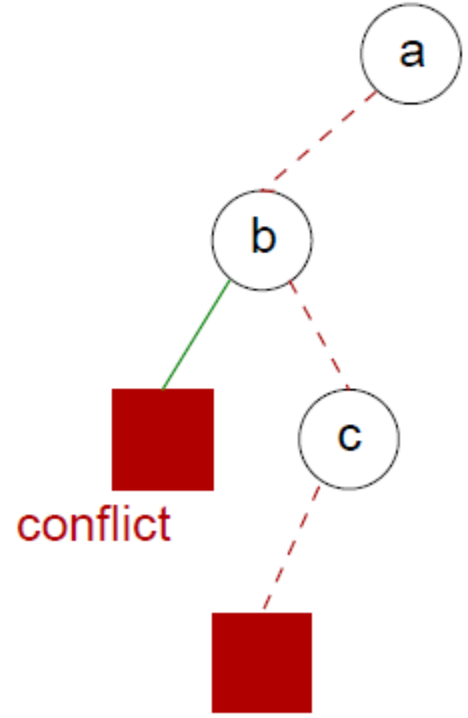


TU Delft

DPLL (example)

Deduce

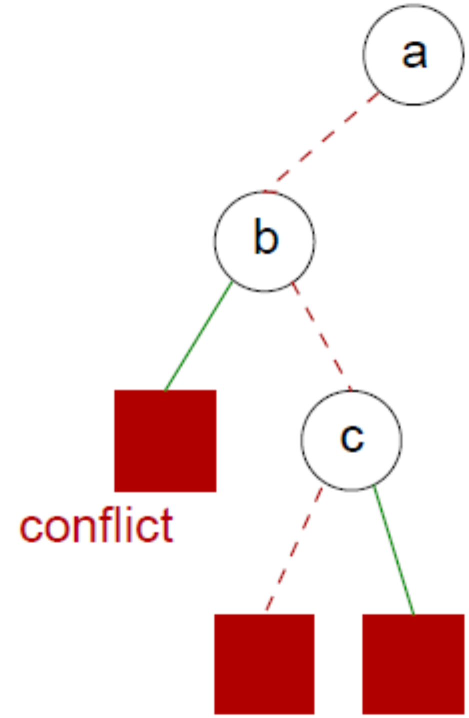
$$\begin{aligned}\varphi \quad = \quad & (a \vee \neg b \vee d) \wedge (a \vee \neg b \vee e) \wedge \\ & (\neg b \vee \neg d \vee \neg e) \wedge \\ & (a \vee b \vee c \vee d) \wedge (a \vee b \vee c \vee \neg d) \wedge \\ & (a \vee b \vee \neg c \vee e) \wedge (a \vee b \vee \neg c \vee \neg e)\end{aligned}$$



DPLL (example)

Deduce

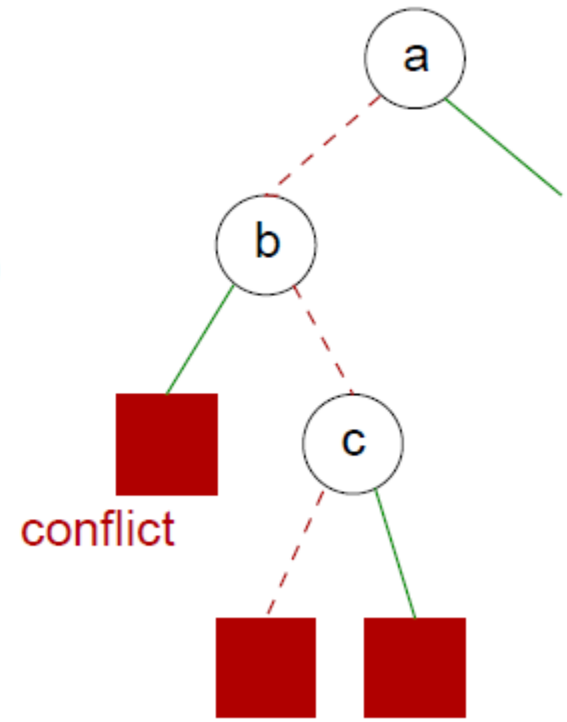
$$\varphi = (a \vee \neg b \vee d) \wedge (a \vee \neg b \vee e) \wedge$$
$$(\neg b \vee \neg d \vee \neg e) \wedge$$
$$(a \vee b \vee c \vee d) \wedge (a \vee b \vee c \vee \neg d) \wedge$$
$$(a \vee b \vee \neg c \vee e) \wedge (a \vee b \vee \neg c \vee \neg e)$$



DPLL (example)

Backtrack

$$\begin{aligned}\varphi \quad = \quad & (a \vee \neg b \vee d) \wedge (a \vee \neg b \vee e) \wedge \\ & (\neg b \vee \neg d \vee \neg e) \wedge \\ & (a \vee b \vee c \vee d) \wedge (a \vee b \vee c \vee \neg d) \wedge \\ & (a \vee b \vee \neg c \vee e) \wedge (a \vee b \vee \neg c \vee \neg e)\end{aligned}$$

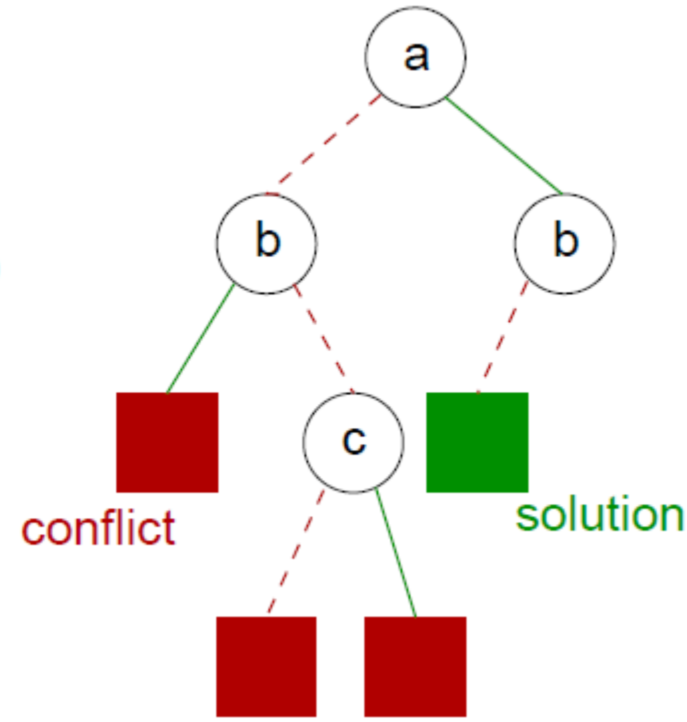


TU Delft

DPLL (example)

Guess

$$\varphi = (a \vee \neg b \vee d) \wedge (a \vee \neg b \vee e) \wedge$$
$$(\neg b \vee \neg d \vee \neg e) \wedge$$
$$(a \vee b \vee c \vee d) \wedge (a \vee b \vee c \vee \neg d) \wedge$$
$$(a \vee b \vee \neg c \vee e) \wedge (a \vee b \vee \neg c \vee \neg e)$$



TU Delft

Modern DPLL

- Do guess-deduce-backtrack very efficiently
  - all represented using bits and binary operators
- In addition
  - Efficient indexing (two-watch literal)
  - Non-chronological backtracking (backjumping)
  - Lemma learning
- Google if interested…building a SAT solver from scratch is unlikely competitive with the state-of-the-art:
  - They improve every year…
  - See SAT Live!: http://www.satlive.org/solvers/

TU Delft

SAT solvers 2

- ..
- Simply very fast at solving formulas in propositional logic
  - In contrast to GAs and local search, SAT solvers are complete!
- But not so good at modeling numeric variables:
  - integers, floats, (non-)linear arithmetic, …
- Possible using Boolean representation:
  - 0 = 000, 1 = 001 2 = 010, 3 = 011, ..
- But representing arithmetic in logic is slow...
- So use a *seperate solver* for such operations!
  - or represent the problem as an integer program...
  - or use other approaches such as genetic algorithms...

Satisfiability Modulo Theories (SMT)

Is formula *F* satisfiable modulo theory *T* ?

SMT solvers have specialized algorithms for *T*

TU Delft

SAT + Theory solvers

Basic Idea

$x \geq 0, y = x + 1, (y > 2 \vee y < 1)$

Abstract (aka "naming" atoms)

$p_1, \; p_2, (p_3 \vee p_4)$ $\quad$ $p_1 \equiv (x \geq 0), p_2 \equiv (y = x + 1),$

$p_3 \equiv (y > 2), p_4 \equiv (y < 1)$

SAT + Theory solvers

Basic Idea

$x \geq 0,\ y = x + 1,\ (y > 2 \vee y < 1)$

Abstract (aka "naming" atoms)

$p_1,\ p_2,\ (p_3 \vee p_4)$

$p_1 \equiv (x \geq 0),\ p_2 \equiv (y = x + 1),$
$p_3 \equiv (y > 2),\ p_4 \equiv (y < 1)$

SAT
Solver

TUDelft

SAT + Theory solvers

Basic Idea

$x \geq 0,\ y = x + 1,\ (y > 2 \vee y < 1)$

Abstract (aka "naming" atoms)

$p_1,\ p_2,\ (p_3 \vee p_4)$

$p_1 \equiv (x \geq 0),\ p_2 \equiv (y = x + 1),$

$p_3 \equiv (y > 2),\ p_4 \equiv (y < 1)$

SAT Solver

$p_1,\ p_2,\ \neg p_3,\ p_4$

SAT + Theory solvers

Basic Idea

$x \geq 0,\ y = x + 1,\ (y > 2 \vee y < 1)$

Abstract (aka "naming" atoms)

$p_1,\ p_2,\ (p_3 \vee p_4)$

$p_1 \equiv (x \geq 0),\ p_2 \equiv (y = x + 1),$
$p_3 \equiv (y > 2),\ p_4 \equiv (y < 1)$

