Automated Quantification of Software Side-Channel Vulnerabilities

Lucas Bang
Committee: Tevfik Bultan (Chair), Ben Hardekopf, Omer Egecioglu

Department of Computer Science
University of California, Santa Barbara

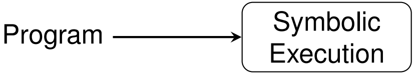
14 April 2016

Overview

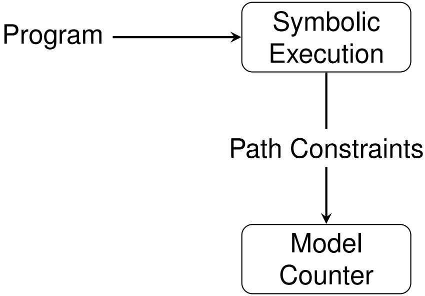
Overview

Program

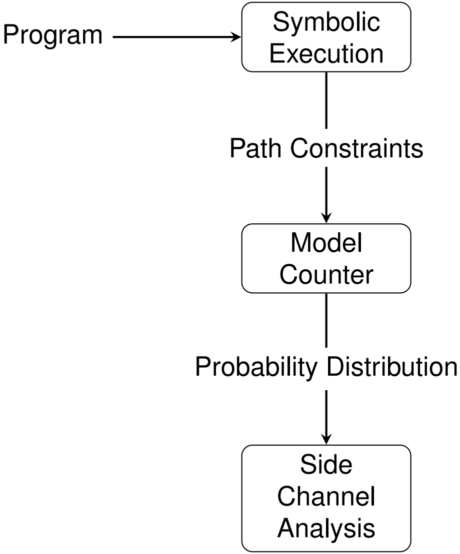
Overview



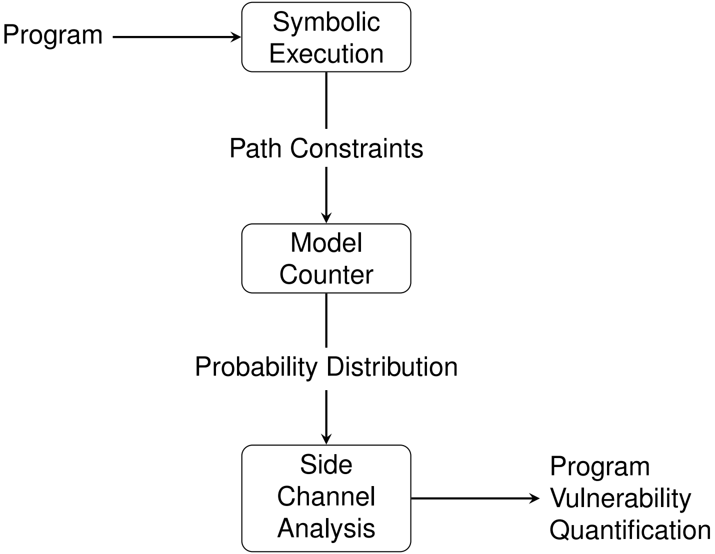
Overview


Program
Symbolic Execution
Path Constraints
Model Counter

Overview


Program
Symbolic Execution
Path Constraints
Model Counter
Probability Distribution
Side Channel Analysis

Overview


Program
Symbolic Execution
Path Constraints
Model Counter
Probability Distribution
Side Channel Analysis
Program Vulnerability Quantification

Outline

Symbolic Execution
- Software Verification
- Symbolic Execution
- Probabilistic Symbolic Execution
- SMT Solvers

Side Channel Analysis
- Background and Information Theory
- Via Probabalistic Symbolic Execution

Model Counting
- Boolean Logic
- Strings
- Linear Ineger Arithmetic

Outline

Software Verification

Goal: Given a program, determine if executions satisfy some property.

Software Verification

Goal: Given a program, determine if executions satisfy some property.

- Never divide by 0

Software Verification

Goal: Given a program, determine if executions satisfy some property.

- Never divide by 0
- Never throws array out of bounds exception

Software Verification

Goal: Given a program, determine if executions satisfy some property.

- Never divide by 0
- Never throws array out of bounds exception
- Never dereferences a null pointer

Software Verification

Goal: Given a program, determine if executions satisfy some property.

- Never divide by 0
- Never throws array out of bounds exception
- Never dereferences a null pointer
- Does not leak too much confidential information

Software Verification

Goal: Given a program, determine if executions satisfy some property.

- Never divide by 0
- Never throws array out of bounds exception
- Never dereferences a null pointer
- Does not leak too much confidential information
- Halts on all inputs

Software Verification

Goal: Given a program, determine if executions satisfy some property.

- Never divide by 0
- Never throws array out of bounds exception
- Never dereferences a null pointer
- Does not leak too much confidential information
- Halts on all inputs

Software Verification

Goal: Given a program, determine if executions satisfy some property.

- Never divide by 0
- Never throws array out of bounds exception
- Never dereferences a null pointer
- Does not leak too much confidential information
- Halts on all inputs

Software verification problem is undecidable!

Software Verification Techniques

Software Verification Techniques

Programs can have infinitely many behaviors.

Software Verification Techniques

Programs can have infinitely many behaviors.

Even simple programs can have exponentially many behaviors.

Software Verification Techniques

Programs can have infinitely many behaviors.

Even simple programs can have exponentially many behaviors.

Feasible Software verification techniques must deal with state space explosion.

Work on Software Verification

- Geldenhuys. Probabilistic symbolic execution. ISSTA 2012
- Bultan. Symbolic Model Checking of Infinite State Systems Using Presburger Arithmetic. CAV 1997
- Yu. Patching Vulnerabilities with Sanitization Synthesis. ICSE 2011
- Ball. Automatically Validating Temporal Safety Properties of Interfaces. SPIN 2001
- Biere. Symbolic Model Checking without BDDs. TACAS 1999
- Visser. Model Checking Programs. ASE 2003.
- Burch. Symbolic Model Checking: 10^{20} States and Beyond, LICS 1990
- Bryant, Graph-Based Algorithms for Boolean Function Manipulation, IEEE Trans. Computers. 1986
- Cadar. Symbolic execution for software testing in practice: preliminary assessment. ICSE 2011
- Cadar. Symbolic Execution for Software Testing: Three Decades Later. CACM 2013
- Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. POPL 1977.
- Cousot. Systematic Design of Program Analysis Frameworks. POPL 1979

Software Verification Tools

A small sample:

- Edmund Clarke. A Tool for Checking ANSI-C Programs. TACAS 2005.
- Holzmann. The Model Checker SPIN. IEEE Trans. Software Eng 1997.
- Musuvathi. CMC: A pragmatic approach to model checking real code. OSDI 2002.
- Yang. Using Model Checking to Find Serious File System Errors. OSDI 2004
- Ball. A decade of software model checking with SLAM. CACM 2011.
- Godefroid, et al. DART: Directed Automated Random Testing. PLDI 2005.
- Sen. CUTE: A Concolic Unit Testing Engine for C. ESEC/FSE 2005.
- SAGE: Whitebox Fuzzing for Security Testing. CACM 2012.

Outline

Symbolic Execution and Path Constraints

Basic Idea

- Represent program variables as symbolic variables:
  - $x_1 \mapsto X_1, x_2 \mapsto X_2, \ldots, x_n \mapsto X_n$
- Program executions are described by formulas over symbolic variables.
  - $f(X_1, X_2, \ldots, X_n)$
  - Path Constraints

Software Verification With Symbolic Execution

```
function f(x,y)
u = x - y
if(x > y)
  u = u + x
if(u < 0)
  assert false
exit
```

Software Verification With Symbolic Execution

```
function f(x,y)
u = x - y
if(x > y)
  u = u + x
if(u < 0)
  assert false
exit
```

Software Verification With Symbolic Execution

$\emptyset$

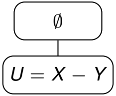
```
function f(x,y)
u = x - y
if(x > y)
  u = u + x
if(u < 0)
  assert false
exit
```

Software Verification With Symbolic Execution

∅

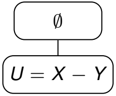
```
function f(x,y)
u = x - y
if(x > y)
  u = u + x
if(u < 0)
  assert false
exit
```

Software Verification With Symbolic Execution



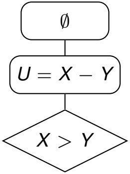

```
0. function f(x,y)
1. u = x - y
2. if(x > y)
3.   u = u + x
4. if(u < 0)
5.   assert false
6. exit
```

Software Verification With Symbolic Execution



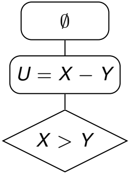

```
function f(x,y)
u = x - y
if(x > y)
  u = u + x
if(u < 0)
  assert false
exit
```

Software Verification With Symbolic Execution



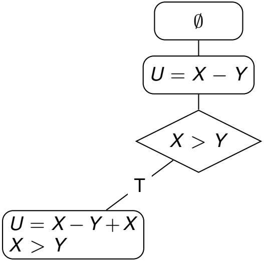

```
0. function f(x,y)
1. u = x - y
2. if(x > y)
3.   u = u + x
4. if(u < 0)
5.   assert false
6. exit
```

Software Verification With Symbolic Execution



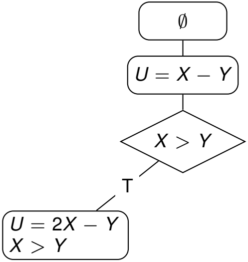

```
function f(x,y)
u = x - y
if(x > y)
  u = u + x
if(u < 0)
  assert false
exit
```

Software Verification With Symbolic Execution



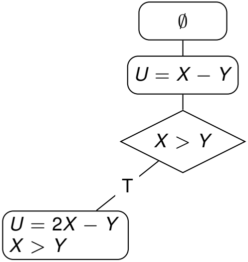

```
function f(x,y)
u = x - y
if(x > y)
  u = u + x
if(u < 0)
  assert false
exit
```

Software Verification With Symbolic Execution




```
function f(x,y)
u = x - y
if(x > y)
  u = u + x
if(u < 0)
  assert false
exit
```

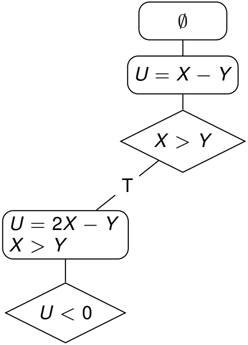
Software Verification With Symbolic Execution




```
function f(x,y)
u = x - y
if(x > y)
  u = u + x
if(u < 0)
  assert false
exit
```

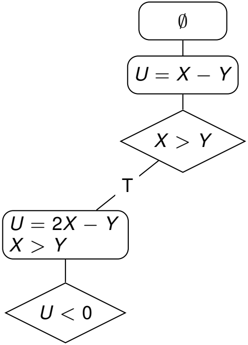
Software Verification With Symbolic Execution

```
0. function f(x,y)
1. u = x - y
2. if(x > y)
3.   u = u + x
4. if(u < 0)
5.   assert false
6. exit
```



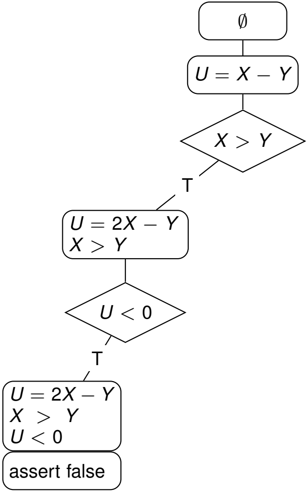
Software Verification With Symbolic Execution

```
0. function f(x,y)
1. u = x - y
2. if(x > y)
3.   u = u + x
4. if(u < 0)
5.   assert false
6. exit
```



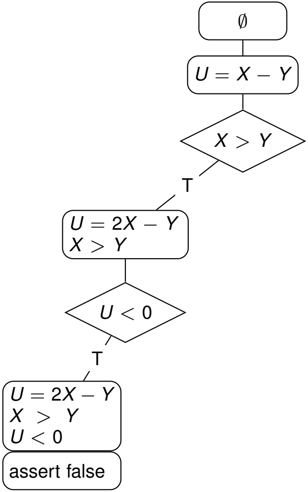
Software Verification With Symbolic Execution

```
0. function f(x,y)
1. u = x - y
2. if(x > y)
3.   u = u + x
4. if(u < 0)
5.   assert false
6. exit
```



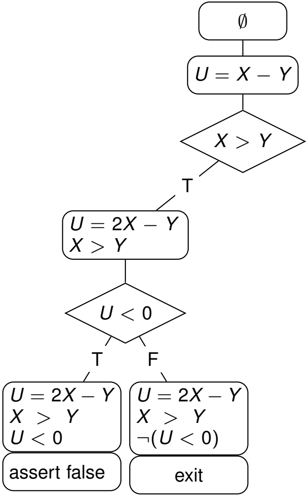
Software Verification With Symbolic Execution

```
0. function f(x,y)
1. u = x - y
2. if(x > y)
3.   u = u + x
4. if(u < 0)
5.   assert false
6. exit
```



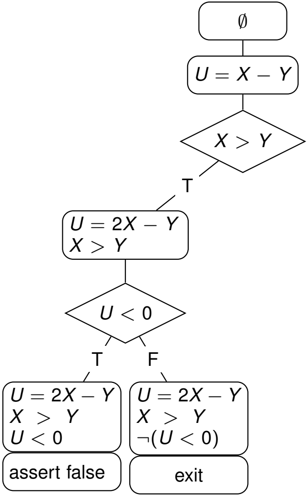
Software Verification With Symbolic Execution

```
0. function f(x,y)
1. u = x - y
2. if(x > y)
3.   u = u + x
4. if(u < 0)
5.   assert false
6. exit
```



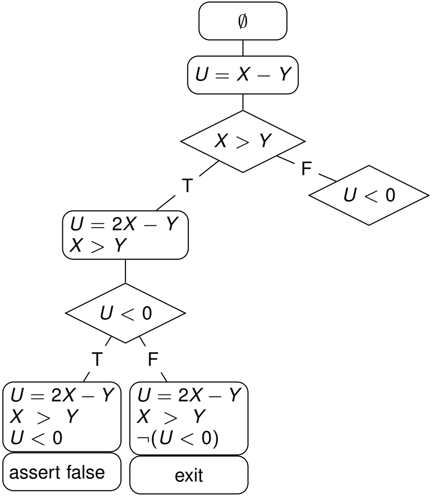
Software Verification With Symbolic Execution

```
function f(x,y)
u = x - y
if(x > y)
  u = u + x
if(u < 0)
  assert false
exit
```



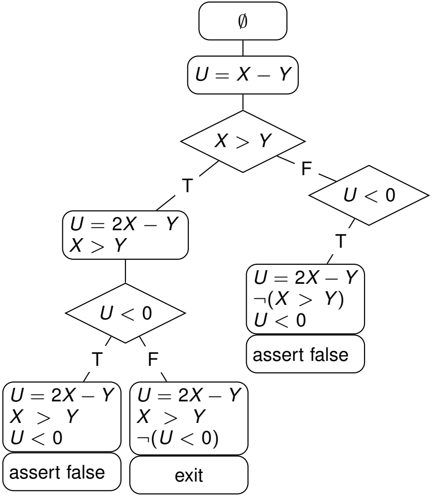
Software Verification With Symbolic Execution

```
0. function f(x,y)
1. u = x - y
2. if(x > y)
3.   u = u + x
4. if(u < 0)
5.   assert false
6. exit
```



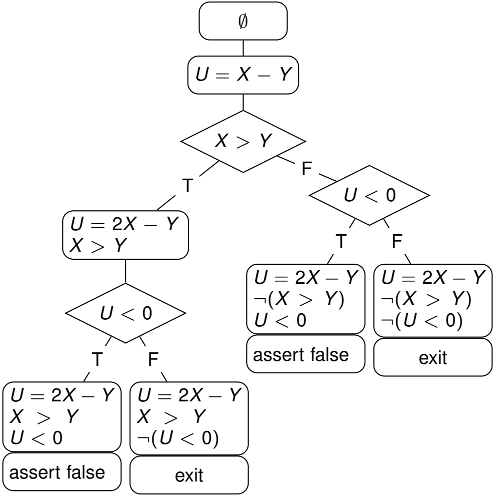
Software Verification With Symbolic Execution

```
0.  function f(x,y)
1.  u = x - y
2.  if(x > y)
3.    u = u + x
4.  if(u < 0)
5.    assert false
6.  exit
```



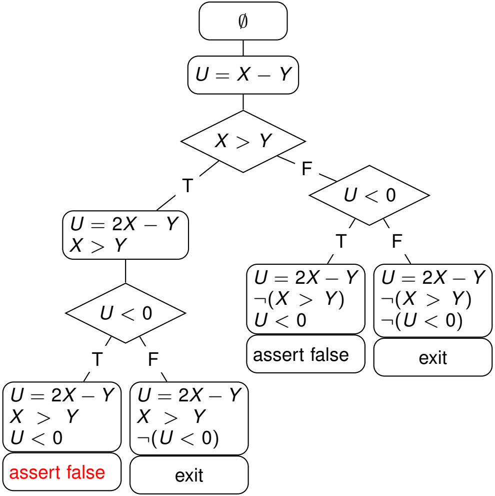
Software Verification With Symbolic Execution

```
0. function f(x,y)
1. u = x - y
2. if(x > y)
3.   u = u + x
4. if(u < 0)
5.   assert false
6. exit
```



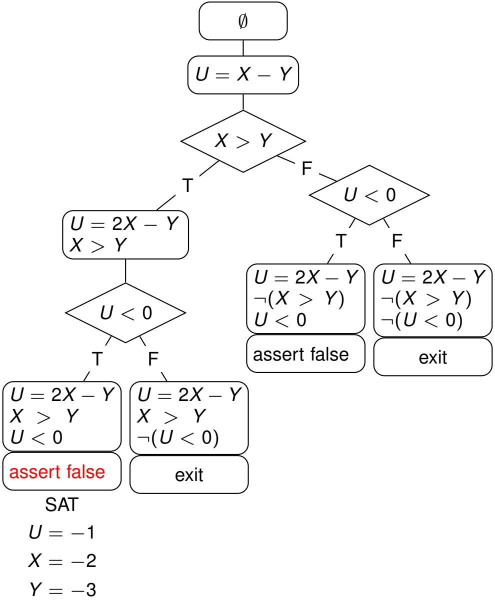
Software Verification With Symbolic Execution

```
0. function f(x,y)
1. u = x - y
2. if(x > y)
3.   u = u + x
4. if(u < 0)
5.   assert false
6. exit
```



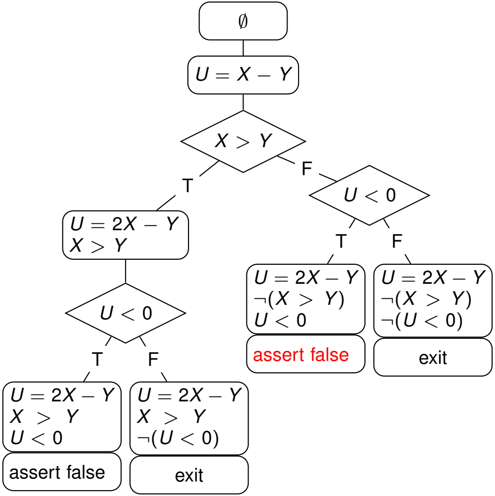
Software Verification With Symbolic Execution

```
0. function f(x,y)
1. u = x - y
2. if(x > y)
3.    u = u + x
4. if(u < 0)
5.    assert false
6. exit
```



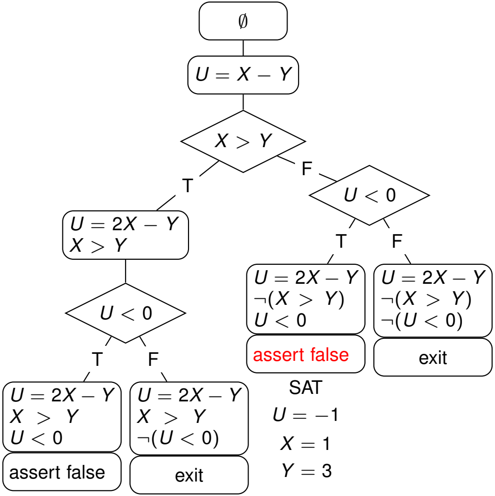
Software Verification With Symbolic Execution

```
0. function f(x,y)
1. u = x - y
2. if(x > y)
3.    u = u + x
4. if(u < 0)
5.    assert false
6. exit
```



Software Verification With Symbolic Execution

```
0. function f(x,y)
1. u = x - y
2. if(x > y)
3.   u = u + x
4. if(u < 0)
5.   assert false
6. exit
```



Outline

Probabilistic Symbolic Execution

Question

How likely is a certain program behavior?

Probabilistic Symbolic Execution

Question

How likely is a certain program behavior?

What is the the probability of a particular program execution path?

Probabilistic Symbolic Execution

Question

How likely is a certain program behavior?

What is the the probability of a particular program execution path?

Path Constraint Probability

Probabilistic Symbolic Execution

Question

How likely is a certain program behavior?

What is the the probability of a particular program execution path?

Path Constraint Probability

Let $|PC_i|$ be the number of solutions to PC_i.

Probabilistic Symbolic Execution

Question

How likely is a certain program behavior?

What is the the probability of a particular program execution path?

Path Constraint Probability

Let $|PC_i|$ be the number of solutions to PC_i.

Let $|D|$ be the size of the input domain D.

Probabilistic Symbolic Execution

Question

How likely is a certain program behavior?

What is the the probability of a particular program execution path?

Path Constraint Probability

Let $|PC_i|$ be the number of solutions to PC_i.

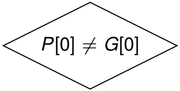
Let $|D|$ be the size of the input domain D.

Assuming D is uniformly distributed:

$$p(PC_i) = \frac{|PC_i|}{|D|}$$

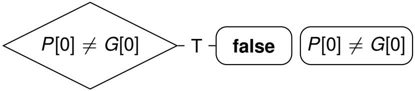
```
bool checkPIN(guess[])
for(i = 0; i < 4; i++)
 if(guess[i] != PIN[i])
  return false
return true
```

P: PIN, *G*: guess



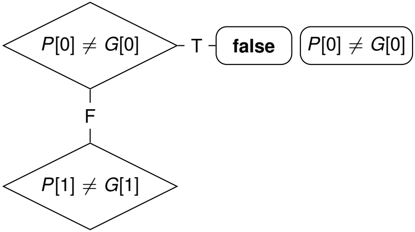

```
bool checkPIN(guess[])
for(i = 0; i < 4; i++)
 if(guess[i] != PIN[i])
  return false
return true
```

P: PIN, *G*: guess




```
bool checkPIN(guess[])
for(i = 0; i < 4; i++)
 if(guess[i] != PIN[i])
  return false
return true
```

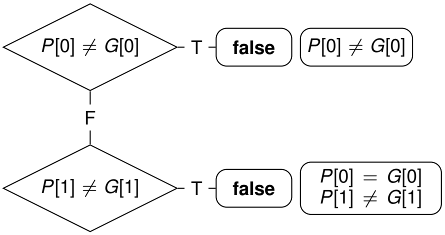
P: PIN, G: guess




```
bool checkPIN(guess[])
for(i = 0; i < 4; i++)
 if(guess[i] != PIN[i])
  return false
return true
```

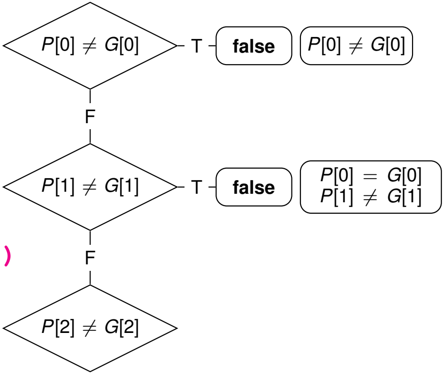
P: PIN, *G*: guess

```
bool checkPIN(guess[])
for(i = 0; i < 4; i++)
 if(guess[i] != PIN[i])
  return false
return true
```



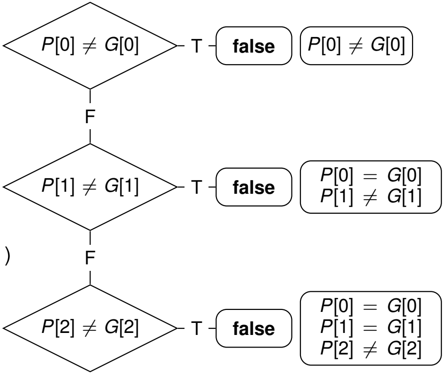
P: PIN, *G*: guess

```
bool checkPIN(guess[])
for(i = 0; i < 4; i++)
 if(guess[i] != PIN[i])
  return false
return true
```



P: PIN, G: guess

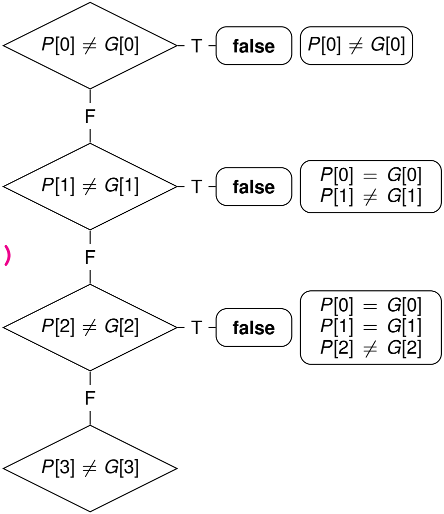
```
bool checkPIN(guess[])
for(i = 0; i < 4; i++)
 if(guess[i] != PIN[i])
  return false
return true
```



P: PIN, *G*: guess

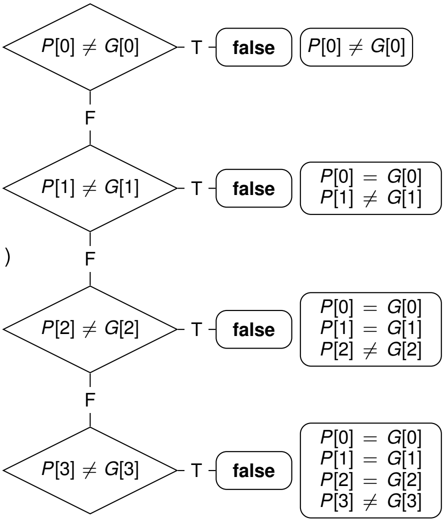
```
bool checkPIN(guess[])
for(i = 0; i < 4; i++)
 if(guess[i] != PIN[i])
  return false
return true
```

P: PIN, *G*: guess



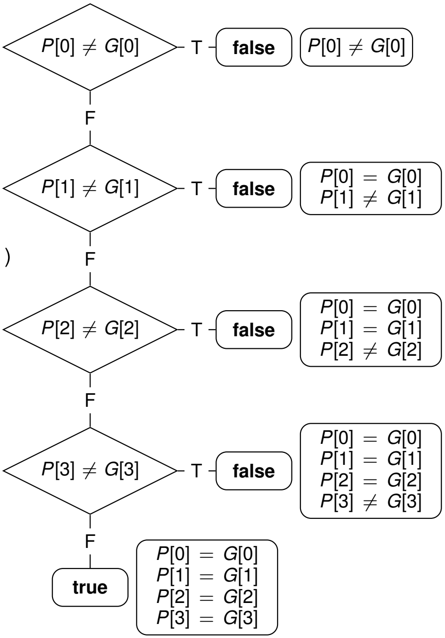
```
bool checkPIN(guess[])
for(i = 0; i < 4; i++)
 if(guess[i] != PIN[i])
  return false
return true
```

P: PIN, *G*: guess



```
bool checkPIN(guess[])
for(i = 0; i < 4; i++)
 if(guess[i] != PIN[i])
  return false
return true
```

P: PIN, G: guess



Probabilistic Symbolic Execution

i	0	1	2	3	4
PC_i	$P[0] \neq G[0]$	$P[0] = G[0]$ $P[1] \neq G[1]$	$P[0] = G[0]$ $P[1] = G[1]$ $P[2] \neq G[2]$	$P[0] = G[0]$ $P[1] = G[1]$ $P[2] = G[2]$ $P[3] \neq G[3]$	$P[0] = G[0]$ $P[1] = G[1]$ $P[2] = G[2]$ $P[3] = G[3]$
$\|PC_i\|$					
p_i					

Probabilistic Symbolic Execution

Assume binary 4 digit PIN. P has 4 bits, G has 4 bits. $|D| = 2^8 = 256$.

i	0	1	2	3	4
PC_i	$P[0] \neq G[0]$	$P[0] = G[0]$ $P[1] \neq G[1]$	$P[0] = G[0]$ $P[1] = G[1]$ $P[2] \neq G[2]$	$P[0] = G[0]$ $P[1] = G[1]$ $P[2] = G[2]$ $P[3] \neq G[3]$	$P[0] = G[0]$ $P[1] = G[1]$ $P[2] = G[2]$ $P[3] = G[3]$
$\|PC_i\|$					
p_i					

Probabilistic Symbolic Execution

Assume binary 4 digit PIN. P has 4 bits, G has 4 bits. $|D| = 2^8 = 256$.

i	0	1	2	3	4
PC_i	$P[0] \neq G[0]$	$P[0] = G[0]$ $P[1] \neq G[1]$	$P[0] = G[0]$ $P[1] = G[1]$ $P[2] \neq G[2]$	$P[0] = G[0]$ $P[1] = G[1]$ $P[2] = G[2]$ $P[3] \neq G[3]$	$P[0] = G[0]$ $P[1] = G[1]$ $P[2] = G[2]$ $P[3] = G[3]$
$\|PC_i\|$					
p_i					

$$p_i = \frac{|PC_i|}{|D|}$$

Probabilistic Symbolic Execution

Assume binary 4 digit PIN. P has 4 bits, G has 4 bits. $|D| = 2^8 = 256$.

i	0	1	2	3	4
PC_i	$P[0] \neq G[0]$	$P[0] = G[0]$ $P[1] \neq G[1]$	$P[0] = G[0]$ $P[1] = G[1]$ $P[2] \neq G[2]$	$P[0] = G[0]$ $P[1] = G[1]$ $P[2] = G[2]$ $P[3] \neq G[3]$	$P[0] = G[0]$ $P[1] = G[1]$ $P[2] = G[2]$ $P[3] = G[3]$
$\|PC_i\|$	?????				
p_i					

$$p_i = \frac{|PC_i|}{|D|}$$

Probabilistic Symbolic Execution

Assume binary 4 digit PIN. P has 4 bits, G has 4 bits. $|D| = 2^8 = 256$.

i	0	1	2	3	4
PC_i	$P[0] \neq G[0]$	$P[0] = G[0]$ $P[1] \neq G[1]$	$P[0] = G[0]$ $P[1] = G[1]$ $P[2] \neq G[2]$	$P[0] = G[0]$ $P[1] = G[1]$ $P[2] = G[2]$ $P[3] \neq G[3]$	$P[0] = G[0]$ $P[1] = G[1]$ $P[2] = G[2]$ $P[3] = G[3]$
$\|PC_i\|$	128				
p_i	?????				

$$p_i = \frac{|PC_i|}{|D|}$$

Probabilistic Symbolic Execution

Assume binary 4 digit PIN. P has 4 bits, G has 4 bits. $|D| = 2^8 = 256$.

i	0	1	2	3	4
PC_i	$P[0] \neq G[0]$	$P[0] = G[0]$ $P[1] \neq G[1]$	$P[0] = G[0]$ $P[1] = G[1]$ $P[2] \neq G[2]$	$P[0] = G[0]$ $P[1] = G[1]$ $P[2] = G[2]$ $P[3] \neq G[3]$	$P[0] = G[0]$ $P[1] = G[1]$ $P[2] = G[2]$ $P[3] = G[3]$
$\|PC_i\|$	128				
p_i	1/2				

$$p_i = \frac{|PC_i|}{|D|}$$

Probabilistic Symbolic Execution

Assume binary 4 digit PIN. P has 4 bits, G has 4 bits. $|D| = 2^8 = 256$.

i	0	1	2	3	4
PC_i	$P[0] \neq G[0]$	$P[0] = G[0]$ $P[1] \neq G[1]$	$P[0] = G[0]$ $P[1] = G[1]$ $P[2] \neq G[2]$	$P[0] = G[0]$ $P[1] = G[1]$ $P[2] = G[2]$ $P[3] \neq G[3]$	$P[0] = G[0]$ $P[1] = G[1]$ $P[2] = G[2]$ $P[3] = G[3]$
$\lvert PC_i \rvert$	128				
p_i	1/2				

$$p_i = \frac{|PC_i|}{|D|}$$

Probabilistic Symbolic Execution

Assume binary 4 digit PIN. P has 4 bits, G has 4 bits. $|D| = 2^8 = 256$.

i	0	1	2	3	4
PC_i	$P[0] \neq G[0]$	$P[0] = G[0]$ $P[1] \neq G[1]$	$P[0] = G[0]$ $P[1] = G[1]$ $P[2] \neq G[2]$	$P[0] = G[0]$ $P[1] = G[1]$ $P[2] = G[2]$ $P[3] \neq G[3]$	$P[0] = G[0]$ $P[1] = G[1]$ $P[2] = G[2]$ $P[3] = G[3]$
$\|PC_i\|$	128				
p_i	1/2				

$$p_i = \frac{|PC_i|}{|D|}$$

Probabilistic Symbolic Execution

Assume binary 4 digit PIN. P has 4 bits, G has 4 bits. $|D| = 2^8 = 256$.