SAT Solver

$p_1,\ p_2,\ \neg p_3,\ p_4$

$x \geq 0,\ y = x + 1,$
$\neg(y > 2),\ y < 1$

SAT + Theory solvers

Basic Idea

$x \geq 0,\ y = x + 1,\ (y > 2 \vee y < 1)$

Abstract (aka "naming" atoms)

$p_1,\ p_2,\ (p_3 \vee p_4)$

$p_1 \equiv (x \geq 0),\ p_2 \equiv (y = x + 1),$
$p_3 \equiv (y > 2),\ p_4 \equiv (y < 1)$

SAT Solver

Assignment
$p_1,\ p_2,\ \neg p_3,\ p_4$

$x \geq 0,\ y = x + 1,$
$\neg(y > 2),\ y < 1$

Theory Solver

Unsatisfiable
$x \geq 0,\ y = x + 1,\ y < 1$

TU Delft

SAT + Theory solvers

Basic Idea

$x \geq 0,\ y = x + 1,\ (y > 2 \lor y < 1)$

Abstract (aka "naming" atoms)

$p_1,\ p_2,\ (p_3 \lor p_4)$

$p_1 \equiv (x \geq 0),\ p_2 \equiv (y = x + 1),$
$p_3 \equiv (y > 2),\ p_4 \equiv (y < 1)$

SAT Solver

Assignment
$p_1,\ p_2,\ \neg p_3,\ p_4$

$x \geq 0,\ y = x + 1,$
$\neg(y > 2),\ y < 1$

Theory Solver

Unsatisfiable
$x \geq 0,\ y = x + 1,\ y < 1$

New Lemma
$\neg p_1 \lor \neg p_2 \lor \neg p_4$

TUDelft

SAT + Theory solvers

New Lemma

$\neg p_1 \vee \neg p_2 \vee \neg p_4$

Unsatisfiable

$x \geq 0,\ y = x + 1,\ y < 1$

Theory Solver

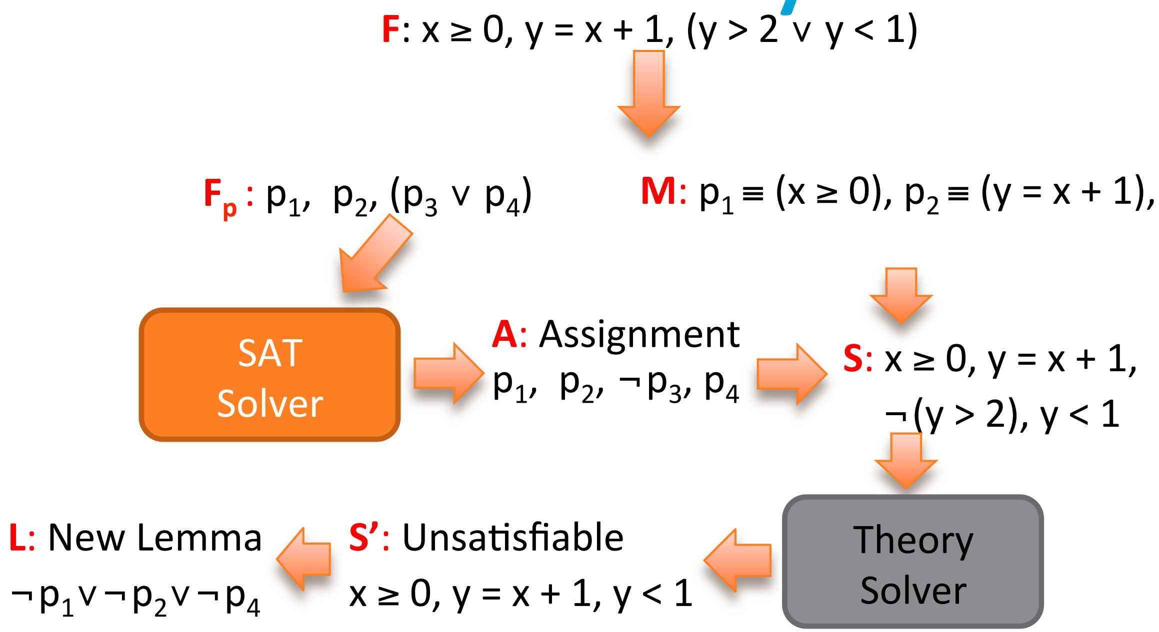
AKA
Theory conflict

SAT + Theory solvers: Main loop

```
procedure SmtSolver(F)
    (F_p, M) := Abstract(F)
    loop
        (R, A) := SAT_solver(F_p)
        if R = UNSAT then return UNSAT
        S := Concretize(A, M)
        (R, S') := Theory_solver(S)
        if R = SAT then return SAT
        L := New_Lemma(S', M)
        Add L to F_p
```

TU Delft

SAT + Theory solvers




```
procedure SMT_Solver(F)
    (Fp, M) := Abstract(F)
    loop
     (R, A) := SAT_solver(Fp)
     if R = UNSAT then return UNSAT
     S = Concretize(A, M)
     (R, S’) := Theory_solver(S)
     if R = SAT then return SAT
     L := New_Lemma(S, M)
     Add L to Fp
```

How can this be fast?

- SAT solvers are extremely efficient
- The obtained theory S is often easy to solve (not NP-hard)
- and

State-of-the-art SMT solvers implement many improvements…

TU Delft

SAT + Theory solvers

Incrementality

Send the literals to the Theory solver as they are assigned by the SAT solver

$p_1 \equiv (x \geq 0)$, $p_2 \equiv (y = x + 1)$,

$p_3 \equiv (y > 2)$, $p_4 \equiv (y < 1)$, $p_5 \equiv (x < 2)$,

$p_1, \; p_2, \; p_4 \;\; | \;\; p_1, \; p_2, (p_3 \vee p_4), (p_5 \vee \neg p_4)$

Partial assignment is already Theory inconsistent.

TU Delft

SAT + Theory solvers

Efficient Lemma Generation (computing a small S')

Avoid lemmas containing redundant literals.

$p_1 \equiv (x \geq 0)$, $p_2 \equiv (y = x + 1)$,

$p_3 \equiv (y > 2)$, $p_4 \equiv (y < 1)$, $p_5 \equiv (x < 2)$,

$p_1, \ p_2, p_3, p_4 \ \mid \ p_1, \ p_2, (p_3 \lor p_4), (p_5 \lor \neg p_4)$

$\neg p_1 \lor \neg p_2 \lor \neg p_3 \lor \neg p_4$

Imprecise Lemma

TUDelft

SAT + Theory solvers

Theory Propagation

$p_1 \equiv (x \geq 0)$, $p_2 \equiv (y = x + 1)$,

$p_3 \equiv (y > 2)$, $p_4 \equiv (y < 1)$, $p_5 \equiv (x < 2)$,

$p_1,\ p_2 \mid p_1,\ p_2,\ (p_3 \vee p_4),\ (p_5 \vee \neg p_4)$

p_1, p_2 imply $\neg p_4$ by theory propagation

$p_1,\ p_2,\ \neg p_4 \mid p_1,\ p_2,\ (p_3 \vee p_4),\ (p_5 \vee \neg p_4)$

SMT in Practice

- *from code to formulas*
- *following program execution*

From code to formulas

```
unsigned GCD(x, y) {
  requires(y > 0);
  while (true) {
 unsigned m = x % y;
   if (m == 0) return y;
   x = y;
   y = m;
    }
}
```

Give a trace where the loop is executed twice.

TUDelft

From code to formulas

```
unsigned GCD(x, y) {
  requires(y > 0);
  while (true) {
  unsigned m = x % y;
   if (m == 0) return y;
   x = y;
   y = m;
    }
}
```

SSA

Static Single Assignment

$(y_0 > 0)$ and
$(m_0 = x_0 \% y_0)$ and
not $(m_0 = 0)$ and
$(x_1 = y_0)$ and
$(y_1 = m_0)$ and
$(m_1 = x_1 \% y_1)$ and
$(m_1 = 0)$

Solver

Variable Assignmen

$x_0 = 2$
$y_0 = 4$
$m_0 = 2$
$x_1 = 4$
$y_1 = 2$
$m_1 = 0$

Give a trace where the loop is executed twice.

From code to formulas

```
unsigned GCD(x, y) {
  requires(y > 0);
  while (true) {
    unsigned m = x % y;
    if (m == 0) return y;
    x = y;
    y = m;
  }
}
```

SSA

Static Single Assignment

$(y_0 > 0)$ and
$(m_0 = x_0 \% y_0)$ and
not $(m_0 = 0)$ and
$(x_1 = y_0)$ and
$(y_1 = m_0)$ and
$(m_1 = x_1 \% y_1)$ and
$(m_1 = 0)$

Solver

Variable Assignmen

$x_0 = 2$
$y_0 = 4$
$m_0 = 2$
$x_1 = 4$
$y_1 = 2$
$m_1 = 0$

AKA Path Constraint

Symbolic execution

- Maps code and a target branch/line entirely into sets of constraints that can be solved using, e.g., an SMT solver
- Used to be a static analysis tool:
  - ***code is only interpreted, not executed!***
- Nowadays it is combined with actually running the code via dynamic analysis (concrete execution)
  - ***run, analyze, and iterate***
- Balance concrete and symbolic parts
- Many tools developed, and used to find real bugs:
  - PEX, KLEE, Angr, SAGE, Triton, …
  - see https://en.wikipedia.org/wiki/Symbolic_execution

TU Delft

Some more theory -- Difference Logic

Very useful in practice!

Most arithmetical constraints in software verification/analysis are in this fragment.