i	0	1	2	3	4
PC_i	$P[0] \neq G[0]$	$P[0] = G[0]$ $P[1] \neq G[1]$	$P[0] = G[0]$ $P[1] = G[1]$ $P[2] \neq G[2]$	$P[0] = G[0]$ $P[1] = G[1]$ $P[2] = G[2]$ $P[3] \neq G[3]$	$P[0] = G[0]$ $P[1] = G[1]$ $P[2] = G[2]$ $P[3] = G[3]$
$\|PC_i\|$	128	?????			
p_i	1/2				

$$p_i = \frac{|PC_i|}{|D|}$$

Probabilistic Symbolic Execution

Assume binary 4 digit PIN. P has 4 bits, G has 4 bits. $|D| = 2^8 = 256$.

i	0	1	2	3	4
PC_i	$P[0] \neq G[0]$	$P[0] = G[0]$ $P[1] \neq G[1]$	$P[0] = G[0]$ $P[1] = G[1]$ $P[2] \neq G[2]$	$P[0] = G[0]$ $P[1] = G[1]$ $P[2] = G[2]$ $P[3] \neq G[3]$	$P[0] = G[0]$ $P[1] = G[1]$ $P[2] = G[2]$ $P[3] = G[3]$
$\lvert PC_i \rvert$	128	64			
p_i	1/2	?????			

$$p_i = \frac{|PC_i|}{|D|}$$

Probabilistic Symbolic Execution

Assume binary 4 digit PIN. P has 4 bits, G has 4 bits. $|D| = 2^8 = 256$.

i	0	1	2	3	4
PC_i	$P[0] \neq G[0]$	$P[0] = G[0]$ $P[1] \neq G[1]$	$P[0] = G[0]$ $P[1] = G[1]$ $P[2] \neq G[2]$	$P[0] = G[0]$ $P[1] = G[1]$ $P[2] = G[2]$ $P[3] \neq G[3]$	$P[0] = G[0]$ $P[1] = G[1]$ $P[2] = G[2]$ $P[3] = G[3]$
$\lvert PC_i \rvert$	128	64			
p_i	1/2	1/4			

$$p_i = \frac{|PC_i|}{|D|}$$

Probabilistic Symbolic Execution

Assume binary 4 digit PIN. P has 4 bits, G has 4 bits. $|D| = 2^8 = 256$.

i	0	1	2	3	4
PC_i	$P[0] \neq G[0]$	$P[0] = G[0]$ $P[1] \neq G[1]$	$P[0] = G[0]$ $P[1] = G[1]$ $P[2] \neq G[2]$	$P[0] = G[0]$ $P[1] = G[1]$ $P[2] = G[2]$ $P[3] \neq G[3]$	$P[0] = G[0]$ $P[1] = G[1]$ $P[2] = G[2]$ $P[3] = G[3]$
$\lvert PC_i \rvert$	128	64			
p_i	1/2	1/4			

$$p_i = \frac{|PC_i|}{|D|}$$

Probabilistic Symbolic Execution

Assume binary 4 digit PIN. P has 4 bits, G has 4 bits. $|D| = 2^8 = 256$.

i	0	1	2	3	4
PC_i	$P[0] \neq G[0]$	$P[0] = G[0]$ $P[1] \neq G[1]$	$P[0] = G[0]$ $P[1] = G[1]$ $P[2] \neq G[2]$	$P[0] = G[0]$ $P[1] = G[1]$ $P[2] = G[2]$ $P[3] \neq G[3]$	$P[0] = G[0]$ $P[1] = G[1]$ $P[2] = G[2]$ $P[3] = G[3]$
$\|PC_i\|$	128	64	32		
p_i	1/2	1/4	1/8		

$$p_i = \frac{|PC_i|}{|D|}$$

Probabilistic Symbolic Execution

Assume binary 4 digit PIN. P has 4 bits, G has 4 bits. $|D| = 2^8 = 256$.

i	0	1	2	3	4
PC_i	$P[0] \neq G[0]$	$P[0] = G[0]$ $P[1] \neq G[1]$	$P[0] = G[0]$ $P[1] = G[1]$ $P[2] \neq G[2]$	$P[0] = G[0]$ $P[1] = G[1]$ $P[2] = G[2]$ $P[3] \neq G[3]$	$P[0] = G[0]$ $P[1] = G[1]$ $P[2] = G[2]$ $P[3] = G[3]$
$\|PC_i\|$	128	64	32	16	
p_i	1/2	1/4	1/8	1/16	

$$p_i = \frac{|PC_i|}{|D|}$$

Probabilistic Symbolic Execution

Assume binary 4 digit PIN. P has 4 bits, G has 4 bits. $|D| = 2^8 = 256$.

i	0	1	2	3	4
PC_i	$P[0] \neq G[0]$	$P[0] = G[0]$ $P[1] \neq G[1]$	$P[0] = G[0]$ $P[1] = G[1]$ $P[2] \neq G[2]$	$P[0] = G[0]$ $P[1] = G[1]$ $P[2] = G[2]$ $P[3] \neq G[3]$	$P[0] = G[0]$ $P[1] = G[1]$ $P[2] = G[2]$ $P[3] = G[3]$
$\|PC_i\|$	128	64	32	16	16
p_i	1/2	1/4	1/8	1/16	1/16

$$p_i = \frac{|PC_i|}{|D|}$$

Probabilistic Symbolic Execution

Assume binary 4 digit PIN. P has 4 bits, G has 4 bits. $|D| = 2^8 = 256$.

i	0	1	2	3	4
PC_i	$P[0] \neq G[0]$	$P[0] = G[0]$ $P[1] \neq G[1]$	$P[0] = G[0]$ $P[1] = G[1]$ $P[2] \neq G[2]$	$P[0] = G[0]$ $P[1] = G[1]$ $P[2] = G[2]$ $P[3] \neq G[3]$	$P[0] = G[0]$ $P[1] = G[1]$ $P[2] = G[2]$ $P[3] = G[3]$
$\|PC_i\|$	128	64	32	16	16
p_i	1/2	1/4	1/8	1/16	1/16

$$p_i = \frac{|PC_i|}{|D|}$$

Probabilistic Symbolic Execution

Assume binary 4 digit PIN. P has 4 bits, G has 4 bits. $|D| = 2^8 = 256$.

i	0	1	2	3	4
PC_i	$P[0] \neq G[0]$	$P[0] = G[0]$ $P[1] \neq G[1]$	$P[0] = G[0]$ $P[1] = G[1]$ $P[2] \neq G[2]$	$P[0] = G[0]$ $P[1] = G[1]$ $P[2] = G[2]$ $P[3] \neq G[3]$	$P[0] = G[0]$ $P[1] = G[1]$ $P[2] = G[2]$ $P[3] = G[3]$
$\|PC_i\|$	128	64	32	16	16
p_i	1/2	1/4	1/8	1/16	1/16

$$p_i = \frac{|PC_i|}{|D|}$$

A measure of program vulnerability

Probability that an adversary can guess a prefix of length i in 1 guess is given by p_i.

Outline

Satisfiability Modulo Theories (SMT) Solvers

Problem: how to solve path constraints?

Satisfiability Modulo Theories (SMT) Solvers

Problem: how to solve path constraints?

Satisfiability Modulo Theories (SMT) Solvers

Satisfiability Modulo Theories (SMT) Solvers

Problem: how to solve path constraints?

Satisfiability Modulo Theories (SMT) Solvers

SMT solvers determine the satisfiability of formulas from combinations of theories including:

- Linear Integer Arithmetic (LIA)
- Strings
- Bitvectors
- Arrays
- Uninterpreted Functions

Satisfiability Modulo Theories (SMT) Solvers

Problem: how to solve path constraints?

Satisfiability Modulo Theories (SMT) Solvers

SMT solvers determine the satisfiability of formulas from combinations of theories including:

- Linear Integer Arithmetic (LIA)
- Strings
- Bitvectors
- Arrays
- Uninterpreted Functions

Existing SMT solvers include: Z3, CVC4, MathSAT, ...

Work in SMT Solvers

- Birnbaum. The good old Davis-Putnam procedure helps counting models. JAIR 1999
- Vijay Ganesh. Decision Procedures for Bit-Vectors, Arrays and Integers(PhD. Thesis) 2007.
- Jha. Engineering an efficient SMT solver for bit-vector arithmetic. CAV 2009.
- Bryant, S. M. German, and M. N. Velev, Microprocessor Verification Using Efficient Decision Procedures for a Logic of Equality with Uninterpreted Functions. ATRM 1999.
- Davis. A Computing Procedure for Quantification Theory. JACM 1960.
- Davis. A Machine Program for Theorem-Proving. CACM 1962.
- Kroening. Decision Procedures - an algorithmic point of view. TCS 2008
- Deters. A tour of CVC4: How it works, and how to use it. FMCAD 2014.
- Barrett. CVC4. CAV 2011
- De Moura. Z3: an efficient SMT solver. TACAS 2008

Satisfiability Modulo Theories (SMT) Solvers

Davis-Putnam-Logemann-Loveland (DPLL) Algorithm

A decision procedure for satisfiability of Boolean formulas in conjunctive normal form (CNF-SAT).

Satisfiability Modulo Theories (SMT) Solvers

Davis-Putnam-Logemann-Loveland (DPLL) Algorithm

A decision procedure for satisfiability of Boolean formulas in conjunctive normal form (CNF-SAT).

This is **the core** algorithm used in SMT solvers.

Davis-Putnam-Logemann-Loveland (DPLL) Algorithm

```
Function : DPLL(φ)
Input    : CNF formula φ over n variables
Output   : true or false, the satisfiability of F
begin
  | UnitPropagate(φ)
  | if φ has false clause then return false
  | if all clauses of φ satisfied then return true
  | x ← SelectBranchVariable(φ)
  | return DPLL(φ[x ↦ true]) ∨ DPLL(φ[x ↦ false])
end
```

Davis-Putnam-Logemann-Loveland (DPLL) Algorithm

Function : DPLL(ϕ)
Input : **CNF formula ϕ over n variables**
Output : true or false, the satisfiability of F
begin
  UnitPropagate(ϕ)
  if ϕ has false clause **then return** false
  if all clauses of ϕ satisfied **then return** true
  x $\leftarrow$ SelectBranchVariable(ϕ)
  return DPLL($\phi[x \mapsto \mathit{true}]$) $\vee$ DPLL($\phi[x \mapsto \mathit{false}]$)
end

Davis-Putnam-Logemann-Loveland (DPLL) Algorithm

Function : DPLL(ϕ)
Input : CNF formula ϕ over n variables
Output : true or false, the satisfiability of F
begin
  UnitPropagate(ϕ)
  if ϕ has false clause **then return** false
  if all clauses of ϕ satisfied **then return** true
  x $\leftarrow$ SelectBranchVariable(ϕ)
  return DPLL($\phi[x \mapsto \mathit{true}]$) $\vee$ DPLL($\phi[x \mapsto \mathit{false}]$)
end

Davis-Putnam-Logemann-Loveland (DPLL) Algorithm

Function : DPLL(ϕ)
Input : CNF formula ϕ over *n* variables
Output : true or false, the satisfiability of F
begin
  UnitPropagate(ϕ)
  if ϕ has false clause **then return** false
  if all clauses of ϕ satisfied **then return** true
  x $\leftarrow$ SelectBranchVariable(ϕ)
  return DPLL($\phi[x \mapsto \mathit{true}]$) $\vee$ DPLL($\phi[x \mapsto \mathit{false}]$)
end

Davis-Putnam-Logemann-Loveland (DPLL) Algorithm

Function : DPLL(ϕ)
Input : CNF formula ϕ over *n* variables
Output : true or false, the satisfiability of F
begin
    UnitPropagate(ϕ)
    if ϕ has false clause then return false
    if all clauses of ϕ satisfied **then return** true
    x $\leftarrow$ SelectBranchVariable(ϕ)
    return DPLL($\phi[x \mapsto \textit{true}]$) $\vee$ DPLL($\phi[x \mapsto \textit{false}]$)
end

Davis-Putnam-Logemann-Loveland (DPLL) Algorithm

Function : DPLL(ϕ)
Input : CNF formula ϕ over *n* variables
Output : true or false, the satisfiability of F
begin
    UnitPropagate(ϕ)
    if ϕ has false clause **then return** false
    if all clauses of ϕ satisfied then return true
    x $\leftarrow$ SelectBranchVariable(ϕ)
    return DPLL($\phi[x \mapsto \mathit{true}]$) $\vee$ DPLL($\phi[x \mapsto \mathit{false}]$)
end

Davis-Putnam-Logemann-Loveland (DPLL) Algorithm

Function : DPLL(ϕ)
Input : CNF formula ϕ over *n* variables
Output : true or false, the satisfiability of F
begin
  UnitPropagate(ϕ)
  if ϕ has false clause **then return** false
  if all clauses of ϕ satisfied **then return** true
  x $\leftarrow$ **SelectBranchVariable(ϕ)**
  return DPLL($\phi[x \mapsto true]$) $\vee$ DPLL($\phi[x \mapsto false]$)
end

Davis-Putnam-Logemann-Loveland (DPLL) Algorithm

Function : DPLL(ϕ)
Input : CNF formula ϕ over n variables
Output : true or false, the satisfiability of F
begin
    UnitPropagate(ϕ)
    if ϕ has false clause **then return** false
    if all clauses of ϕ satisfied **then return** true
    x $\leftarrow$ SelectBranchVariable(ϕ)
    return DPLL($\phi[x \mapsto \mathit{true}]$) $\vee$ **DPLL**($\phi[x \mapsto \mathit{false}]$)
end

Satisfiability Modulo Theories (SMT) Solvers

DPLL uses **Unit Propagation**.

$$\phi = \{x \vee y \neg x \vee z, z \vee w, x, y \vee v\}$$

Satisfiability Modulo Theories (SMT) Solvers

DPLL uses **Unit Propagation**.

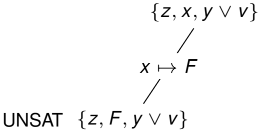
$$\phi = \{x \vee y \neg x \vee z, z \vee w, x, y \vee v\}$$

$$\phi' = \{z, x, y \vee v\}$$

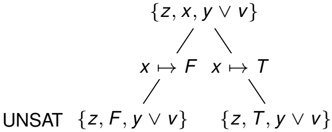
DPLL Execution Example

$$\{z, x, y \vee v\}$$

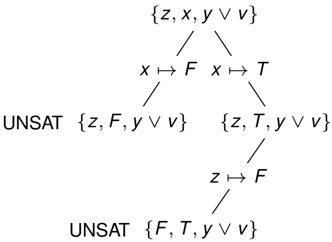
DPLL Execution Example



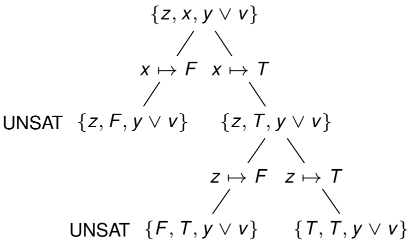
DPLL Execution Example



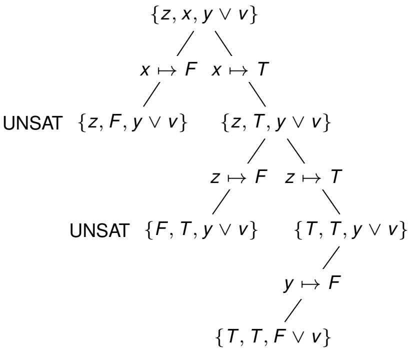
DPLL Execution Example



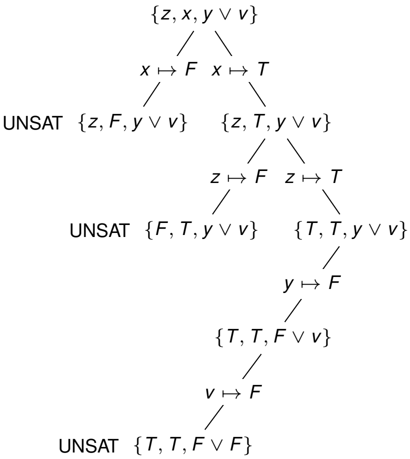
DPLL Execution Example



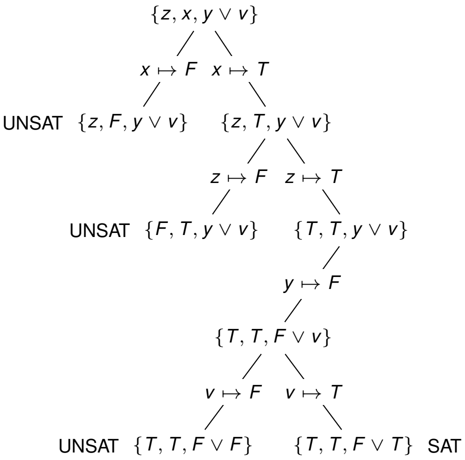
DPLL Execution Example



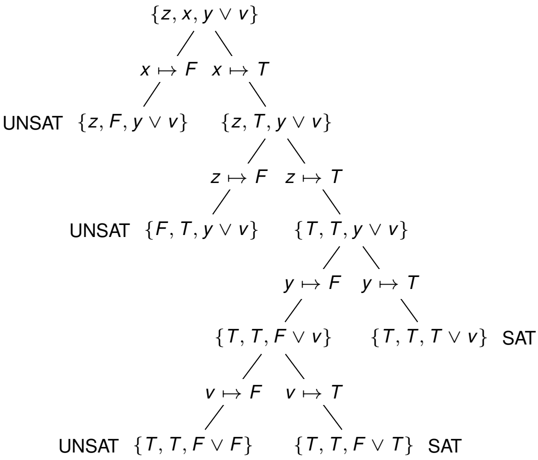
DPLL Execution Example



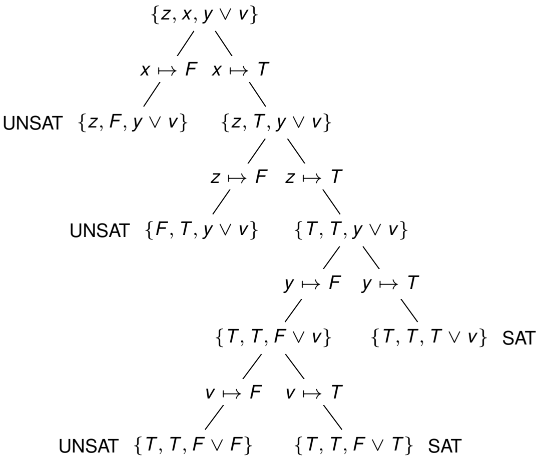
DPLL Execution Example



DPLL Execution Example



DPLL Execution Example



Result: ϕ is satisfiable.

Software Verification With Symbolic Execution

Symbolic Execution

Software Verification With Symbolic Execution

Symbolic Execution

- Summarizes program executions with path constraints.

Software Verification With Symbolic Execution

Symbolic Execution

- Summarizes program executions with path constraints.
- Relies on efficient solution of PCs - use SMT solvers.

Software Verification With Symbolic Execution

Symbolic Execution

- Summarizes program executions with path constraints.
- Relies on efficient solution of PCs - use SMT solvers.
- Warning: very effective, but unsound and can be expensive.

Software Verification With Symbolic Execution

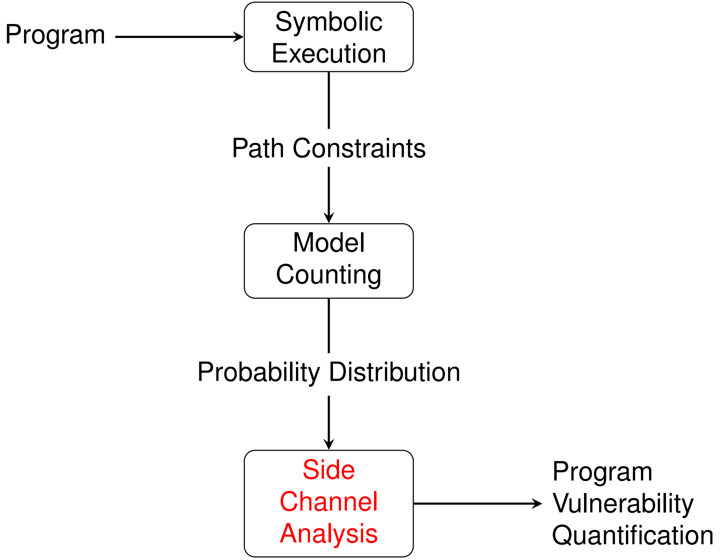
Symbolic Execution

- Summarizes program executions with path constraints.
- Relies on efficient solution of PCs - use SMT solvers.
- Warning: very effective, but unsound and can be expensive.

Variants of Symbolic Execution

- Standard
  - Cadar. Symbolic execution for software testing in practice: preliminary assessment. ICSE 2011
  - Cadar. Symbolic Execution for Software Testing: Three Decades Later. CACM 2013
- Probabilistic
  - Geldenhuys. Probabilistic symbolic execution. ISSTA 2012

Overview



Program
Symbolic Execution
Path Constraints
Model Counting
Probability Distribution
Side Channel Analysis
Program Vulnerability Quantification

Outline

What is a side channel?

How's the weather?

What is a side channel?

How's the weather?

Direct Channel: Go outside and look up.

What is a side channel?

How's the weather?

Direct Channel: Go outside and look up.

But, I'm too busy working on my MAE.

What is a side channel?

How's the weather?

Direct Channel: Go outside and look up.

But, I'm too busy working on my MAE.

Side Channel: Did Bo ride his bike today?

What is a side channel?

How's the weather?

Direct Channel: Go outside and look up.

But, I'm too busy working on my MAE.

Side Channel: Did Bo ride his bike today?

Learn some information through an indirect observation.

Observe Bo instead of the weather.

Side Channel Analysis

As a software verification problem

Side Channel Analysis

As a software verification problem

Verify that a program does not leak “too much” confidential information to an adversary who can observe:

- Computation time
- Power usage
- Memory allocations
- Network packet size
- Keystroke time

Side Channel Analysis

First considered at the hardware level.

```
int modPow(int num, int privatekey, int publickey)
  int s = 1, y = num, result = 0;
  while (privatekey > 0)
    if (privatekey % 2 == 1)
      result = (s * y) % publickey;
    else
      result = s;
    s = (result * result) % publickey;
    privatekey /= 2;
  return result;
```

Side Channel Analysis

First considered at the hardware level.

```
int modPow(int num, int privatekey, int publickey)
  int s = 1, y = num, result = 0;
  while (privatekey > 0)
    if (privatekey % 2 == 1)
      result = (s * y) % publickey;
    else
      result = s;
    s = (result * result) % publickey;
    privatekey /= 2;
  return result;
```

Side Channel Analysis

First considered at the hardware level.

```
int modPow(int num, int privatekey, int publickey)
  int s = 1, y = num, result = 0;
  while (privatekey > 0)
    if (privatekey % 2 == 1)
      result = (s * y) % publickey;
    else
      result = s;
    s = (result * result) % publickey;
    privatekey /= 2;
  return result;
```

Side Channel Analysis

First considered at the hardware level.

```
int modPow(int num, int privatekey, int publickey)
  int s = 1, y = num, result = 0;
  while (privatekey > 0)
    if (privatekey % 2 == 1)
      result = (s * y) % publickey;
    else
      result = s;
    s = (result * result) % publickey;
    privatekey /= 2;
  return result;
```

Side Channel Analysis

First considered at the hardware level.

```
int modPow(int num, int privatekey, int publickey)
  int s = 1, y = num, result = 0;
  while (privatekey > 0)
    if (privatekey % 2 == 1)
      result = (s * y) % publickey;
    else
      result = s;
    s = (result * result) % publickey;
    privatekey /= 2;
  return result;
```

Side Channel Analysis

A lot of research interest

- Geoffrey Smith. On the Foundations of Quantitative Information Flow. FOSSACS 2009
- Pasquale Malacaria. Assessing security threats of looping constructs. POPL 2007
- David Clark. A static analysis for quantifying information flow in a simple imperative language. JCS (2007)
- Jonathan Heusser. Quantifying information leaks in software. ACSAC 2010: 261-269
- Quoc-Sang Phan. Symbolic quantitative information flow. ACM SIGSOFT SEN 2012
- Quoc-Sang Phan. Quantifying information leaks using reliability analysis. SPIN 2014
- Stephen McCamant. QIF as network flow capacity. PLDI 2008
- Stephen McCamant. QIF tracking for C and related languages. MIT CSAIL 2006
- Michael Backes. Automatic Discovery and Quantification of Information Leaks. SSP 2009
- Shuo Chen. Side-Channel Leaks in Web Applications: A Reality Today, a Challenge Tomorrow. IEEE SSP 2010
- Goran Doychev. CacheAudit: A Tool for the Static Analysis of Cache Side Channels. USENIX Security 2013
- Boris Kopf. Automatically deriving information-theoretic bounds for adaptive side-channel attacks. JCS 2011
- Dawn Xiaodong Song. Timing analysis of keystrokes and timing attacks on SSH. USENIX Security SSYM 2001
- Thomas S. Messerges. Power Analysis Attacks of Modular Exponentiation in Smartcards, CHES 2002

Quantitative Information Flow

A Concepetual Framework

- Let C be a program with inputs $I \in \mathcal{I}$ and observables $O \in \mathcal{O}$
- C is deterministic.
- $\mathcal{I} \sim U(min, max)$

Quantitative Information Flow

A Concepetual Framework

- Let C be a program with inputs $I \in \mathcal{I}$ and observables $O \in \mathcal{O}$
- C is deterministic.
- $\mathcal{I} \sim U(min, max)$

Then there exists a function $f : \mathcal{I} \to \mathcal{O}$ such that

- f induces an equivalence relation on $\mathcal{I}$
- $I_1 \sim I_2$ iff $f(I_1) = f(I_2)$

Quantitative Information Flow

A Concepetual Framework

- Let C be a program with inputs $I \in \mathcal{I}$ and observables $O \in \mathcal{O}$
- C is deterministic.
- $\mathcal{I} \sim U(min, max)$

Then there exists a function $f : \mathcal{I} \to \mathcal{O}$ such that

- f induces an equivalence relation on $\mathcal{I}$
- $I_1 \sim I_2$ iff $f(I_1) = f(I_2)$

Example: C outputs last 4 digits of $CC\#$

Quantitative Information Flow

A Concepetual Framework

- Let C be a program with inputs $I \in \mathcal{I}$ and observables $O \in \mathcal{O}$
- C is deterministic.
- $\mathcal{I} \sim U(min, max)$

Then there exists a function $f : \mathcal{I} \to \mathcal{O}$ such that

- f induces an equivalence relation on $\mathcal{I}$
- $I_1 \sim I_2$ iff $f(I_1) = f(I_2)$

Example: C outputs last 4 digits of $CC\#$

- $f(n) = n \mod 10000$

Quantitative Information Flow

A Concepetual Framework

- Let C be a program with inputs $I \in \mathcal{I}$ and observables $O \in \mathcal{O}$
- C is deterministic.
- $\mathcal{I} \sim U(min, max)$

Then there exists a function $f : \mathcal{I} \to \mathcal{O}$ such that

- f induces an equivalence relation on $\mathcal{I}$
- $I_1 \sim I_2$ iff $f(I_1) = f(I_2)$

Example: C outputs last 4 digits of $CC\#$

- $f(n) = n \mod 10000$
- $f(0000\ 0000\ 0000\ 6789) = 6789$

Quantitative Information Flow

A Concepetual Framework

- Let C be a program with inputs $I \in \mathcal{I}$ and observables $O \in \mathcal{O}$
- C is deterministic.
- $\mathcal{I} \sim U(min, max)$

Then there exists a function $f : \mathcal{I} \to \mathcal{O}$ such that

- f induces an equivalence relation on $\mathcal{I}$
- $I_1 \sim I_2$ iff $f(I_1) = f(I_2)$

Example: C outputs last 4 digits of $CC\#$

- $f(n) = n \mod 10000$
- $f(0000\ 0000\ 0000\ 6789) = 6789 = f(1111\ 1111\ 1111\ 6789)$

Quantitative Information Flow

A Concepetual Framework

- Let C be a program with inputs $I \in \mathcal{I}$ and observables $O \in \mathcal{O}$
- C is deterministic.
- $\mathcal{I} \sim U(min, max)$

Then there exists a function $f : \mathcal{I} \to \mathcal{O}$ such that

- f induces an equivalence relation on $\mathcal{I}$
- $I_1 \sim I_2$ iff $f(I_1) = f(I_2)$

Example: C outputs last 4 digits of $CC\#$

- $f(n) = n \mod 10000$
- $f(0000\ 0000\ 0000\ 6789) = 6789 = f(1111\ 1111\ 1111\ 6789)$
- $0000\ 0000\ 0000\ 6789 \sim 1111\ 1111\ 1111\ 6789$

Information Gain

Adversarial Model

A malicious adversary can see the observables, O.

This tells adversary which equivalence class I belonged to.

That is, the adversary gains information about what the input was.

Information Gain

Adversarial Model

A malicious adversary can see the observables, O.

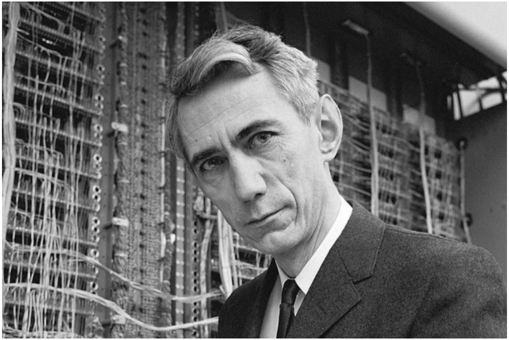
This tells adversary which equivalence class I belonged to.

That is, the adversary gains information about what the input was.

How much can the adversary learn?

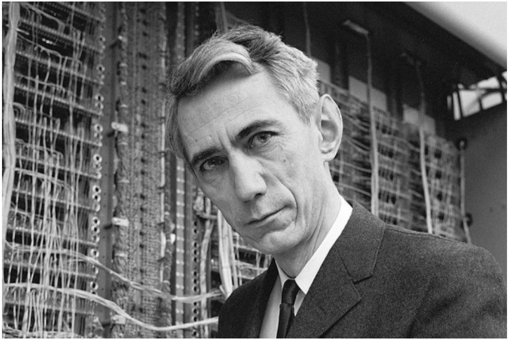
Quantify using information theory.

Information Theory



Claude Shannon

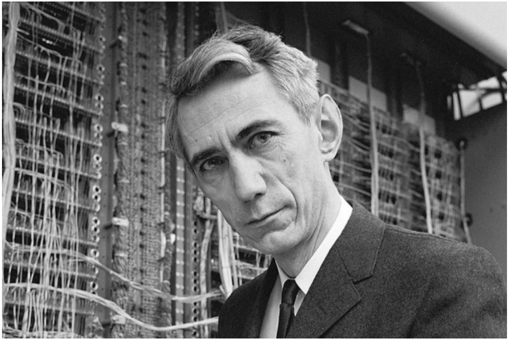
Information Theory



Claude Shannon

"A Theory of Communication". Bell System Technical Journal, 1948.

Information Theory



Claude Shannon

"A Theory of Communication". Bell System Technical Journal, 1948.

$$H = \sum p_i \log \frac{1}{p_i}$$

Information Theory Intuition

Logarithm gives the necessary number of bits

$$S = \{0, 1, 2, 3, \ldots, 254, 255\}$$

Information Theory Intuition

Logarithm gives the necessary number of bits

$$S = \{0, 1, 2, 3, \ldots, 254, 255\}$$

How many bits needed to distingish $x, y \in S$?

Information Theory Intuition

Logarithm gives the necessary number of bits

$$S = \{0, 1, 2, 3, \ldots, 254, 255\}$$

How many bits needed to distingish $x, y \in S$? $\log_2(256) = 8$

Information Theory Intuition

Logarithm gives the necessary number of bits

$$S = \{0, 1, 2, 3, \ldots, 254, 255\}$$

How many bits needed to distingish $x, y \in S$? $\log_2(256) = 8$

What about a partition?

Information Theory Intuition

Logarithm gives the necessary number of bits

$$S = \{0, 1, 2, 3, \ldots, 254, 255\}$$

How many bits needed to distingish $x, y \in S$? $\log_2(256) = 8$

What about a partition?

$S_0 = \{0, \ldots, 31\},$

Information Theory Intuition

Logarithm gives the necessary number of bits

$$S = \{0, 1, 2, 3, \ldots, 254, 255\}$$

How many bits needed to distingish $x, y \in S$? $\log_2(256) = 8$

What about a partition?

$$S_0 = \{0, \ldots, 31\}, S_1 = \{32, \ldots, 63\},$$

Information Theory Intuition

Logarithm gives the necessary number of bits

$$S = \{0, 1, 2, 3, \ldots, 254, 255\}$$

How many bits needed to distingish $x, y \in S$? $\log_2(256) = 8$

What about a partition?

$$S_0 = \{0, \ldots, 31\}, S_1 = \{32, \ldots, 63\}, \ldots, S_8 = \{224, \ldots, 255\}$$

Information Theory Intuition

Logarithm gives the necessary number of bits

$$S = \{0, 1, 2, 3, \ldots, 254, 255\}$$

How many bits needed to distingish $x, y \in S$? $\log_2(256) = 8$

What about a partition?

$$S_0 = \{0, \ldots, 31\}, S_1 = \{32, \ldots, 63\}, \ldots, S_8 = \{224, \ldots, 255\}$$

How many bits needed to distinguish $S_i, S_j \subseteq S$?