$x := x + 1$

$x_1 = x_0 + 1$

$x_1 - x_0 \leq 1,\ x_0 - x_1 \leq -1$

TU Delft

Difference Logic

Satisfiable?

$$\begin{array}{lcccc} z & - & t_{1,1} & \leq & 0 \\ z & - & t_{2,1} & \leq & 0 \\ z & - & t_{3,1} & \leq & 0 \\ t_{3,2} & - & z & \leq & 5 \\ t_{3,1} & - & t_{3,2} & \leq & -2 \\ t_{2,1} & - & t_{3,1} & \leq & -3 \\ t_{1,1} & - & t_{2,1} & \leq & -2 \end{array}$$

Difference Logic

NO!

$$\begin{array}{lcl}
z - t_{1,1} & \leq & 0 \\
z - t_{2,1} & \leq & 0 \\
z - t_{3,1} & \leq & 0 \\
t_{3,2} - z & \leq & 5 \\
t_{3,1} - t_{3,2} & \leq & -2 \\
t_{2,1} - t_{3,1} & \leq & -3 \\
t_{1,1} - t_{2,1} & \leq & -2
\end{array}$$

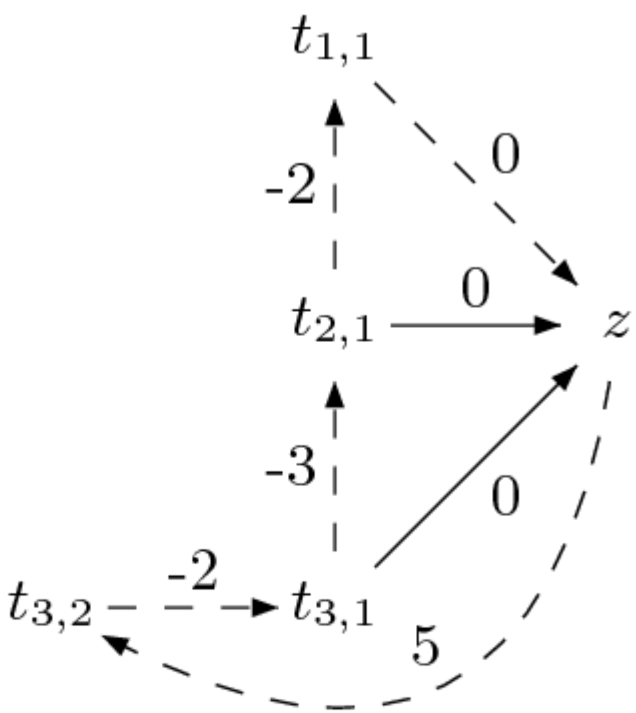
TU Delft

Difference Logic complexity

Satisfiable if and only if there are no negative cycles!

Algorithms based on Bellman-Ford (O(mn)).

$$
\begin{array}{lclcl}
z & - & t_{1,1} & \leq & 0 \\
z & - & t_{2,1} & \leq & 0 \\
z & - & t_{3,1} & \leq & 0 \\
t_{3,2} & - & z & \leq & 5 \\
t_{3,1} & - & t_{3,2} & \leq & -2 \\
t_{2,1} & - & t_{3,1} & \leq & -3 \\
t_{1,1} & - & t_{2,1} & \leq & -2
\end{array}
$$



Z3 – a frequently used solver

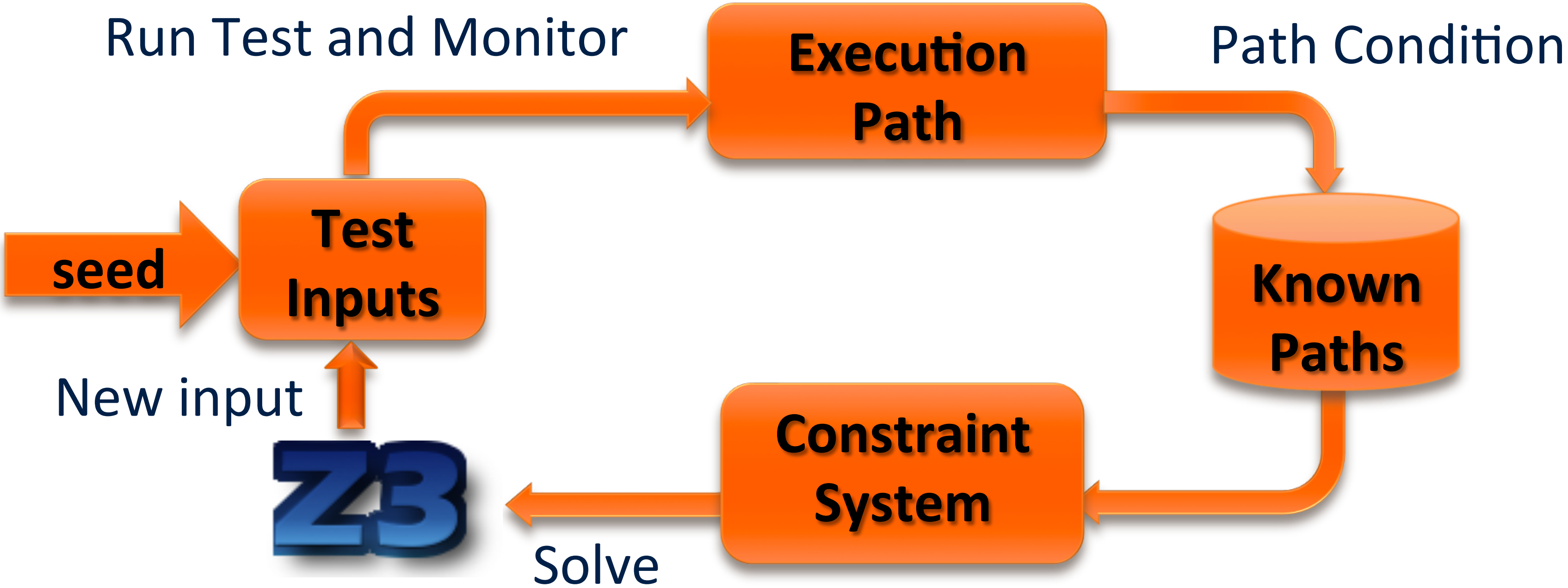
- https://github.com/Z3Prover/z3
- https://github.com/Z3Prover/z3/wiki

- One of the most powerful SMT solvers, developed by Microsoft, check out http://rise4fun.com/

- Can quickly solve problems such as Sudoku, Scheduling, …

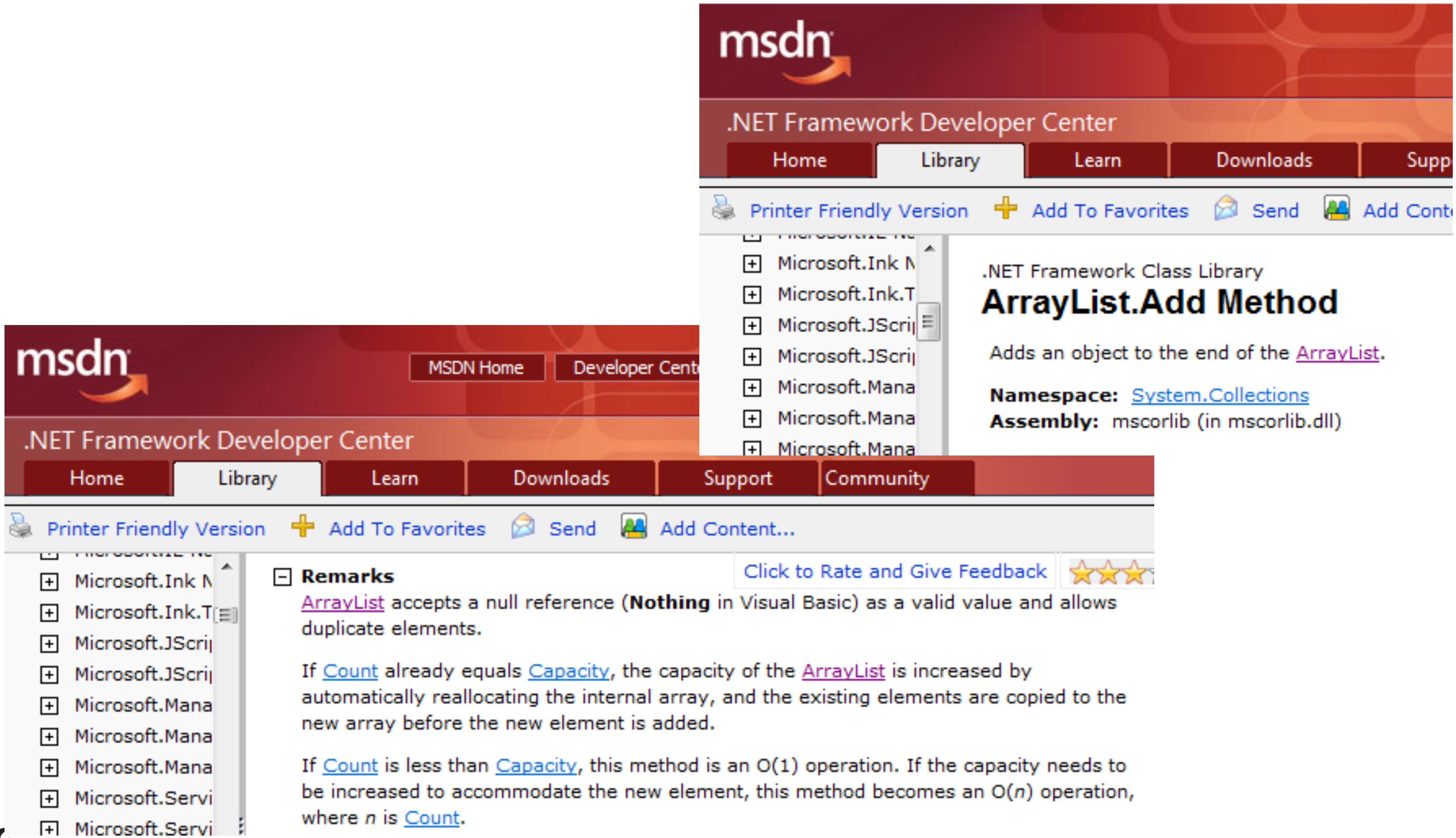
- *and efficiently analyze code (using difference logic)!*

Example: Directed Automated Random Testing (DART) in Microsoft PEX



TU Delft

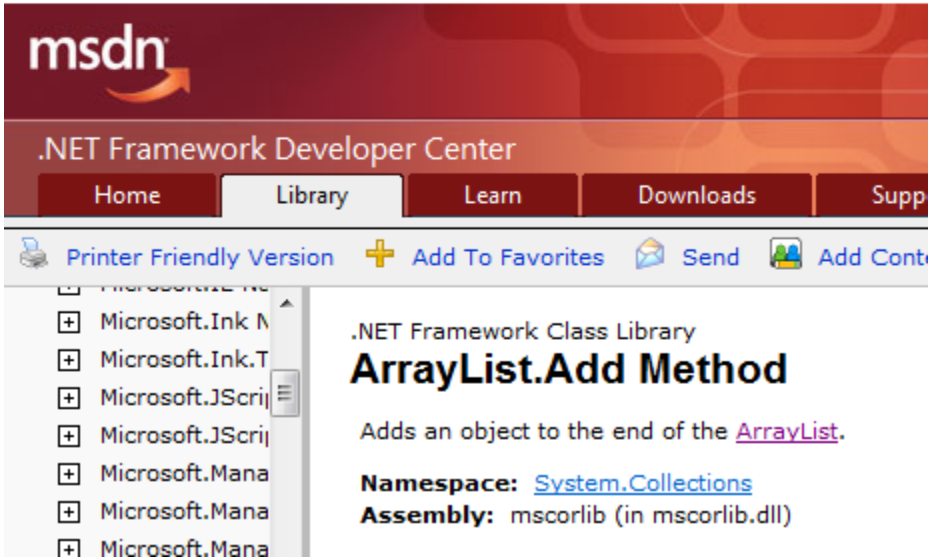
PEX concolic testing of ArrayList



ArrayList: AddItem Test

```
class ArrayListTest {
  [PexMethod]
  void AddItem(int c, object item) {
      var list = new ArrayList(c);
      list.Add(item);
      Assert(list[0] == item); }
}
```

```
class ArrayList {
  object[] items;
  int count;

  ArrayList(int capacity) {
    if (capacity < 0) throw ...;
    items = new object[capacity];
  }

  void Add(object item) {
    if (count == items.Length)
      ResizeArray();

    items[this.count++] = item; }
...
```



ArrayList: Starting Pex…

```
class ArrayListTest {
  [PexMethod]
  void AddItem(int c, object item) {
      var list = new ArrayList(c);
      list.Add(item);
      Assert(list[0] == item); }
}
```

```
class ArrayList {
  object[] items;
  int count;

  ArrayList(int capacity) {
    if (capacity < 0) throw ...;
    items = new object[capacity];
  }

  void Add(object item) {
    if (count == items.Length)
      ResizeArray();

    items[this.count++] = item; }
...
```

Inputs

ArrayList: Run 1, (0,null)

```
class ArrayListTest {
  [PexMethod]
  void AddItem(int c, object item) {
      var list = new ArrayList(c);
      list.Add(item);
      Assert(list[0] == item); }
}
```

Inputs
(0,null)

```
class ArrayList {
  object[] items;
  int count;

  ArrayList(int capacity) {
    if (capacity < 0) throw ...;
    items = new object[capacity];
  }

  void Add(object item) {
    if (count == items.Length)
      ResizeArray();

    items[this.count++] = item; }
...
```

TUDelft

ArrayList: Run 1, (0,null)

```
class ArrayListTest {
  [PexMethod]
  void AddItem(int c, object item) {
      var list = new ArrayList(c);
      list.Add(item);
      Assert(list[0] == item); }
}
```

	Inputs	Observed Constraints
	(0,null)	!(c<0)

```
class ArrayList {
  object[] items;
  int count;

  ArrayList(int capacity) {
    if (capacity < 0) throw ...;
    items = new object[capacity];
  }

  void Add(object item) {
    if (count == items.Length)
      ResizeArray();

    items[this.count++] = item; }
...
```

c < 0 → false

ArrayList: Run 1, (0,null)

```
class ArrayListTest {
  [PexMethod]
  void AddItem(int c, object item) {
      var list = new ArrayList(c);
      list.Add(item);
      Assert(list[0] == item); }
}
```

	Inputs	Observed Constraints
	(0,null)	!(c<0) && 0==c

```
class ArrayList {
  object[] items;
  int count;

  ArrayList(int capacity) {
    if (capacity < 0) throw ...;
    items = new object[capacity];
  }

  void Add(object item) {
    if (count == items.Length)
      ResizeArray();

    items[this.count++] = item; }
...
```

0 == c → true

ArrayList: Run 1, (0,null)

```
class ArrayListTest {
  [PexMethod]
  void AddItem(int c, object item) {
      var list = new ArrayList(c);
      list.Add(item);
      Assert(list[0] == item); }
}
```

	Inputs	Observed Constraints
	(0,null)	!(c<0) && 0==c

`item == item → true`

```
class ArrayList {
  object[] items;
  int count;

  ArrayList(int capacity) {
    if (capacity < 0) throw ...;
    items = new object[capacity];
  }

  void Add(object item) {
    if (count == items.Length)
      ResizeArray();

    items[this.count++] = item; }
...
```

This is a *tautology*,
i.e. a constraint that is always true, regardless of the chosen values.

We can ignore such constraints.

TU Delft

ArrayList: Run 1, (0,null)

```
class ArrayListTest {
  [PexMethod]
  void AddItem(int c, object item) {
      var list = new ArrayList(c);
      list.Add(item);
      Assert(list[0] == item); }
}
```

```
class ArrayList {
  object[] items;
  int count;

  ArrayList(int capacity) {
    if (capacity < 0) throw ...;
    items = new object[capacity];
  }

  void Add(object item) {
    if (count == items.Length)
      ResizeArray();

    items[this.count++] = item; }
...
```

	Inputs	Observed Constraints
	(0,null)	!(c<0) && 0==c

Q: How to trigger another branch?

ArrayList: Picking next branch

```
class ArrayListTest {
  [PexMethod]
  void AddItem(int c, object item) {
      var list = new ArrayList(c);
      list.Add(item);
      Assert(list[0] == item); }
}
```

```
class ArrayList {
  object[] items;
  int count;

  ArrayList(int capacity) {
    if (capacity < 0) throw ...;
    items = new object[capacity];
  }

  void Add(object item) {
    if (count == items.Length)
      ResizeArray();

    items[this.count++] = item; }
...
```

Constraints to solve	Inputs	Observed Constraints
	(0,null)	!(c<0) && 0==c
!(c<0) && 0!=c		

Negate the observed constraints!

ArrayList: Solve using SMT solver

```
class ArrayListTest {
  [PexMethod]
  void AddItem(int c, object item) {
      var list = new ArrayList(c);
      list.Add(item);
      Assert(list[0] == item); }
}
```

```
class ArrayList {
  object[] items;
  int count;

  ArrayList(int capacity) {
    if (capacity < 0) throw ...;
    items = new object[capacity];
  }

  void Add(object item) {
    if (count == items.Length)
      ResizeArray();

    items[this.count++] = item; }
...
```

Constraints to solve	Inputs	Observed Constraints
	(0,null)	!(c<0) && 0==c
!(c<0) &&	**(1,null)**	

ArrayList: Run 2, (1, null)

```
class ArrayListTest {
  [PexMethod]
  void AddItem(int c, object item) {
      var list = new ArrayList(c);
      list.Add(item);
      Assert(list[0] == item); }
}
```

Constraints to solve	Inputs	Observed Constraints
	(0,null)	!(c<0) && 0==c
!(c<0) &&	(1,null)	!(c<0) && 0!=c

```
class ArrayList {
  object[] items;
  int count;

  ArrayList(int capacity) {
    if (capacity < 0) throw ...;
    items = new object[capacity];
  }

  void Add(object item) {
    if (count == items.Length)
      ResizeArray();

    items[this.count++] = item; }
...
```

0 == c → false

TU Delft

ArrayList: Pick new branch

```
class ArrayListTest {
  [PexMethod]
  void AddItem(int c, object item) {
      var list = new ArrayList(c);
      list.Add(item);
      Assert(list[0] == item); }
}
```

```
class ArrayList {
  object[] items;
  int count;

  ArrayList(int capacity) {
    if (capacity < 0) throw ...;
    items = new object[capacity];
  }

  void Add(object item) {
    if (count == items.Length)
      ResizeArray();

    items[this.count++] = item; }
...
```

Constraints to solve	Inputs	Observed Constraints
	(0,null)	!(c<0) && 0==c
!(c<0) &&	(1,null)	!(c<0) && 0!=c
c<0		

Z3

ArrayList: Run 3, (-1, null)

```
class ArrayListTest {
  [PexMethod]
  void AddItem(int c, object item) {
      var list = new ArrayList(c);
      list.Add(item);
      Assert(list[0] == item); }
}
```

```
class ArrayList {
  object[] items;
  int count;

  ArrayList(int capacity) {
    if (capacity < 0) throw ...;
    items = new object[capacity];
  }

  void Add(object item) {
    if (count == items.Length)
      ResizeArray();

    items[this.count++] = item; }
...
```

Constraints to solve	Inputs	Observed Constraints
	(0,null)	!(c<0) && 0==c
!(c<0) &&	(1,null)	!(c<0) && 0!=c
c<0	**(-1,null)**	

ArrayList: Run 3, (-1, null)

```
class ArrayListTest {
  [PexMethod]
  void AddItem(int c, object item) {
      var list = new ArrayList(c);
      list.Add(item);
      Assert(list[0] == item); }
}
```

Constraints to solve	Inputs	Observed Constraints
	(0,null)	!(c<0) && 0==c
!(c<0) &&	(1,null)	!(c<0) && 0!=c
c<0	(-1,null)	c<0

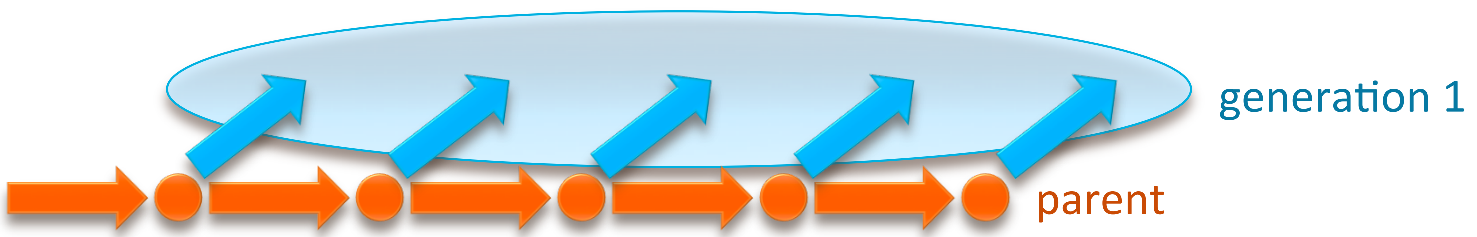
```
class ArrayList {
  object[] items;
  int count;

  ArrayList(int capacity) {
    if (capacity < 0) throw ...;
    items = new object[capacity];
  }

  void Add(object item) {
    if (count == items.Length)
      ResizeArray();

    items[this.count++] = item; }
...
```

c < 0 → true

TUDelft

SAGE

- Apply DART to large applications (not units).
- Start with well-formed input (not random).
- Combine with generational search (not DFS).
  - Negate 1-by-1 each constraint in a path constraint.
  - Generate many children for each parent run.