Information Theory Intuition

Logarithm gives the necessary number of bits

$$S = \{0, 1, 2, 3, \ldots, 254, 255\}$$

How many bits needed to distingish $x, y \in S$? $\log_2(256) = 8$

What about a partition?

$$S_0 = \{0, \ldots, 31\}, S_1 = \{32, \ldots, 63\}, \ldots, S_8 = \{224, \ldots, 255\}$$

How many bits needed to distinguish $S_i, S_j \subseteq S$?

$$\log \frac{256}{32} = \log 8 = 3$$

Information Theory Intuition

Logarithm gives the necessary number of bits

$$S = \{0, 1, 2, 3, \ldots, 254, 255\}$$

How many bits needed to distingish $x, y \in S$? $\log_2(256) = 8$

What about a partition?

$$S_0 = \{0, \ldots, 31\}, S_1 = \{32, \ldots, 63\}, \ldots, S_8 = \{224, \ldots, 255\}$$

How many bits needed to distinguish $S_i, S_j \subseteq S$?

$$\log \frac{256}{32} = \log 8 = 3$$

$$\log \frac{256}{32}$$

Information Theory Intuition

Logarithm gives the necessary number of bits

$$S = \{0, 1, 2, 3, \ldots, 254, 255\}$$

How many bits needed to distingish $x, y \in S$? $\log_2(256) = 8$

What about a partition?

$$S_0 = \{0, \ldots, 31\}, S_1 = \{32, \ldots, 63\}, \ldots, S_8 = \{224, \ldots, 255\}$$

How many bits needed to distinguish $S_i, S_j \subseteq S$?

$$\log \frac{256}{32} = \log 8 = 3$$

$$\log \frac{256}{32} = \log \left(\frac{32}{256}\right)^{-1}$$

Information Theory Intuition

Logarithm gives the necessary number of bits

$$S = \{0, 1, 2, 3, \ldots, 254, 255\}$$

How many bits needed to distingish $x, y \in S$? $\log_2(256) = 8$

What about a partition?

$$S_0 = \{0, \ldots, 31\}, S_1 = \{32, \ldots, 63\}, \ldots, S_8 = \{224, \ldots, 255\}$$

How many bits needed to distinguish $S_i, S_j \subseteq S$?

$$\log \frac{256}{32} = \log 8 = 3$$

$$\log \frac{256}{32} = \log \left(\frac{32}{256}\right)^{-1} = \log \left(\frac{|S_i|}{|S|}\right)^{-1}$$

Information Theory Intuition

Logarithm gives the necessary number of bits

$$S = \{0, 1, 2, 3, \ldots, 254, 255\}$$

How many bits needed to distingish $x, y \in S$? $\log_2(256) = 8$

What about a partition?

$$S_0 = \{0, \ldots, 31\}, S_1 = \{32, \ldots, 63\}, \ \ldots, S_8 = \{224, \ldots, 255\}$$

How many bits needed to distinguish $S_i, S_j \subseteq S$?

$$\log \frac{256}{32} = \log 8 = 3$$

$$\log \frac{256}{32} = \log \left(\frac{32}{256}\right)^{-1} = \log \left(\frac{|S_i|}{|S|}\right)^{-1} = \log \frac{1}{p(S_i)}$$

Information Theory Intuition

Information Entropy, $H = \sum p_i \log \frac{1}{p_i}$

Information Theory Intuition

Information Entropy, $H = \sum p_i \log \frac{1}{p_i} = E\left[\log \frac{1}{p_i}\right]$

Information Theory Intuition

Information Entropy, $H = \sum p_i \log \frac{1}{p_i} = E\left[\log \frac{1}{p_i}\right]$

The expected amount of information gain.

Information Theory Intuition

Information Entropy, $H = \sum p_i \log \frac{1}{p_i} = E\left[\log \frac{1}{p_i}\right]$

The expected amount of information gain.
The expected amount of **"surprise"**.

Information Theory Intuition

Information Entropy, $H = \sum p_i \log \frac{1}{p_i} = E\left[\log \frac{1}{p_i}\right]$

The expected amount of information gain.
The expected amount of **"surprise"**.

Seattle Weather, Always Raining

$p_{rain} = 1, p_{sun} = 0$

Information Theory Intuition

Information Entropy, $H = \sum p_i \log \frac{1}{p_i} = E\left[\log \frac{1}{p_i}\right]$

The expected amount of information gain.
The expected amount of **"surprise"**.

Seattle Weather, Always Raining

$p_{rain} = 1, p_{sun} = 0$ $\qquad$ $H = 0$

Information Theory Intuition

Information Entropy, $H = \sum p_i \log \frac{1}{p_i} = E\left[\log \frac{1}{p_i}\right]$

The expected amount of information gain.
The expected amount of **“surprise”**.

Seattle Weather, Always Raining

$p_{rain} = 1, p_{sun} = 0$ $\qquad H = 0$

Costa Rica Weather, Coin Flip

$p_{rain} = \frac{1}{2}, p_{sun} = \frac{1}{2}$

Information Theory Intuition

Information Entropy, $H = \sum p_i \log \frac{1}{p_i} = E\left[\log \frac{1}{p_i}\right]$

The expected amount of information gain.
The expected amount of "**surprise**".

Seattle Weather, Always Raining

$p_{rain} = 1, p_{sun} = 0$ $\quad H = 0$

Costa Rica Weather, Coin Flip

$p_{rain} = \frac{1}{2}, p_{sun} = \frac{1}{2}$ $\quad H = 1$

Information Theory Intuition

Information Entropy, $H = \sum p_i \log \frac{1}{p_i} = E\left[\log \frac{1}{p_i}\right]$

The expected amount of information gain.
The expected amount of **“surprise”**.

Seattle Weather, Always Raining

$p_{rain} = 1, p_{sun} = 0 \qquad H = 0$

Costa Rica Weather, Coin Flip

$p_{rain} = \frac{1}{2}, p_{sun} = \frac{1}{2} \qquad H = 1$

Santa Barbara Weather, Almost Always Beautiful!

$p_{rain} = \frac{1}{10}, p_{sun} = \frac{9}{10}$

Information Theory Intuition

Information Entropy, $H = \sum p_i \log \frac{1}{p_i} = E\left[\log \frac{1}{p_i}\right]$

The expected amount of information gain.
The expected amount of "**surprise**".

Seattle Weather, Always Raining

$p_{rain} = 1, p_{sun} = 0$ $\qquad H = 0$

Costa Rica Weather, Coin Flip

$p_{rain} = \frac{1}{2}, p_{sun} = \frac{1}{2}$ $\qquad H = 1$

Santa Barbara Weather, Almost Always Beautiful!

$p_{rain} = \frac{1}{10}, p_{sun} = \frac{9}{10}$ $\qquad H = 0.4960$

Outline

Software Side Channel Analysis

High Level Idea:

- Define symbolic execution observation model (o_i):

Software Side Channel Analysis

High Level Idea:

- Define symbolic execution observation model (o_i):
  - Execution time $\mapsto$ number of instructions (lines of code) executed.
  - Memory $\mapsto$ number of `malloc`, bytes written to file, ...

Software Side Channel Analysis

High Level Idea:

- Define symbolic execution observation model (o_i):
  - Execution time $\mapsto$ number of instructions (lines of code) executed.
  - Memory $\mapsto$ number of `malloc`, bytes written to file, ...
- Keep track of observations o_i during PSE.

Software Side Channel Analysis

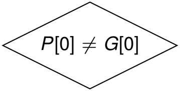
High Level Idea:

- Define symbolic execution observation model (o_i):
  - Execution time $\mapsto$ number of instructions (lines of code) executed.
  - Memory $\mapsto$ number of `malloc`, bytes written to file, ...
- Keep track of observations o_i during PSE.
- Quantify information gain: $H = \sum p_i \log \frac{1}{p_i}$

```
bool checkPIN(guess[])
for(i = 0; i < 4; i++)
 if(guess[i] != PIN[i])
  return false
return true
```

P: PIN, G: guess

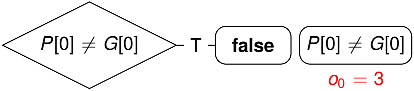
$o_i =$ lines of code




```
bool checkPIN(guess[])
for(i = 0; i < 4; i++)
 if(guess[i] != PIN[i])
  return false
return true
```

P: PIN, G: guess

$o_i =$ lines of code




```
bool checkPIN(guess[])
for(i = 0; i < 4; i++)
 if(guess[i] != PIN[i])
  return false
return true
```

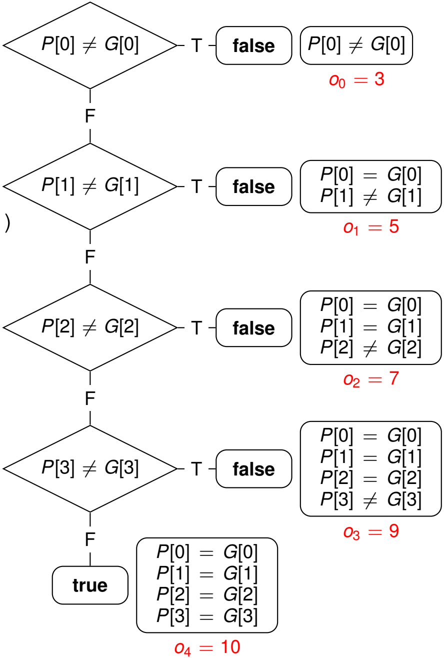
P: PIN, G: guess

o_i = lines of code

```
bool checkPIN(guess[])
for(i = 0; i < 4; i++)
 if(guess[i] != PIN[i])
  return false
return true
```

P: PIN, G: guess

o_i = lines of code



i	0	1	2	3	4
PC_i	$P[0] \neq G[0]$	$P[0] = G[0]$ $P[1] \neq G[1]$	$P[0] = G[0]$ $P[1] = G[1]$ $P[2] \neq G[2]$	$P[0] = G[0]$ $P[1] = G[1]$ $P[2] = G[2]$ $P[3] \neq G[3]$	$P[0] = G[0]$ $P[1] = G[1]$ $P[2] = G[2]$ $P[3] = G[3]$
return	false	false	false	false	true
$\lvert PC_i \rvert$	128	64	32	16	16
p_i	1/2	1/4	1/8	1/16	1/16
o_i	3	5	7	9	10

i	0	1	2	3	4
PC_i	$P[0] \neq G[0]$	$P[0] = G[0]$ $P[1] \neq G[1]$	$P[0] = G[0]$ $P[1] = G[1]$ $P[2] \neq G[2]$	$P[0] = G[0]$ $P[1] = G[1]$ $P[2] = G[2]$ $P[3] \neq G[3]$	$P[0] = G[0]$ $P[1] = G[1]$ $P[2] = G[2]$ $P[3] = G[3]$
return	false	false	false	false	true
$\|PC_i\|$	128	64	32	16	16
p_i	1/2	1/4	1/8	1/16	1/16
o_i	3	5	7	9	10

i	0	1	2	3	4
PC_i	$P[0] \neq G[0]$	$P[0] = G[0]$ $P[1] \neq G[1]$	$P[0] = G[0]$ $P[1] = G[1]$ $P[2] \neq G[2]$	$P[0] = G[0]$ $P[1] = G[1]$ $P[2] = G[2]$ $P[3] \neq G[3]$	$P[0] = G[0]$ $P[1] = G[1]$ $P[2] = G[2]$ $P[3] = G[3]$
return	false	false	false	false	true
$\|PC_i\|$	128	64	32	16	16
p_i	1/2	1/4	1/8	1/16	1/16
o_i	3	5	7	9	10

i	0	1	2	3	4
PC_i	$P[0] \neq G[0]$	$P[0] = G[0]$ $P[1] \neq G[1]$	$P[0] = G[0]$ $P[1] = G[1]$ $P[2] \neq G[2]$	$P[0] = G[0]$ $P[1] = G[1]$ $P[2] = G[2]$ $P[3] \neq G[3]$	$P[0] = G[0]$ $P[1] = G[1]$ $P[2] = G[2]$ $P[3] = G[3]$
return	false	false	false	false	true
$\|PC_i\|$	128	64	32	16	16
p_i	1/2	1/4	1/8	1/16	1/16
o_i	3	5	7	9	10

i	0	1	2	3	4
PC_i	$P[0] \neq G[0]$	$P[0] = G[0]$ $P[1] \neq G[1]$	$P[0] = G[0]$ $P[1] = G[1]$ $P[2] \neq G[2]$	$P[0] = G[0]$ $P[1] = G[1]$ $P[2] = G[2]$ $P[3] \neq G[3]$	$P[0] = G[0]$ $P[1] = G[1]$ $P[2] = G[2]$ $P[3] = G[3]$
return	false	false	false	false	true
$\|PC_i\|$	128	64	32	16	16
p_i	1/2	1/4	1/8	1/16	1/16
o_i	3	5	7	9	10

i	0	1	2	3	4
PC_i	$P[0] \neq G[0]$	$P[0] = G[0]$ $P[1] \neq G[1]$	$P[0] = G[0]$ $P[1] = G[1]$ $P[2] \neq G[2]$	$P[0] = G[0]$ $P[1] = G[1]$ $P[2] = G[2]$ $P[3] \neq G[3]$	$P[0] = G[0]$ $P[1] = G[1]$ $P[2] = G[2]$ $P[3] = G[3]$
return	false	false	false	false	true
$\|PC_i\|$	128	64	32	16	16
p_i	1/2	1/4	1/8	1/16	1/16
o_i	3	5	7	9	10

$$H = \sum p_i \log \frac{1}{p_i} = 1.8750$$

i	0	1	2	3	4
PC_i	$P[0] \neq G[0]$	$P[0] = G[0]$ $P[1] \neq G[1]$	$P[0] = G[0]$ $P[1] = G[1]$ $P[2] \neq G[2]$	$P[0] = G[0]$ $P[1] = G[1]$ $P[2] = G[2]$ $P[3] \neq G[3]$	$P[0] = G[0]$ $P[1] = G[1]$ $P[2] = G[2]$ $P[3] = G[3]$
return	false	false	false	false	true
$\|PC_i\|$	128	64	32	16	16
p_i	1/2	1/4	1/8	1/16	1/16
o_i	3	5	7	9	10

$$H = \sum p_i \log \frac{1}{p_i} = 1.8750$$

A measure of program vulnerability

H = expected amount of information that an adversary can gain in 1 guess.

Side Channel Analysis

A more secure 4 digit PIN verification function:

```
public verifyPassword (guess[])
  matched = true
  for (int i = 0; i < 4; i++)
    if (guess[i] != PIN[i])
        matched = false
    else
        matched = matched
  return matched
```

Side Channel Analysis

A more secure 4 digit PIN verification function:

```
public verifyPassword (guess[])
  matched = true
  for (int i = 0; i < 4; i++)
    if (guess[i] != PIN[i])
            matched = false
    else
            matched = matched
  return matched
```

Only 2 oservables: o_0 = perfect match, o_1 = not perfect match.

Side Channel Analysis

A more secure 4 digit PIN verification function:

```
public verifyPassword (guess[])
  matched = true
  for (int i = 0; i < 4; i++)
    if (guess[i] != PIN[i])
          matched = false
    else
          matched = matched
  return matched
```

Only 2 oservables: o_0 = perfect match, o_1 = not perfect match.

$$p(o_0) = 1/16, p(o_1) = 15/16$$

Side Channel Analysis

A more secure 4 digit PIN verification function:

```
public verifyPassword (guess[])
  matched = true
  for (int i = 0; i < 4; i++)
    if (guess[i] != PIN[i])
          matched = false
    else
          matched = matched
  return matched
```

Only 2 oservables: o_0 = perfect match, o_1 = not perfect match.

$$p(o_0) = 1/16, p(o_1) = 15/16$$

$$H_{secure} = 0.33729$$

Side Channel Analysis

A more secure 4 digit PIN verification function:

```
public verifyPassword (guess[])
  matched = true
  for (int i = 0; i < 4; i++)
    if (guess[i] != PIN[i])
          matched = false
    else
          matched = matched
  return matched
```

Only 2 oservables: o_0 = perfect match, o_1 = not perfect match.

$$p(o_0) = 1/16, p(o_1) = 15/16$$

$$H_{secure} = 0.33729 < H_{insecure} = 1.8750$$

Side Channel Analysis

Summary

- Observe non-functional aspects of computatation to learn information.
- Probabalistic symbolic execution provides p_i, o_i
- Quantify information gain: $H = \sum p_i \log \frac{1}{p_i}$

Side Channel Analysis

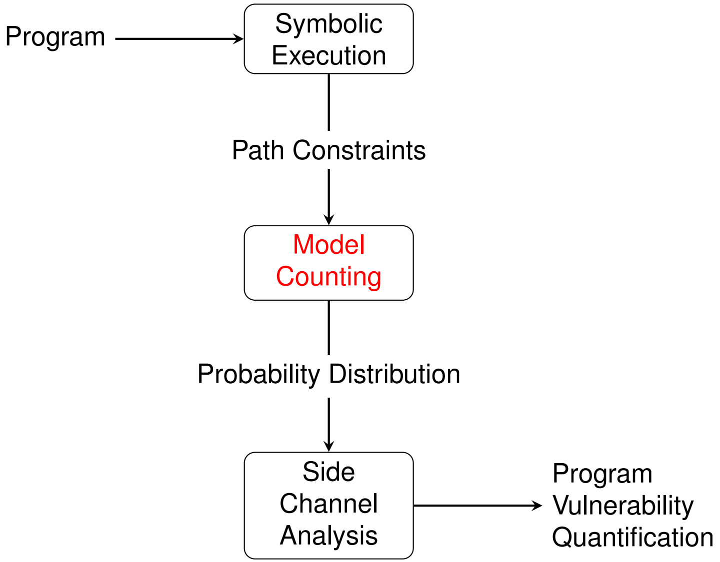
Summary

- Observe non-functional aspects of computatation to learn information.
- Probabalistic symbolic execution provides p_i, o_i
- Quantify information gain: $H = \sum p_i \log \frac{1}{p_i}$

Remaining issues

- How to determine the number of solutions to path constraints?
- Path constraints for real programs could involve boolean formulas, strings, numeric constraints.

Overview



Program
Symbolic Execution
Path Constraints
Model Counting
Probability Distribution
Side Channel Analysis
Program Vulnerability Quantification

Model Counting

Recall the classic (boolean) SAT problem

Given a formula ϕ from propositional logic, is it possible to assign all variables the values T (true) or F (false) so that the formula is true?

Model Counting

Recall the classic (boolean) SAT problem

Given a formula ϕ from propositional logic, is it possible to assign all variables the values T (true) or F (false) so that the formula is true?

Example:

$$\phi = (x \vee y) \wedge (\neg x \vee z) \wedge (z \vee w) \wedge x \wedge (y \vee v)$$

Model Counting

Recall the classic (boolean) SAT problem

Given a formula ϕ from propositional logic, is it possible to assign all variables the values T (true) or F (false) so that the formula is true?

Example:

$$\phi = (x \vee y) \wedge (\neg x \vee z) \wedge (z \vee w) \wedge x \wedge (y \vee v)$$

ϕ is satisfiable by setting

$$(x, y, z, w, v) = (T, F, T, F, T).$$

Model Counting

Recall the classic (boolean) SAT problem

Given a formula ϕ from propositional logic, is it possible to assign all variables the values T (true) or F (false) so that the formula is true?

Example:

$$\phi = (x \vee y) \wedge (\neg x \vee z) \wedge (z \vee w) \wedge x \wedge (y \vee v)$$

ϕ is satisfiable by setting

$$(x, y, z, w, v) = (T, F, T, F, T).$$

A satisfying assignment is called a **model** for ϕ.

Model Counting

The **model counting problem**

Given a formula ϕ over some theory (Boolean, LIA, Strings, ...)

how many models are there for ϕ?

Model Counting

The **model counting problem**

Given a formula ϕ over some theory (Boolean, LIA, Strings, . . .)

how many models are there for ϕ?

Difficulty of Model Counting

Model counting is "at least as hard" than satisfiability check.

Model Counting

The **model counting problem**

Given a formula ϕ over some theory (Boolean, LIA, Strings, ...)

how many models are there for ϕ?

Difficulty of Model Counting

Model counting is "at least as hard" than satisfiability check.

$$|\phi| > 0 \iff \phi \text{ is satisfiable}$$

Work on Model Counting

- Stanley. Enumerative Combinatorics Chapter 4. 2004.
- Sedgwick. Analytic Combinatorics Chapter 5: Generating Functions. 2009
- Biere. Handbook of Satisfiability. Chapter 20: Model Counting. 2009
- Pugh. Counting Solutions to Presburger Formulas: How and Why. 1994
- Parker. An Automata-Theoretic Algorithm for Counting Solutions to Presburger Formulas. Compiler Construction 2004
- Boigelot. Counting the solutions of Presburger equations without enumerating them. TCS 2004.
- Barvinok. A polynomial time algorithm for counting integral points in polyhedra when the dimension is fixed. Mathematics of Operations Research 1994
- De Loerab. Effective lattice point counting in rational convex polytopes. JSC 2004
- Verdoolaege. Counting integer points in parametric polytopes using Barvinoks's Rational Functions. 2007
- Kopf Symbolic Polytopes for Quantitative Interpolation and Verification. CAV 2015
- Luu. A Model Counter For Constraints Over Unbounded Strings. PLDI 2014
- Ravikumara. Weak minimization of DFA - an algorithm and applications.Implementation and Application of Automata 2004
- Chomsky. The Algebraic Theory of Context-Free Languages. 1963
- Phan. Model Counting Modulo Theories. PhD Thesis 2014.
- Birnbaum. The good old Davis-Putnam procedure helps counting models. JAIR 1999

Outline

Model Counting Boolean SAT

x	y	z	w	v	F
F	F	F	F	F	F
⋮	⋮	⋮	⋮	⋮	⋮
T	F	F	T	T	F
T	F	T	F	F	F
T	F	T	F	T	T
T	F	T	T	F	F
T	F	T	T	T	T
T	T	F	F	F	F
T	T	F	F	T	F
T	T	F	T	F	F
T	T	F	T	T	F
T	T	T	F	F	T
T	T	T	F	T	T
T	T	T	T	F	T
T	T	T	T	T	T

Model Counting Boolean SAT

x	y	z	w	v	F
F	F	F	F	F	F
⋮	⋮	⋮	⋮	⋮	⋮
T	F	F	T	T	F
T	F	T	F	F	F
T	**F**	**T**	**F**	**T**	**T**
T	F	T	T	F	F
T	**F**	**T**	**T**	**T**	**T**
T	T	F	F	F	F
T	T	F	F	T	F
T	T	F	T	F	F
T	T	F	T	T	F
T	**T**	**T**	**F**	**F**	**T**
T	**T**	**T**	**F**	**T**	**T**
T	**T**	**T**	**T**	**F**	**T**
T	**T**	**T**	**T**	**T**	**T**

ϕ has 6 models.

Model Counting Boolean SAT

x	y	z	w	v	F
F	F	F	F	F	F
⋮	⋮	⋮	⋮	⋮	⋮
T	F	F	T	T	F
T	F	T	F	F	F
T	**F**	**T**	**F**	**T**	**T**
T	F	T	T	F	F
T	**F**	**T**	**T**	**T**	**T**
T	T	F	F	F	F
T	T	F	F	T	F
T	T	F	T	F	F
T	T	F	T	T	F
T	**T**	**T**	**F**	**F**	**T**
T	**T**	**T**	**F**	**T**	**T**
T	**T**	**T**	**T**	**F**	**T**
T	**T**	**T**	**T**	**T**	**T**

ϕ has 6 models.

Truth table method is $\theta(2^n)$.

Davis-Putnam-Logemann-Loveland (DPLL) Algorithm

DPLL can be converted into a procedure for $\#$CNF-SAT.

Function : DPLL(ϕ, t)
Input : CNF formula ϕ over *n* variables; $t \in \mathbb{Z}$
Output : $\#\phi$, the model count of ϕ
begin
  UnitPropagate(ϕ)
  if ϕ has false clause **then return** *false*
  if all clauses of ϕ satisfied **then return** *true*
  x $\leftarrow$ SelectBranchVariable(ϕ)
  return DPLL($\phi[x \mapsto true]$, $t-1$) $\vee$ DPLL($\phi[x \mapsto true]$, $t-1$)
end

Davis-Putnam-Logemann-Loveland (DPLL) Algorithm

DPLL can be converted into a procedure for $\#$CNF-SAT.

Function : DPLL(ϕ, t)
Input : CNF formula ϕ over n variables; $t \in \mathbb{Z}$
Output : $\#\phi$, the model count of ϕ
begin
  UnitPropagate(ϕ)
  if ϕ has false clause **then return** *false*
  if all clauses of ϕ satisfied **then return** *true*
  x $\leftarrow$ SelectBranchVariable(ϕ)
  return DPLL($\phi[x \mapsto true], t-1$) $\vee$ DPLL($\phi[x \mapsto true], t-1$)
end

Davis-Putnam-Logemann-Loveland (DPLL) Algorithm

DPLL can be converted into a procedure for $\#$CNF-SAT.

Function : DPLL(ϕ, t)
Input : CNF formula ϕ over n variables; $t \in \mathbb{Z}$
Output : $\#\phi$, the model count of ϕ
begin
UnitPropagate(ϕ)
if ϕ has false clause **then return** *false*
if all clauses of ϕ satisfied **then return** *true*
x $\leftarrow$ SelectBranchVariable(ϕ)
return DPLL($\phi[x \mapsto true], t - 1) \vee$ DPLL($\phi[x \mapsto true], t - 1$)
end

Davis-Putnam-Logemann-Loveland (DPLL) Algorithm

DPLL can be converted into a procedure for $\#$CNF-SAT.

Function : DPLL(ϕ, t)
Input : CNF formula ϕ over *n* variables; $t \in \mathbb{Z}$
Output : $\#\phi$, the model count of ϕ
begin
  UnitPropagate(ϕ)
  if ϕ has false clause **then return** 0
  if all clauses of ϕ satisfied **then return** *true*
  x $\leftarrow$ SelectBranchVariable(ϕ)
  return DPLL($\phi[x \mapsto true], t-1$) $\vee$ DPLL($\phi[x \mapsto true], t-1$)
end

Davis-Putnam-Logemann-Loveland (DPLL) Algorithm

DPLL can be converted into a procedure for $\#$CNF-SAT.

Function : DPLL(ϕ, t)
Input : CNF formula ϕ over n variables; $t \in \mathbb{Z}$
Output : $\#\phi$, the model count of ϕ
begin
  UnitPropagate(ϕ)
  if ϕ has false clause **then return** 0
  if all clauses of ϕ satisfied **then return** *true*
  x $\leftarrow$ SelectBranchVariable(ϕ)
  return DPLL($\phi[x \mapsto \textit{true}], t - 1$) $\vee$ DPLL($\phi[x \mapsto \textit{true}], t - 1$)
end

Davis-Putnam-Logemann-Loveland (DPLL) Algorithm

DPLL can be converted into a procedure for $\#$CNF-SAT.

Function : DPLL(ϕ, t)
Input : CNF formula ϕ over n variables; $t \in \mathbb{Z}$
Output : $\#\phi$, the model count of ϕ
begin
  UnitPropagate(ϕ)
  if ϕ has false clause **then return** 0
  if all clauses of ϕ satisfied **then return** *true*
  x $\leftarrow$ SelectBranchVariable(ϕ)
  return DPLL($\phi[x \mapsto true], t - 1$) $\vee$ DPLL($\phi[x \mapsto true], t - 1$)
end

Davis-Putnam-Logemann-Loveland (DPLL) Algorithm

DPLL can be converted into a procedure for #CNF-SAT.

Function : DPLL(ϕ, t)
Input : CNF formula ϕ over n variables; $t \in \mathbb{Z}$
Output : $\#\phi$, the model count of ϕ
begin
  UnitPropagate(ϕ)
  if ϕ has false clause **then return** 0
  if all clauses of ϕ satisfied **then return** 2^t
  x $\leftarrow$ SelectBranchVariable(ϕ)
  return DPLL($\phi[x \mapsto true], t-1$) $\vee$ DPLL($\phi[x \mapsto true], t-1$)
end

Davis-Putnam-Logemann-Loveland (DPLL) Algorithm

DPLL can be converted into a procedure for $\#$CNF-SAT.

Function : DPLL(ϕ, t)
Input : CNF formula ϕ over n variables; $t \in \mathbb{Z}$
Output : $\#\phi$, the model count of ϕ
begin
  UnitPropagate(ϕ)
  if ϕ has false clause **then return** 0
  if all clauses of ϕ satisfied **then return** 2^t
  x $\leftarrow$ SelectBranchVariable(ϕ)
  return DPLL($\phi[x \mapsto \mathit{true}], t - 1$) $+$ DPLL($\phi[x \mapsto \mathit{true}], t - 1$)
end