Zero to Crash in 10 Generations

- Starting with 100 zero bytes …
- SAGE generates a crashing test for Media1 parser

```
00000000h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000010h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000020h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000030h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000040h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000050h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000060h: 00 00 00 00                                     ; ....
```

Generation 0

Zero to Crash in 10 Generations

- Starting with 100 zero bytes …
- SAGE generates a crashing test for Media1 parser

```
00000000h: 52 49 46 46 00 00 00 00 00 00 00 00 00 00 00 00 ; RIFF............
00000010h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000020h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000030h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000040h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000050h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000060h: 00 00 00 00                                     ; ....
```

Generation 1

Zero to Crash in 10 Generations

- Starting with 100 zero bytes …
- SAGE generates a crashing test for Media1 parser

```
00000000h: 52 49 46 46 00 00 00 00 ** ** ** 20 00 00 00 00 ; RIFF....***....
00000010h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000020h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000030h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000040h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000050h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000060h: 00 00 00 00                                     ; ....
```

Generation 2

Zero to Crash in 10 Generations

- Starting with 100 zero bytes …
- SAGE generates a crashing test for Media1 parser

```
00000000h: 52 49 46 46 3D 00 00 00 ** ** ** 20 00 00 00 00 ; RIFF=...*** ....
00000010h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000020h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000030h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000040h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000050h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000060h: 00 00 00 00                                     ; ....
```

Generation 3

TU Delft

SMT@Microsoft

Zero to Crash in 10 Generations

- Starting with 100 zero bytes …
- SAGE generates a crashing test for Media1 parser

```
00000000h: 52 49 46 46 3D 00 00 00 ** ** ** 20 00 00 00 00 ; RIFF=...*** ....
00000010h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000020h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000030h: 00 00 00 00 73 74 72 68 00 00 00 00 00 00 00 00 ; ....strh........
00000040h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000050h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000060h: 00 00 00 00                                     ; ....
```

Generation 4

TU Delft

Zero to Crash in 10 Generations

- Starting with 100 zero bytes …
- SAGE generates a crashing test for Media1 parser

```
00000000h: 52 49 46 46 3D 00 00 00 ** ** ** 20 00 00 00 00 ; RIFF=...*** ....
00000010h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000020h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000030h: 00 00 00 00 73 74 72 68 00 00 00 00 76 69 64 73 ; ....strh....vids
00000040h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000050h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000060h: 00 00 00 00                                     ; ....
```

Generation 5

TU Delft

SMT@Microsoft

Zero to Crash in 10 Generations

- Starting with 100 zero bytes …
- SAGE generates a crashing test for Media1 parser

```
00000000h: 52 49 46 46 3D 00 00 00 ** ** ** 20 00 00 00 00 ; RIFF=...*** ....
00000010h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000020h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000030h: 00 00 00 00 73 74 72 68 00 00 00 00 76 69 64 73 ; ....strh....vids
00000040h: 00 00 00 00 73 74 72 66 00 00 00 00 00 00 00 00 ; ....strf........
00000050h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000060h: 00 00 00 00                                     ; ....
```

Generation 6

TU Delft

SMT@Microsoft

Zero to Crash in 10 Generations

- Starting with 100 zero bytes …
- SAGE generates a crashing test for Media1 parser

```
00000000h: 52 49 46 46 3D 00 00 00 ** ** ** 20 00 00 00 00 ; RIFF=...*** ....
00000010h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000020h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000030h: 00 00 00 00 73 74 72 68 00 00 00 00 76 69 64 73 ; ....strh....vids
00000040h: 00 00 00 00 73 74 72 66 00 00 00 00 28 00 00 00 ; ....strf....(...
00000050h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000060h: 00 00 00 00                                     ; ....
```

Generation 7

TUDelft

SMT@Microsoft

Zero to Crash in 10 Generations

- Starting with 100 zero bytes …
- SAGE generates a crashing test for Media1 parser

```
00000000h: 52 49 46 46 3D 00 00 00 ** ** ** 20 00 00 00 00 ; RIFF=...*** ....
00000010h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000020h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000030h: 00 00 00 00 73 74 72 68 00 00 00 00 76 69 64 73 ; ....strh....vids
00000040h: 00 00 00 00 73 74 72 66 00 00 00 00 28 00 00 00 ; ....strf....(...
00000050h: 00 00 00 00 00 00 00 00 00 00 00 00 C9 9D E4 4E ; ............É  äN
00000060h: 00 00 00 00                                     ; ....
```

Generation 8

TU Delft

SMT@Microsoft

Zero to Crash in 10 Generations

- Starting with 100 zero bytes …
- SAGE generates a crashing test for Media1 parser

```
00000000h: 52 49 46 46 3D 00 00 00 ** ** ** 20 00 00 00 00 ; RIFF=...*** ....
00000010h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000020h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000030h: 00 00 00 00 73 74 72 68 00 00 00 00 76 69 64 73 ; ....strh....vids
00000040h: 00 00 00 00 73 74 72 66 00 00 00 00 28 00 00 00 ; ....strf....(...
00000050h: 00 00 00 00 00 00 00 00 00 00 00 00 01 00 00 00 ; ................
00000060h: 00 00 00 00                                     ; ....
```

Generation 9

TU Delft

SMT@Microsoft

Zero to Crash in 10 Generations

- Starting with 100 zero bytes …
- SAGE generates a crashing test for Media1 parser

```
00000000h: 52 49 46 46 3D 00 00 00 ** ** ** 20 00 00 00 00 ; RIFF=...*** ....
00000010h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000020h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000030h: 00 00 00 00 73 74 72 68 00 00 00 00 76 69 64 73 ; ....strh....vids
00000040h: 00 00 00 00 73 74 72 66 B2 75 76 3A 28 00 00 00 ; ....strf²uv:(...
00000050h: 00 00 00 00 00 00 00 00 00 00 00 00 01 00 00 00 ; ................
00000060h: 00 00 00 00                                     ; ....
```

Generation 10

CRASH

TUDelft

SAGE (cont.)

- SAGE is very effective at finding bugs.
- Works on large applications.
- Fully automated
- Easy to deploy (x86 analysis – any language)
- Used in various groups inside Microsoft
- Powered by Z3.

Concolic execution in research/practice

- Godefroid, Patrice, et al. "Automating software testing using program analysis.”, 2008.
- Godefroid, Patrice, Michael Y. Levin, and David Molnar. "SAGE: whitebox fuzzing for security testing.”, 2012. Sen, Koushik. "Concolic testing.”, 2007.
- Cadar, Cristian, and Koushik Sen. "Symbolic execution for software testing: three decades later.”, 2013.
- Stephens, Nick, et al. "Driller: Augmenting Fuzzing Through Selective Symbolic Execution.”, 2016.
- Shoshitaishvili, Yan, et al. "Sok:(state of) the art of war: Offensive techniques in binary analysis.”, 2016.
- Cha, Sang Kil, et al. "Unleashing mayhem on binary code.”, 2012.

TU Delft

Binaries

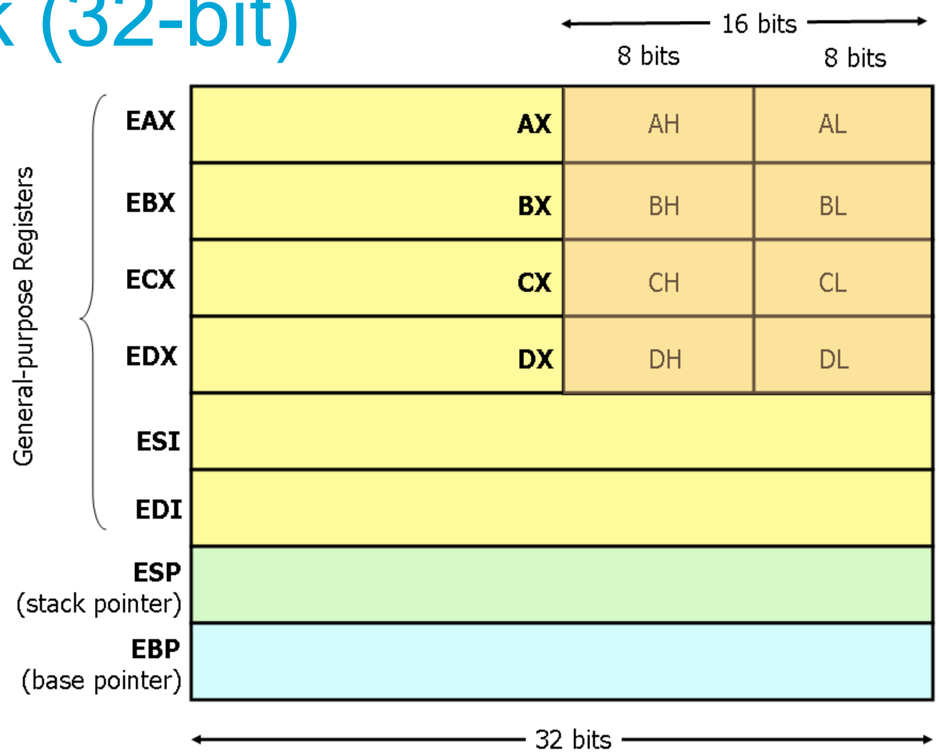
- instrumentation and tainting
- concolic execution

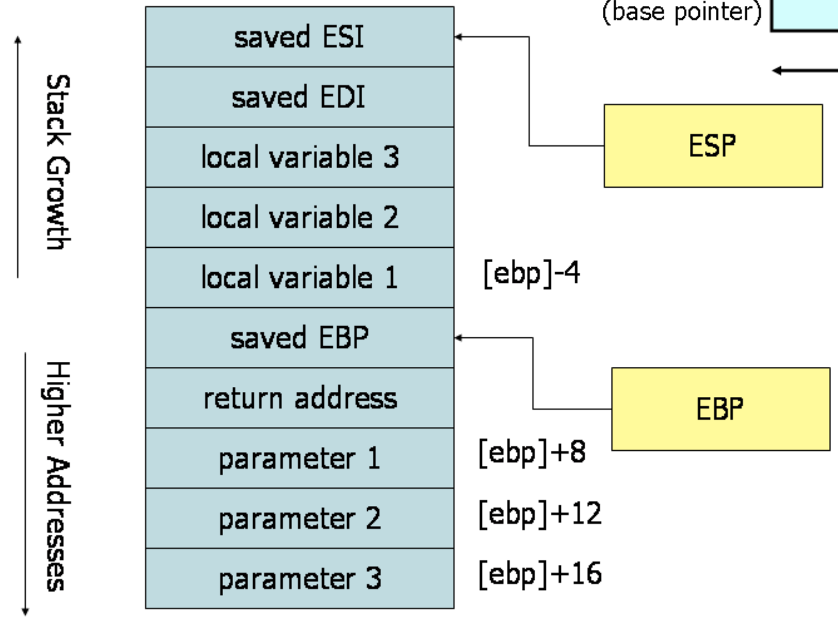
Instrumentation

```
initial_instruction_1
initial_instruction_2
initial_instruction_3
initial_instruction_4
```

```
jmp_call_back_before
initial_instruction_1
jmp_call_back_after

jmp_call_back_before
initial_instruction_2
jmp_call_back_after

jmp_call_back_before
initial_instruction_3
jmp_call_back_after

jmp_call_back_before
initial_instruction_4
jmp_call_back_after
```

Recap: the stack (32-bit)





TUDelft

Pin

- Developed by Intel
- Pin is a dynamic binary instrumentation framework for the IA-32 and x86-64 instruction-set architectures
  - The tools created using Pin, called Pintools, can be used to perform program analysis on user space applications in Linux and Windows

TU Delft

Example Pintool (inscount0.cpp)

```
...

static UINT64 icount = 0;

VOID docount() { icount++; }

VOID Instruction(INS ins, VOID *v) {
  INS_InsertCall(ins, IPOINT_BEFORE, (AFUNPTR)docount, IARG_END);
}

...

int main(int argc, char * argv[])
{
    if (PIN_Init(argc, argv)) return Usage();

    OutFile.open(KnobOutputFile.Value().c_str());

    // Register Instruction to be called to instrument instructions
    INS_AddInstrumentFunction(Instruction, 0);

    // Register Fini to be called when the application exits
    PIN_AddFiniFunction(Fini, 0);

    // Start the program, never returns
    PIN_StartProgram();

    return 0;
}
```

TU Delft

Example Pintool (syscount.cpp)

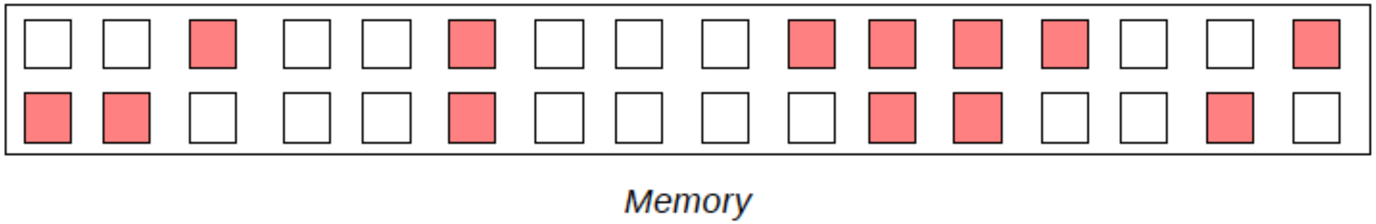
```
...
static UINT64 icount = 0;

VOID docount() { icount++; }

VOID Instruction(INS ins, VOID *v) {
  if(INS_IsSyscall(ins)){
    INS_InsertCall(ins, IPOINT_BEFORE, (AFUNPTR)docount, IARG_END);
  }
}

...

int main(int argc, char * argv[])
{
    if (PIN_Init(argc, argv)) return Usage();

    OutFile.open(KnobOutputFile.Value().c_str());

    // Register Instruction to be called to instrument instructions
    INS_AddInstrumentFunction(Instruction, 0);

    // Register Fini to be called when the application exits
    PIN_AddFiniFunction(Fini, 0);

    // Start the program, never returns
    PIN_StartProgram();

    return 0;
}
```

Example Pintool (writecount.cpp)

```
...

static UINT64 icount = 0;

VOID docount() { icount++; }

VOID Instruction(INS ins, VOID *v) {

  if(INS_IsSyscall(ins) && INS_IsMemoryWrite(ins)){

    INS_InsertCall(ins, IPOINT_BEFORE, (AFUNPTR)docount, IARG_END);
  }
}

...

int main(int argc, char * argv[])
{
    if (PIN_Init(argc, argv)) return Usage();

    OutFile.open(KnobOutputFile.Value().c_str());

    // Register Instruction to be called to instrument instructions
    INS_AddInstrumentFunction(Instruction, 0);

    // Register Fini to be called when the application exits
    PIN_AddFiniFunction(Fini, 0);

    // Start the program, never returns
    PIN_StartProgram();

    return 0;
}
```

Tainting

- We are interested in memory reads and writes
- The goal is to capture the effects of input modifications
- We keep track of which memory locations and CPU registers can be influenced by the input data
- This is called tainting



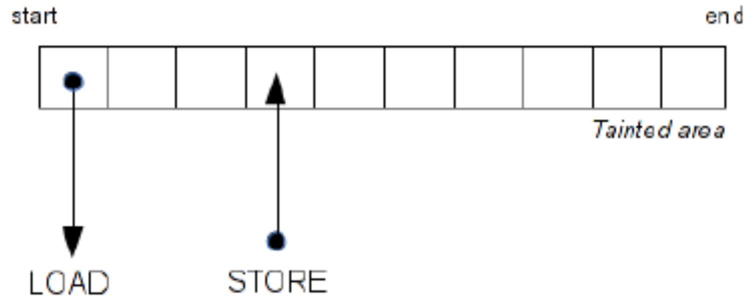
Tainting in Pin

- Instrument file read syscalls:

```
...
VOID Syscall_entry(THREADID thread_id, CONTEXT *ctx, SYSCALL_STANDARD std, void *v)
{
  struct range taint;

  // file read syscall nr on my Macbook
  if (PIN_GetSyscallNumber(ctx, std) == 33554435){

      start = static_cast<UINT64>((PIN_GetSyscallArgument(ctx, std, 1)));
      size  = static_cast<UINT64>((PIN_GetSyscallArgument(ctx, std, 2)));

      taint.start = start;
      taint.end   = start + size;
      bytesTainted.push_back(taint);

      std::cout << "[TAINT]\t\t\tbytes tainted from " << std::hex << "0x" <<
          taint.start << " to 0x" << taint.end << " (via read)"<< std::endl;
  }
}
...

int main(int argc, char *argv[]){
    ...
    PIN_AddSyscallEntryFunction(Syscall_entry, 0);
...
}
```

Tainting in Pin

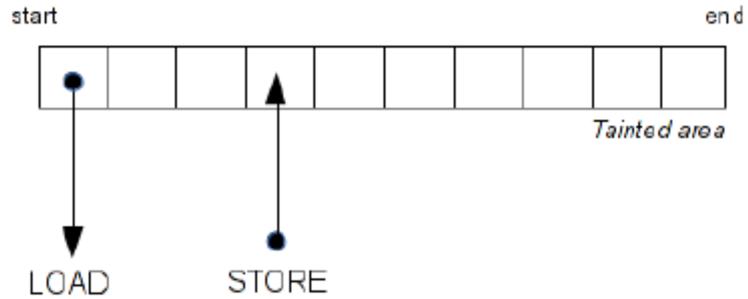
- Instrument memory reads and check taint: *mov regA, [regB]*




```
VOID ReadMem(UINT64 insAddr, std::string insDis, UINT64 memOp){
  list<struct range>::iterator i;
  UINT64 addr = memOp;

  for(i = bytesTainted.begin(); i != bytesTainted.end(); ++i){
      if (addr >= i->start && addr < i->end){
        std::cout << std::hex << "[READ in " << addr << "]\t" << insAddr << ": " <<
          insDis << std::endl;
      }
  }
}


VOID Instruction(INS ins, VOID *v){
  if (INS_MemoryOperandIsRead(ins, 0) && INS_OperandIsReg(ins, 0)){
    INS_InsertCall(
        ins, IPOINT_BEFORE, (AFUNPTR)ReadMem,
        IARG_ADDRINT, INS_Address(ins),
        IARG_PTR, new string(INS_Disassemble(ins)),
        IARG_MEMORYOP_EA, 0,
        IARG_END);
  }
}
```

Tainting in Pin

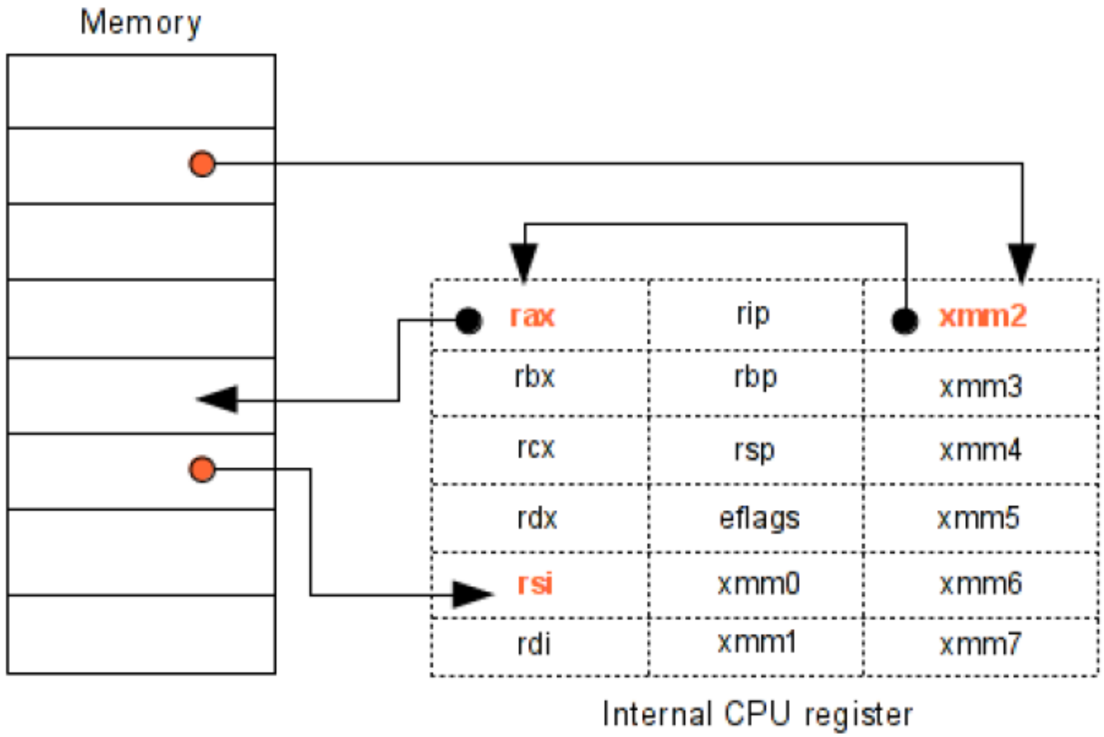
- Instrument memory writes and check taint: *mov [regA], regB*




```
VOID WriteMem(UINT64 insAddr, std::string insDis, UINT64 memOp){
  list<struct range>::iterator i;
  UINT64 addr = memOp;

  for(i = bytesTainted.begin(); i != bytesTainted.end(); ++i){
      if (addr >= i->start && addr < i->end){
        std::cout << std::hex << "[WRITE in " << addr << "]\t" << insAddr << ": " <<
          insDis << std::endl;
      }
  }
}


VOID Instruction(INS ins, VOID *v){
  if (INS_MemoryOperandIsWritten(ins, 0)){
    INS_InsertCall(
        ins, IPOINT_BEFORE, (AFUNPTR)WriteMem,
        IARG_ADDRINT, INS_Address(ins),
        IARG_PTR, new string(INS_Disassemble(ins)),
        IARG_MEMORYOP_EA, 0,
        IARG_END);
  }
}
```

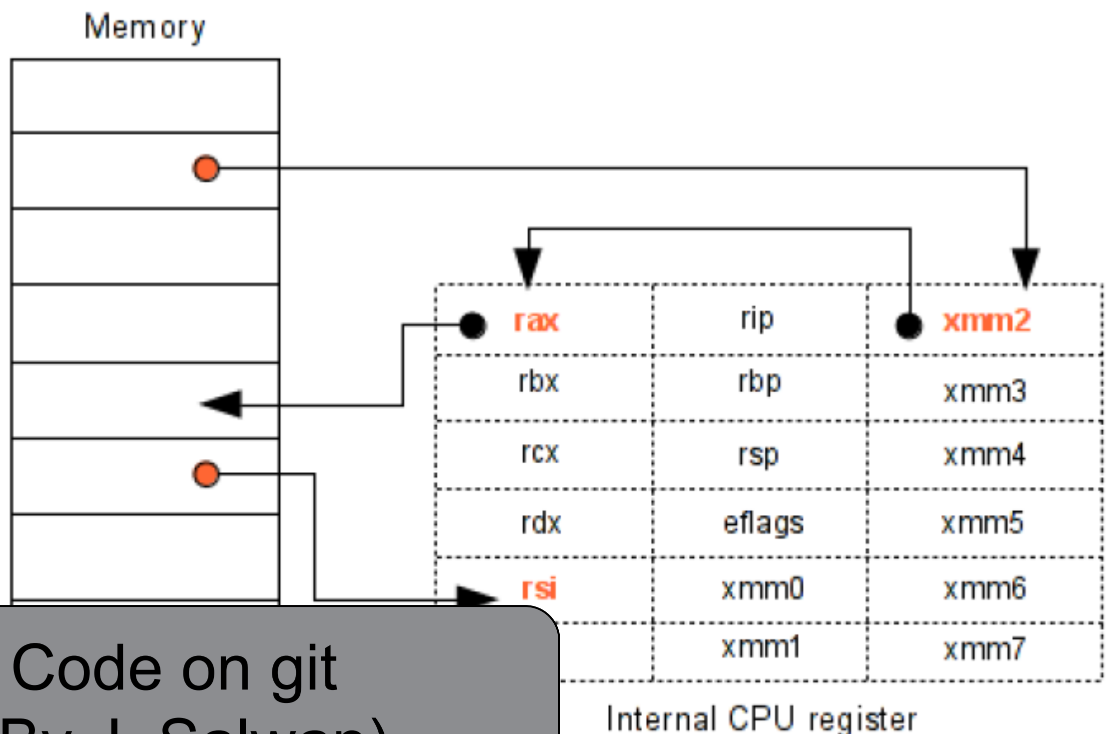
Tainting memory is not enough!

- By monitoring all STORE/LOAD and GET/PUT instructions, from memory to registers, and between registers, we know exactly which memory areas, and code, can be influenced



Tainting memory is not enough!

- By monitoring all STORE/LOAD and GET/PUT instructions, from memory to registers, and between registers, we know exactly which memory areas, and code, can be influenced



Code on git
(By J. Salwan)

Tainting is already very useful!

- Wang, Zhi, et al. "ReFormat: Automatic reverse engineering of encrypted messages.", 2009
- Caballero, Juan, et al. "Dispatcher: Enabling active botnet infiltration using automatic protocol reverse-engineering.", 2009.
- Cui, Weidong, et al. "Tupni: Automatic reverse engineering of input formats.", 2008.
- Caballero, Juan, and Dawn Song. "Automatic protocol reverse-engineering: Message format extraction and field semantics inference.", 2013.
- Bosman, Erik, Asia Slowinska, and Herbert Bos. "Minemu: The world's fastest taint tracker.", 2011.
- Portokalidis, Georgios, Asia Slowinska, and Herbert Bos. "Argos: an emulator for fingerprinting zero-day attacks for advertised honeypots with automatic signature generation.", 2006.

TU Delft

Concolic execution using tainting

- We can keep track of a single static assignment of symbolic expressions

```
mov eax 1
add eax 2
mov ebx, eax
```

Q: What is the single static assignment?

Concolic execution using tainting

- We can keep track of a single static assignment of symbolic expressions

```
mov eax 1           e1 = 1
add eax 2           e2 = e1 + 2
mov ebx, eax        e3 = e2
```

- We also keep track of which register contains which expression

Q: What register contains which expression?

Concolic execution using tainting

- We can keep track of a single static assignment of symbolic expressions

```
mov eax 1            e1 = 1
add eax 2            e2 = e1 + 2
mov ebx, eax         e3 = e2
```

- We also keep track of which register contains which expression

```
eax e2
ebx e3
...
```

TUDelft

Concolic execution using tainting

- We can keep track of a single static assignment of symbolic expressions

```
mov eax 1                   e1 = 1
add eax 2                   e2 = e1 + 2
mov ebx, eax                e3 = e2
```

- We also keep track of which register contains which expression

```
eax e2
ebx e3
...
```

We can do this for any binary, for instance the GCD…

TU Delft

GCD disassembled (using Hopper)

```
; ==== B E G I N N I N G   O F   P R O C E D U R E ====

       ; Variables:
       ;    var_4: -4
       ;    var_8: -8
       ;    var_C: -12


                     __Z3GCDii:        // GCD(int, int)
0000000100000f40         push        rbp
0000000100000f41         mov         rbp, rsp
0000000100000f44         mov         dword [rbp+var_4], edi
0000000100000f47         mov         dword [rbp+var_8], esi

                     loc_100000f4a:
0000000100000f4a         mov         eax, dword [rbp+var_4]
0000000100000f4d         cdq
0000000100000f4e         idiv        dword [rbp+var_8]
0000000100000f51         mov         dword [rbp+var_C], edx
0000000100000f54         cmp         dword [rbp+var_C], 0x0
0000000100000f58         jne         loc_100000f63

0000000100000f5e         mov         eax, dword [rbp+var_8]
0000000100000f61         pop         rbp
0000000100000f62         ret
                        ; endp

                     loc_100000f63:
0000000100000f63         mov         eax, dword [rbp+var_8]
0000000100000f66         mov         dword [rbp+var_4], eax
0000000100000f69         mov         eax, dword [rbp+var_C]
0000000100000f6c         mov         dword [rbp+var_8], eax
0000000100000f6f         jmp         loc_100000f4a
0000000100000f74         align       128
```

TUDelft

GCD disassembled (using Hopper)

```
; ==== B E G I N N I N G   O F   P R O C E D U R E ====

       ; Variables:
       ;    var_4: -4
       ;    var_8: -8
       ;    var_C: -12


                    __Z3GCDii:        // GCD(int, int)
0000000100000f40        push        rbp
0000000100000f41        mov         rbp, rsp
0000000100000f44        mov         dword [rbp+var_4], edi
0000000100000f47        mov         dword [rbp+var_8], esi

                    loc_100000f4a:
0000000100000f4a        mov         eax, dword [rbp+var_4]
0000000100000f4d        cdq
0000000100000f4e        idiv        dword [rbp+var_8]
0000000100000f51        mov         dword [rbp+var_C], edx
0000000100000f54        cmp         dword [rbp+var_C], 0x0
0000000100000f58        jne         loc_100000f63

0000000100000f5e        mov         eax, dword [rbp+var_8]
0000000100000f61        pop         rbp
0000000100000f62        ret
                       ; endp

                    loc_100000f63:
0000000100000f63        mov         eax, dword [rbp+var_8]
0000000100000f66        mov         dword [rbp+var_4], eax
0000000100000f69        mov         eax, dword [rbp+var_C]
0000000100000f6c        mov         dword [rbp+var_8],
0000000100000f6f        jmp         loc_100000f4a
0000000100000f74        align       128
```

Q: What is the single static assignment until first jne?

GCD disassembled (using Hopper)

```
; ==== B E G I N N I N G   O F   P R O C E D U R E ====          Symbolic Expression Set

       ; Variables:
       ;    var_4: -4
       ;    var_8: -8
       ;    var_C: -12


                    __Z3GCDii:         // GCD(int, int)
0000000100000f40        push        rbp
0000000100000f41        mov         rbp, rsp
0000000100000f44        mov         dword [rbp+var_4], edi             ---> e1 (var_4) = #1
0000000100000f47        mov         dword [rbp+var_8], esi             ---> e2 (var_8) = #2

                    loc_100000f4a:
0000000100000f4a        mov         eax, dword [rbp+var_4]             ---> e3 (eax) = e1
0000000100000f4d        cdq                                            ---> e4 (eax) = e3 / e2
0000000100000f4e        idiv        dword [rbp+var_8]                  ---> e5 (edx) = e3 % e2
0000000100000f51        mov         dword [rbp+var_C], edx             ---> e6 (var_C) = e5
0000000100000f54        cmp         dword [rbp+var_C], 0x0             ---> e7 compare (e6, 0)
0000000100000f58        jne         loc_100000f63                      jump not equal

0000000100000f5e        mov         eax, dword [rbp+var_8]
0000000100000f61        pop         rbp
0000000100000f62        ret
                      ; endp

                    loc_100000f63:
0000000100000f63        mov         eax, dword [rbp+var_8]
0000000100000f66        mov         dword [rbp+var_4], eax
0000000100000f69        mov         eax, dword [rbp+var_C]
0000000100000f6c        mov         dword [rbp+var_8], eax
0000000100000f6f        jmp         loc_100000f4a
0000000100000f74        align       128
```

GCD disassembled (using Hopper)

```
; ==== B E G I N N I N G   O F   P R O C E D U R E ====            Symbolic Expression Set

       ; Variables:
       ;    var_4: -4
       ;    var_8: -8
       ;    var_C: -12


                     __Z3GCDii:        // GCD(int, int)
0000000100000f40         push        rbp
0000000100000f41         mov         rbp, rsp
0000000100000f44         mov         dword [rbp+var_4], edi         ---> e1 (var_4) = #1
0000000100000f47         mov         dword [rbp+var_8], esi         ---> e2 (var_8) = #2

                     loc_100000f4a:
0000000100000f4a         mov         eax, dword [rbp+var_4]         ---> e3 (eax) = e1
0000000100000f4d         cdq                                        ---> e4 (eax) = e3 / e2
0000000100000f4e         idiv        dword [rbp+var_8]              ---> e5 (edx) = e3 % e2
0000000100000f51         mov         dword [rbp+var_C], edx         ---> e6 (var_C) = e5
0000000100000f54         cmp         dword [rbp+var_C], 0x0         ---> e7 compare (e6, 0)
0000000100000f58         jne         loc_100000f63                  jump not equal

0000000100000f5e         mov         eax, dword [rbp+var_8]
0000000100000f61         pop         rbp
0000000100000f62         ret
                            ; endp

                     loc_100000f63:
0000000100000f63         mov         eax, dword [rbp+var_8]
0000000100000f66         mov         dword [rbp+var_4], eax
0000000100000f69         mov         eax, dword [rbp+var_C]
0000000100000f6c         mov         dword [rbp+var_8], eax
0000000100000f6f         jmp         loc_100000f4a
0000000100000f74         align       128
```

Q: What is the path constraint?

TUDelft

GCD disassembled (using Hopper)

```
; ==== B E G I N N I N G   O F   P R O C E D U R E ====

       ; Variables:
       ;    var_4: -4
       ;    var_8: -8
       ;    var_C: -12
```

1st iteration:

e6 != 0

e5 != 0

(e3%e2) != 0

(e1%e2) != 0

(#1%#2) != 0

```
                    loc_100000f63:
0000000100000f63        mov        eax, dword [rbp+var_8]
0000000100000f66        mov        dword [rbp+var_4], eax
0000000100000f69        mov        eax, dword [rbp+var_C]
0000000100000f6c        mov        dword [rbp+var_8], eax
0000000100000f6f        jmp        loc_100000f4a
0000000100000f74        align      128
```

Symbolic Expression Set

```
--->  e1 (var_4) = #1
--->  e2 (var_8) = #2

--->  e3 (eax) = e1
--->  e4 (eax) = e3 / e2
--->  e5 (edx) = e3 % e2
--->  e6 (var_C) = e5
--->  e7 compare (e6, 0)
jump not equal
```

GCD disassembled (using Hopper)

```
; ==== B E G I N N I N G   O F   P R O C E D U R E ====

        ; Variables:
        ;    var_4: -4
        ;    var_8: -8
        ;    var_C: -12
```

Symbolic Expression Set

1st iteration:

e6 != 0

e5 != 0

(e3%e2) != 0

(e1%e2) != 0

(#1%#2) != 0

```
--->  e1 (var_4) = #1
--->  e2 (var_8) = #2

--->  e3 (eax) = e1
--->  e4 (eax) = e3 / e2
--->  e5 (edx) = e3 % e2
--->  e6 (var_C) = e5
--->  e7 compare (e6, 0)
jump not equal
```

```
                    loc_100000f63:
0000000100000f63        mov        eax, dword [rbp+var_8]
0000000100000f66        mov        dword [rbp+var_4], eax
0000000100000f69        mov        eax, dword [rbp+var_C]
0000000100000f6c        mov        dword [rbp+var_8], eax
0000000100000f6f        jmp        loc_100000f4a
0000000100000f74        align      128
```

```
--->  e8  (eax) = e2
--->  e9  (var_4) = e8
--->  e10 (eax) = e6
--->  e11 (var_8) = e10
jump
```

GCD disassembled (using Hopper)

```
; ==== B E G I N N I N G   O F   P R O C E D U R E ====

       ; Variables:
       ;    var_4: -4
       ;    var_8: -8
       ;    var_C: -12
```

$(\#1\%\#2)$!= 0 ---> 2[nd] iteration:

$e_26 == 0$

...

$(e_21\%e_22) == 0$

$(e9\%e11) == 0$

$(e8\%e10) == 0$

$(e2\%e6) == 0$

$(\#2\%e5) == 0$

$(\#2\%(e3\%e2)) == 0$

$(\#2\%(\#1\%\#2)) == 0$

```
Symbolic Expression Set

--->  e1 (var_4) = #1
--->  e2 (var_8) = #2

--->  e3 (eax) = e1
--->  e4 (eax) = e3 / e2
--->  e5 (edx) = e3 % e2
--->  e6 (var_C) = e5
--->  e7 compare (e6, 0)
jump not equal

--->  e8  (eax) = e2
--->  e9  (var_4) = e8
--->  e10 (eax) = e6
--->  e11 (var_8) = e10
jump
```

GCD disassembled (using Hopper)

```
; ==== B E G I N N I N G   O F   P R O C E D U R E ====
        ; Variables:
        ;    var_4: -4
        ;    var_8: -8
        ;    var_C: -12
```

Symbolic Expression Set

(#1%#2) != 0 ---> 2^{nd} iteration:

e_26 == 0

...

(e_21%e_22) == 0

(e9%e11) == 0

(e8%e10) == 0

(e2%e6) == 0

(#2%e5) == 0

(#2%(e3%e2)) == 0

(#2%(#1%#2)) == 0

Solving

(#1%#2) != 0

and

(#2%(#1%#2)) == 0

gives

#1 = 2

#2 = 4

```
e2
e2

0)

---> e8  (eax) = e2
---> e9  (var_4) = e8
---> e10 (eax) = e6
---> e11 (var_8) = e10
jump
```

Very cool

- that this can be done on arbitrary binaries
- Check out:
  - shellphish http://shellphish.net/cgc/#tools
  - in particular:
  - angr http://angr.io/, https://github.com/angr/angr
  - driller https://github.com/shellphish/driller
  - tools for automatic patching and exploitation are also available
    - *outside the scope of this course, but contact me if interested in MSc project!*

TU Delft

KLEE

- An LLVM based concolic execution engine
  - LLVM is an Intermediate Representation, so not working directly on assembly, but one level higher, abstracting away some of the low-level implementations
- http://klee.github.io/docs/
- Very easy to use, simply declare parameters as symbolic, and run KLEE! (see Git)
- We use KLEE for the second lab assignment

A final note: from Wikipedia

```
void f(int x, int y) {
    int z = 2*y;
    if (x == 100000) {
        if (x < z) {
            assert(0); /* error */
        }
    }
}
```

"Simple random testing, trying random values of *x* and *y*, would require an impractically large number of tests to reproduce the failure."

Q. Why?

A final note: from Wikipedia

```
void f(int x, int y) {
    int z = 2*y;
    if (



    }
}
```

AFL finds it in milliseconds

Why?

“Simple random testing, trying random values of *x* and *y*, would require an impractically large number of tests to reproduce the failure.”

Q. Why?

AFL

- What AFL does exactly is not very clear, but see:
- http://lcamtuf.blogspot.nl/2014/08/binary-fuzzing-strategies-what-works.html
  - Walking bit/byte flips, simple arithmetic, ..
  - *But, form http://lcamtuf.coredump.cx/afl/technical_details.txt:*
    - *"AFL generally does not try to reason about the relationship between specific mutations and program states; the fuzzing steps are nominally blind, and are guided only by the evolutionary design of the input queue"*

AFL

- Wh
- h
  v

Challenge:

Find small code samples (+- 20 lines) that takes amazingly long to Fuzz/Test, but is quick to execute concolicly

and vice versa!

Example code is provided on Git

TU Delft