Counting with DPLL

$$\phi = \{x \vee y, \neg x \vee z, z \vee w, x, y \vee v\}, n = 5$$

$\{z, x, y \vee v\}$ $t = 5$

Counting with DPLL

$$\phi = \{x \vee y, \neg x \vee z, z \vee w, x, y \vee v\}, n = 5$$

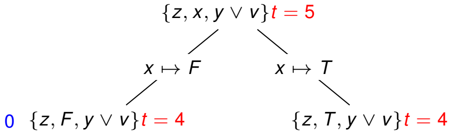
$\{z, x, y \vee v\}$ $t = 5$

$x \mapsto F$

0 $\{z, F, y \vee v\}$ $t = 4$

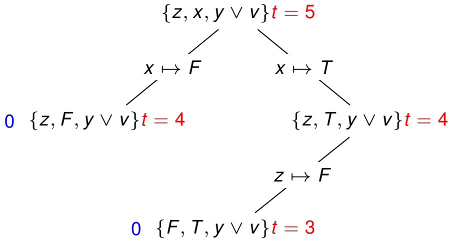
Counting with DPLL

$$\phi = \{x \vee y, \neg x \vee z, z \vee w, x, y \vee v\}, n = 5$$



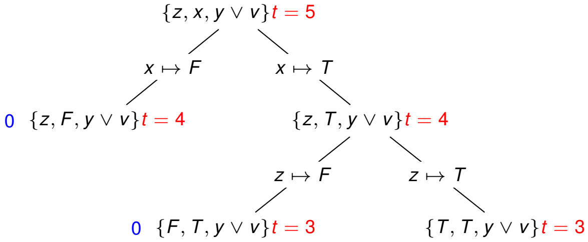
Counting with DPLL

$$\phi = \{x \vee y, \neg x \vee z, z \vee w, x, y \vee v\}, n = 5$$



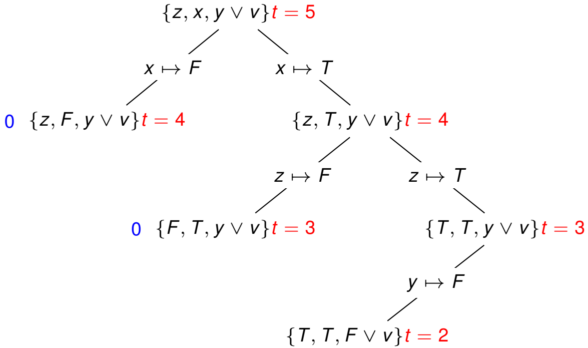
Counting with DPLL

$$\phi = \{x \vee y, \neg x \vee z, z \vee w, x, y \vee v\}, n = 5$$



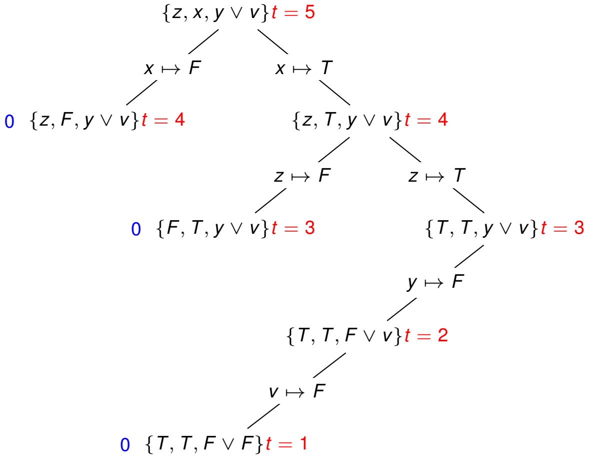
Counting with DPLL

$$\phi = \{x \vee y, \neg x \vee z, z \vee w, x, y \vee v\}, n = 5$$



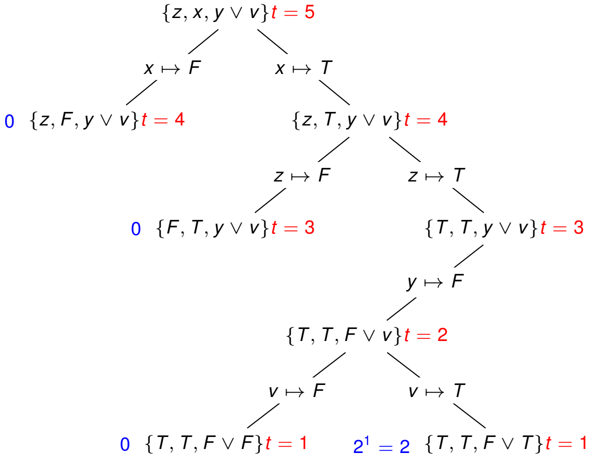
Counting with DPLL

$$\phi = \{x \vee y, \neg x \vee z, z \vee w, x, y \vee v\}, n = 5$$



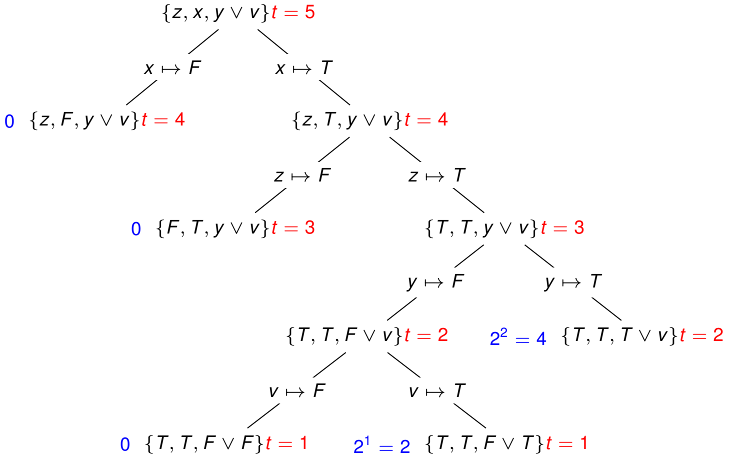
Counting with DPLL

$$\phi = \{x \vee y, \neg x \vee z, z \vee w, x, y \vee v\}, n = 5$$



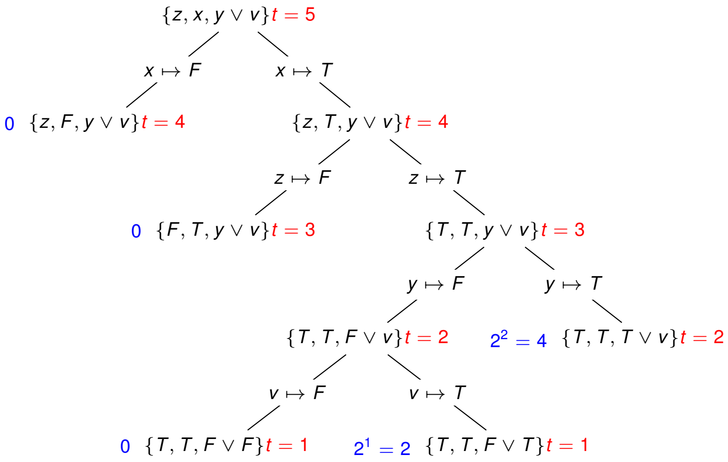
Counting with DPLL

$$\phi = \{x \vee y, \neg x \vee z, z \vee w, x, y \vee v\}, n = 5$$



Counting with DPLL

$$\phi = \{x \vee y, \neg x \vee z, z \vee w, x, y \vee v\}, n = 5$$



Result: $0 + 0 + 0 + 2 + 4 = 6$ models

Model Counting for Other Theories

Model Counting for Other Theories

Generating functions are a way to compactly represent (possibly infinite) sequences.

Model Counting for Other Theories

Generating functions are a way to compactly represent (possibly infinite) sequences.

$$g(z) = \frac{1}{(1-z)^3}$$

Model Counting for Other Theories

Generating functions are a way to compactly represent (possibly infinite) sequences.

$$g(z) = \frac{1}{(1-z)^3} = \sum_{k=0}^{\infty} a_k z^k$$

Model Counting for Other Theories

Generating functions are a way to compactly represent (possibly infinite) sequences.

$$g(z) = \frac{1}{(1-z)^3} = \sum_{k=0}^{\infty} a_k z^k$$

$$g(z) = 1z^0 + 3z^1 + 6z^2 + 10z^3 + 15z^4 + \dots$$

Model Counting for Other Theories

Generating functions are a way to compactly represent (possibly infinite) sequences.

$$g(z) = \frac{1}{(1-z)^3} = \sum_{k=0}^{\infty} a_k z^k$$

$$g(z) = 1z^0 + 3z^1 + 6z^2 + 10z^3 + 15z^4 + \dots$$

$$g(z) = a_0 z^0 + a_1 z^1 + a_2 z^2 + a_3 z^3 + a_4 z^4 + \dots$$

Outline

Model Counting Strings

A formula over the theory of strings can involve

- Word Equations: $X \circ U = Y \circ Z$

Model Counting Strings

A formula over the theory of strings can involve

- Word Equations: $X \circ U = Y \circ Z$
- Length Constraints: $4 < Length(X) < 10$

Model Counting Strings

A formula over the theory of strings can involve

- Word Equations: $X \circ U = Y \circ Z$
- Length Constraints: $4 < Length(X) < 10$
- Regular Language Membership: $X \in (a|b)^*$

Model Counting Strings

A formula over the theory of strings can involve

- Word Equations: $X \circ U = Y \circ Z$
- Length Constraints: $4 < Length(X) < 10$
- Regular Language Membership: $X \in (a|b)^*$
- and more complex constraints: $(X = \text{substring}(Y, i, j), \ldots)$

Model Counting Strings

A formula over the theory of strings can involve

- Word Equations: $X \circ U = Y \circ Z$
- Length Constraints: $4 < Length(X) < 10$
- Regular Language Membership: $X \in (a|b)^*$
- and more complex constraints: $(X = \text{substring}(Y, i, j), \ldots)$

Regular Expressions

$$X \in (0|(1(01^*0)^*1))^*$$

Q: How many solutions for X?

Regular Expressions

$$X \in (0|(1(01^*0)^*1))^*$$

Q: How many solutions for X? A: Infinitely many!

Regular Expressions

$$X \in (0|(1(01^*0)^*1))^*$$

Q: How many solutions for X? A: Infinitely many!

Q: How many solutions for X of length k?

Regular Expressions

$$X \in (0|(1(01^*0)^*1))^*$$

Q: How many solutions for X? A: Infinitely many!

Q: How many solutions for X of length k?

A generating function for language $\mathcal{L}$ encodes

$$a_k = |\{s : s \in \mathcal{L}, \text{len}(s) = k\}|$$

Regular Expressions

$$X \in (0|(1(01^*0)^*1))^*$$

Q: How many solutions for X? A: Infinitely many!

Q: How many solutions for X of length k?

A generating function for language $\mathcal{L}$ encodes

$$a_k = |\{s : s \in \mathcal{L}, \text{len}(s) = k\}|$$

$$g(z) =$$

Regular Expressions

$$X \in (0|(1(01^*0)^*1))^*$$

Q: How many solutions for X? A: Infinitely many!

Q: How many solutions for X of length k?

A generating function for language $\mathcal{L}$ encodes

$$a_k = |\{s : s \in \mathcal{L}, \text{len}(s) = k\}|$$

$$g(z) = 1z^0$$

k	X	a_k
0	ε	1

Regular Expressions

$$X \in (0|(1(01^*0)^*1))^*$$

Q: How many solutions for X? A: Infinitely many!

Q: How many solutions for X of length k?

A generating function for language $\mathcal{L}$ encodes

$$a_k = |\{s : s \in \mathcal{L}, \text{len}(s) = k\}|$$

$$g(z) = 1z^0 + 1z^1$$

k	X	a_k
0	ε	1
1	0	1

Regular Expressions

$$X \in (0|(1(01^*0)^*1))^*$$

Q: How many solutions for X? A: Infinitely many!

Q: How many solutions for X of length k?

A generating function for language $\mathcal{L}$ encodes

$$a_k = |\{s : s \in \mathcal{L}, \text{len}(s) = k\}|$$

$$g(z) = 1z^0 + 1z^1 + 1z^2$$

k	X	a_k
0	ε	1
1	0	1
2	11	1

Regular Expressions

$$X \in (0|(1(01^*0)^*1))^*$$

Q: How many solutions for X? A: Infinitely many!

Q: How many solutions for X of length k?

A generating function for language $\mathcal{L}$ encodes

$$a_k = |\{s : s \in \mathcal{L}, \text{len}(s) = k\}|$$

$$g(z) = 1z^0 + 1z^1 + 1z^2 + 1z^3$$

k	X	a_k
0	ε	1
1	0	1
2	11	1
3	110	1

Regular Expressions

$$X \in (0|(1(01^*0)^*1))^*$$

Q: How many solutions for X? A: Infinitely many!

Q: How many solutions for X of length k?

A generating function for language $\mathcal{L}$ encodes

$$a_k = |\{s : s \in \mathcal{L}, \text{len}(s) = k\}|$$

$$g(z) = 1z^0 + 1z^1 + 1z^2 + 1z^3 + 3z^4$$

k	X	a_k
0	ε	1
1	0	1
2	11	1
3	110	1
4	1001, 1100, 1111	3

Regular Expressions

$$X \in (0|(1(01^*0)^*1))^*$$

Q: How many solutions for X? A: Infinitely many!

Q: How many solutions for X of length k?

A generating function for language $\mathcal{L}$ encodes

$$a_k = |\{s : s \in \mathcal{L}, \text{len}(s) = k\}|$$

$$g(z) = 1z^0 + 1z^1 + 1z^2 + 1z^3 + 3z^4 + 5z^5 + \dots$$

k	X	a_k
0	ε	1
1	0	1
2	11	1
3	110	1
4	1001, 1100, 1111	3
5	10010, 10101, 11000, 11011, 11110	5

Regular Expressions

For a regular expression constraint, GF can be derived recursively.

Regular Expressions

For a regular expression constraint, GF can be derived recursively.

$$\varepsilon \mapsto 1z^0$$

Regular Expressions

For a regular expression constraint, GF can be derived recursively.

$$\begin{aligned} \varepsilon &\mapsto 1z^0 \\ c &\mapsto 1z^1 \end{aligned}$$

Regular Expressions

For a regular expression constraint, GF can be derived recursively.

$$\begin{array}{lcl} \varepsilon & \mapsto & 1z^0 \\ c & \mapsto & 1z^1 \\ A|B & \mapsto & A(z) + B(z) \end{array}$$

Regular Expressions

For a regular expression constraint, GF can be derived recursively.

$$\begin{array}{lcl} \varepsilon & \mapsto & 1z^0 \\ c & \mapsto & 1z^1 \\ A|B & \mapsto & A(z) + B(z) \\ A \circ B & \mapsto & A(z) \times B(z) \end{array}$$

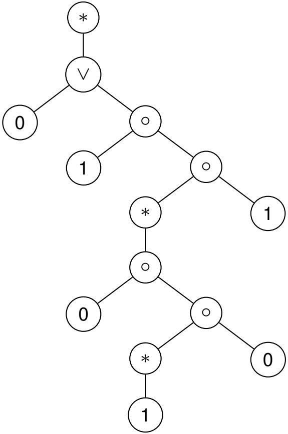
Regular Expressions

For a regular expression constraint, GF can be derived recursively.

$$\begin{array}{lll} \varepsilon & \mapsto & 1z^0 \\ c & \mapsto & 1z^1 \\ A|B & \mapsto & A(z) + B(z) \\ A \circ B & \mapsto & A(z) \times B(z) \\ A^* & \mapsto & 1/(1 - A(z)) \end{array}$$

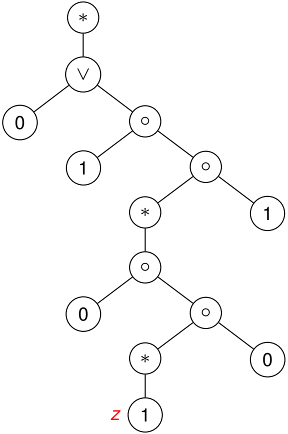
Regular Expressions

$$X \in (0|(1(01^*0)^*1))^*$$



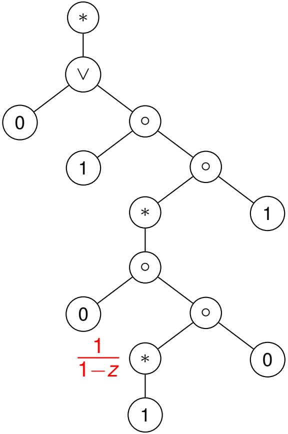
Regular Expressions

$$X \in (0|(1(01^*0)^*1))^*$$



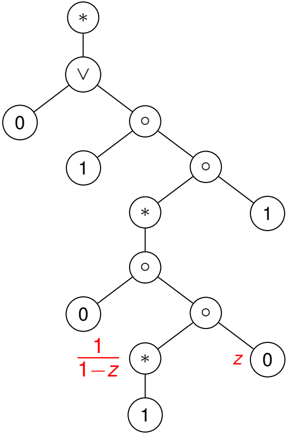
Regular Expressions

$$X \in (0|(1(01^*0)^*1))^*$$



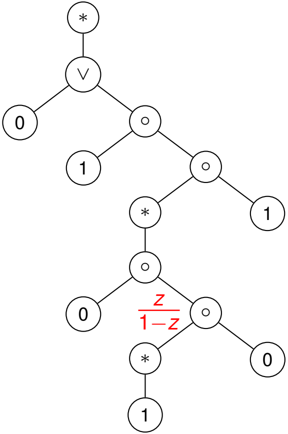
Regular Expressions

$$X \in (0|(1(01^*0)^*1))^*$$



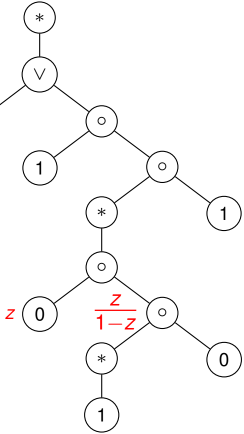
Regular Expressions

$$X \in (0|(1(01^*0)^*1))^*$$



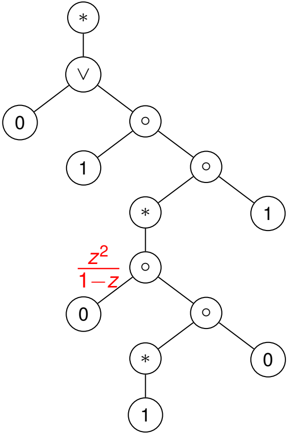
Regular Expressions

$$X \in (0|(1(01^*0)^*1))^*$$



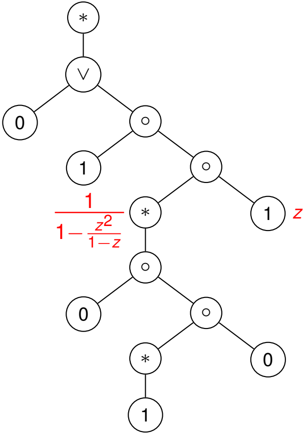
Regular Expressions

$$X \in (0|(1(01^*0)^*1))^*$$



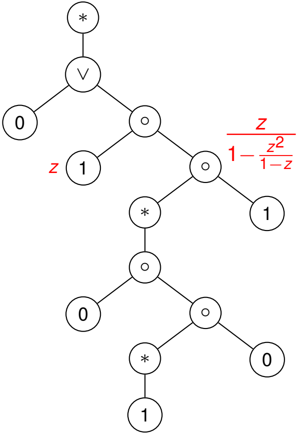
Regular Expressions

$$X \in (0|(1(01^*0)^*1))^*$$



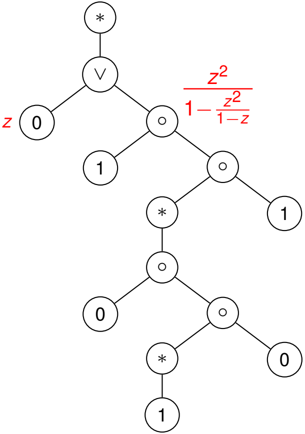
Regular Expressions

$$X \in (0|(1(01^*0)^*1))^*$$



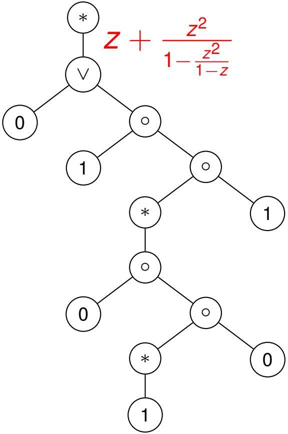
Regular Expressions

$$X \in (0|(1(01^*0)^*1))^*$$



Regular Expressions

$$X \in (0|(1(01^*0)^*1))^*$$

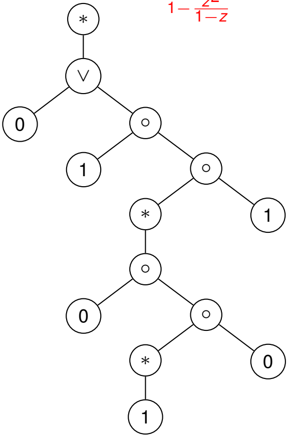


$$z + \frac{z^2}{1-\frac{z^2}{1-z}}$$

Regular Expressions

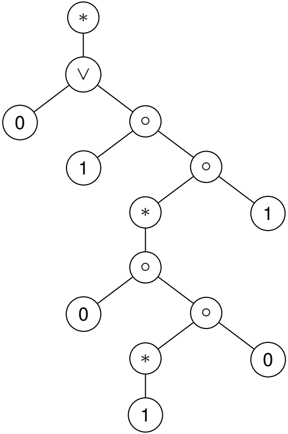
$$X \in (0|(1(01^*0)^*1))^*$$

$$\frac{1}{1-z-\frac{z^2}{1-\frac{z^2}{1-z}}}$$



Regular Expressions

$$X \in (0|(1(01^*0)^*1))^*$$

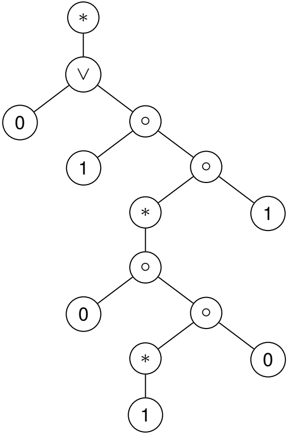


Generating Function:

$$g(z) = \frac{1}{1-z-\frac{z^2}{1-\frac{z^2}{1-z}}}$$

Regular Expressions

$$X \in (0|(1(01^*0)^*1))^*$$



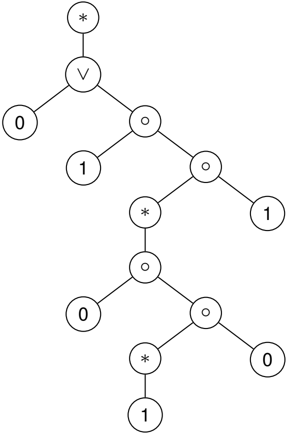
Generating Function:

$$g(z) = \frac{1}{1-z-\frac{z^2}{1-\frac{z^2}{1-z}}}$$

$$= \frac{1-z-z^2}{(z-1)(2z^2+z-1)}$$

Regular Expressions

$$X \in (0|(1(01^*0)^*1))^*$$



Generating Function:

$$g(z) = \frac{1}{1 - z - \frac{z^2}{1 - \frac{z^2}{1-z}}}$$

$$= \frac{1 - z - z^2}{(z-1)(2z^2 + z - 1)}$$

$$g(z) = 1z^0 + 1z^1 + 1z^2 + 1z^3 + 3z^4 + 5z^5 + \dots$$

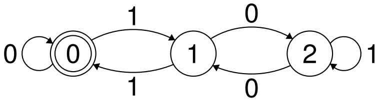
Deterministic Finite Automata

Deterministic Finite Automata

$$X \in (0|(1(01^*0)^*1))^*$$

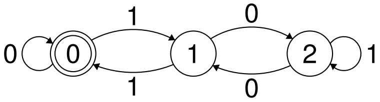
Deterministic Finite Automata

$$X \in (0|(1(01^*0)^*1))^*$$



Deterministic Finite Automata

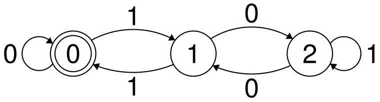
$$X \in (0|(1(01^*0)^*1))^*$$



$$|\{s : s \in \mathcal{L}, \text{len}(s) = k\}| \equiv |\{\pi : \pi \text{ is accepting path of length } k\}|$$

Deterministic Finite Automata

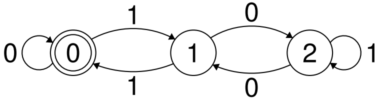
$$X \in (0|(1(01^*0)^*1))^*$$



$$|\{s : s \in \mathcal{L}, \text{len}(s) = k\}| \equiv |\{\pi : \pi \text{ is accepting path of length } k\}|$$

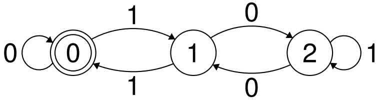
String counting $\equiv$ path counting

Deterministic Finite Automata



How to count paths of length k?

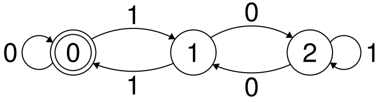
Deterministic Finite Automata



How to count paths of length k?

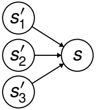
Dynamic Programming

Deterministic Finite Automata



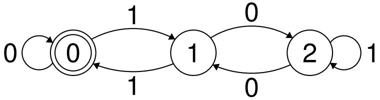
How to count paths of length k?

Dynamic Programming



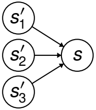
$\eta_s(k)$

Deterministic Finite Automata



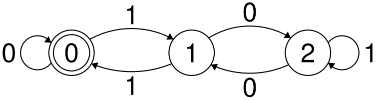
How to count paths of length k?

Dynamic Programming



$$\eta_s(k) = \sum_{s' \to s} \eta_{s'}(k-1)$$

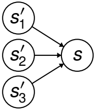
Deterministic Finite Automata



How to count paths of length k?

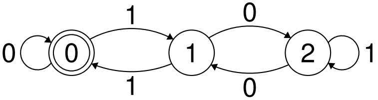
Dynamic Programming

Matrix Exponentiation



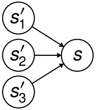
$$\eta_s(k) = \sum_{s' \to s} \eta_{s'}(k-1)$$

Deterministic Finite Automata



How to count paths of length k?

Dynamic Programming

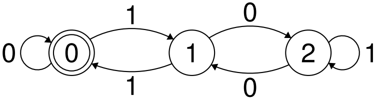


$$\eta_s(k) = \sum_{s' \to s} \eta_{s'}(k-1)$$

Matrix Exponentiation

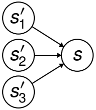
$$A = \begin{pmatrix} 1 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 1 \end{pmatrix}$$

Deterministic Finite Automata



How to count paths of length k?

Dynamic Programming



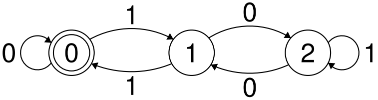
$$\eta_s(k) = \sum_{s' \to s} \eta_{s'}(k-1)$$

Matrix Exponentiation

$$A = \begin{pmatrix} 1 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 1 \end{pmatrix}$$

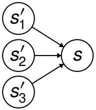
$$(A^k)_{i,j}$$

Deterministic Finite Automata



How to count paths of length k?

Dynamic Programming



$$\eta_s(k) = \sum_{s' \to s} \eta_{s'}(k-1)$$

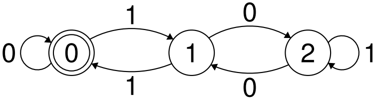
Matrix Exponentiation

$$A = \begin{pmatrix} 1 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 1 \end{pmatrix}$$

$$(A^k)_{i,j}$$

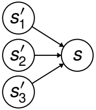
$$(A^4)_{0,0} = 3$$

Deterministic Finite Automata



How to count paths of length k?

Dynamic Programming



$$\eta_s(k) = \sum_{s' \to s} \eta_{s'}(k-1)$$

Matrix Exponentiation

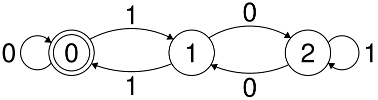
$$A = \begin{pmatrix} 1 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 1 \end{pmatrix}$$

$$(A^k)_{i,j}$$

$$(A^4)_{0,0} = 3$$

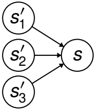
Generating Functions

Deterministic Finite Automata



How to count paths of length k?

Dynamic Programming



$$\eta_s(k) = \sum_{s' \to s} \eta_{s'}(k-1)$$

Matrix Exponentiation

$$A = \begin{pmatrix} 1 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 1 \end{pmatrix}$$

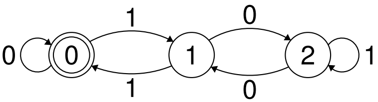
$$(A^k)_{i,j}$$

$$(A^4)_{0,0} = 3$$

Generating Functions

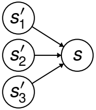
$$A = \begin{pmatrix} 1 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 1 \end{pmatrix}$$

Deterministic Finite Automata



How to count paths of length k?

Dynamic Programming



$$\eta_s(k) = \sum_{s' \to s} \eta_{s'}(k-1)$$

Matrix Exponentiation

$$A = \begin{pmatrix} 1 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 1 \end{pmatrix}$$

$$(A^k)_{i,j}$$

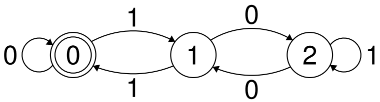
$$(A^4)_{0,0} = 3$$

Generating Functions

$$A = \begin{pmatrix} 1 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 1 \end{pmatrix}$$

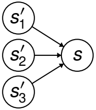
$$g(z) = \frac{\det(I - zA : i, j)}{(-1)^n \det(I - zA)}$$

Deterministic Finite Automata



How to count paths of length k?

Dynamic Programming



$$\eta_s(k) = \sum_{s' \to s} \eta_{s'}(k-1)$$

Matrix Exponentiation

$$A = \begin{pmatrix} 1 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 1 \end{pmatrix}$$

$$(A^k)_{i,j}$$

$$(A^4)_{0,0} = 3$$

Generating Functions

$$A = \begin{pmatrix} 1 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 1 \end{pmatrix}$$

$$g(z) = \frac{\det(I - zA : i, j)}{(-1)^n \det(I - zA)}$$

$$g(z) = \frac{1 - z - z^2}{(z-1)(2z^2 + z - 1)}$$

Outline

Model Counting Linear Integer Arithmetic

Model Counting Linear Integer Arithmetic

What is this language?

$$X \in (0|(1(01^*0)^*1))^*$$

Model Counting Linear Integer Arithmetic

What is this language?

$$X \in (0|(1(01^*0)^*1))^*$$

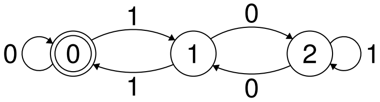
$$L(X) = \{s | s \text{ is a binary number divisible by 3}\}$$

Model Counting Linear Integer Arithmetic

What is this language?

$$X \in (0|(1(01^*0)^*1))^*$$

$$L(X) = \{s | s \text{ is a binary number divisible by } 3\}$$

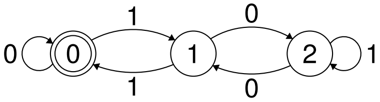


Model Counting Linear Integer Arithmetic

What is this language?

$$X \in (0|(1(01^*0)^*1))^*$$

$$L(X) = \{s | s \text{ is a binary number divisible by 3}\}$$



Idea: DFA can represent (some) relations on sets of binary integers. We can use similar techniques that we used for #String to solve #LIA.

Model Counting Linear Integer Arithmetic

Quantifier-Free Linear Integer Arithmetic $(\mathbb{Z}, +, <)$.

Model Counting Linear Integer Arithmetic

Quantifier-Free Linear Integer Arithmetic $(\mathbb{Z}, +, <)$.

Constraints of the form:

$$Ax < B, x \in \mathbb{Z}^n$$

Model Counting Linear Integer Arithmetic

Quantifier-Free Linear Integer Arithmetic $(\mathbb{Z}, +, <)$.

Constraints of the form:

$$Ax < B, x \in \mathbb{Z}^n$$

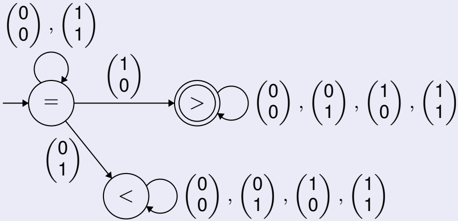
It is possible to represent the solutions to a set of LIA constraints as a binary multi-track DFA.

Binary Multi-track DFA

Solution DFA for LIA constraints.

- Read bits of x and y from most to least significant.
- Alphabet is a tuple of bits: $\begin{pmatrix} b_x \\ b_y \end{pmatrix}$

Solution DFA for the constraint $x > y$.

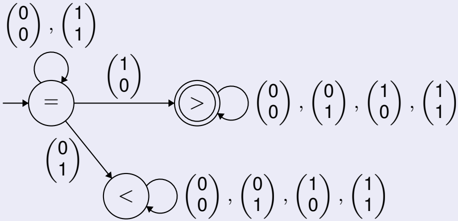


Binary Multi-track DFA

Solution DFA for LIA constraints.

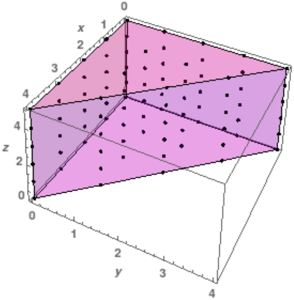
- Read bits of x and y from most to least significant.
- Alphabet is a tuple of bits: $\binom{b_x}{b_y}$

Solution DFA for the constraint $x > y$.

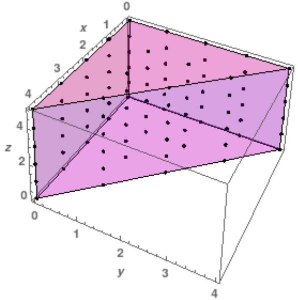


Solutions of length $n \equiv$ solutions within bound 2^n

Integer Grid Points Inside a Polytope, $\mathbb{Z}^n \cap P$



Integer Grid Points Inside a Polytope, $\mathbb{Z}^n \cap P$



- Barvinok Algorithm
- LattE Integrale

Model Counting Summary

Counting Techniques for Different Theories

- Boolean

Model Counting Summary

Counting Techniques for Different Theories

- Boolean
  - Truth Table (Brute Force)
  - DPLL

Model Counting Summary

Counting Techniques for Different Theories

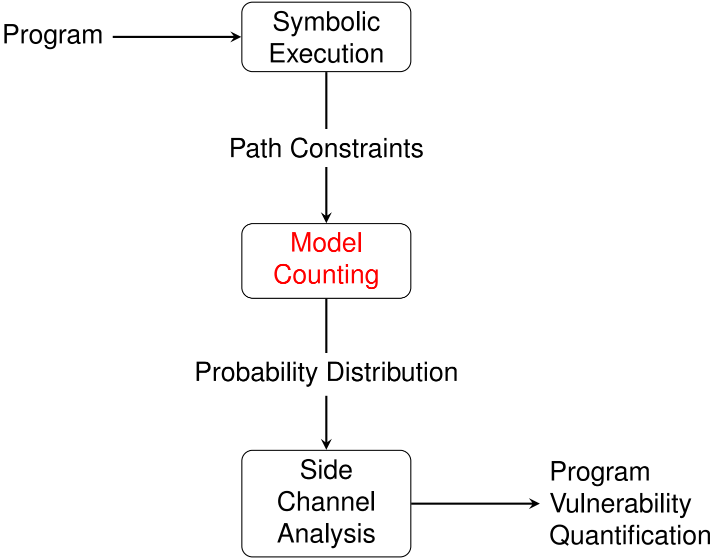
- Boolean
  - Truth Table (Brute Force)
  - DPLL
- Strings
  - Regular Expression with GFs
  - DFA with Dynamic Programming, Matrix Multiplication, GFs

Model Counting Summary

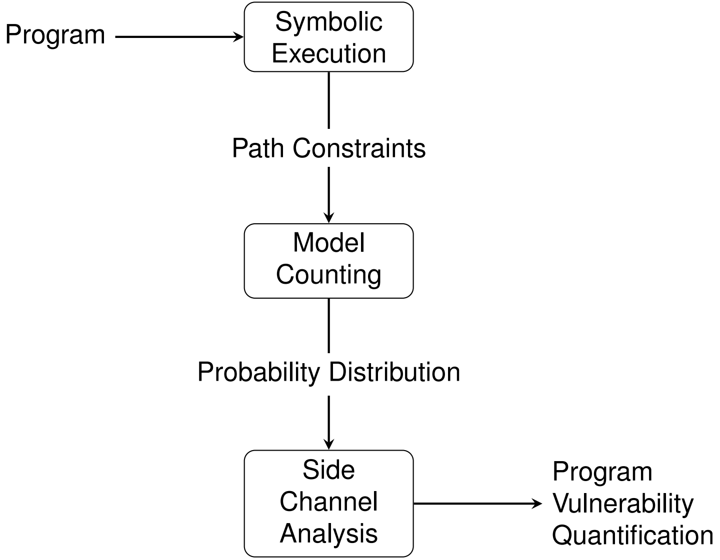
Counting Techniques for Different Theories

- Boolean
  - Truth Table (Brute Force)
  - DPLL
- Strings
  - Regular Expression with GFs
  - DFA with Dynamic Programming, Matrix Multiplication, GFs
- Linear Integer Arithmetic
  - Binary Multi-track DFA
  - Polytope Methods

Review


Program
Symbolic Execution
Path Constraints
Model Counting
Probability Distribution
Side Channel Analysis
Program Vulnerability Quantification

Review


Program
Symbolic Execution
Path Constraints
Model Counting
Probability Distribution
Side Channel Analysis
Program Vulnerability Quantification

My Recent Research

- CAV 2015: "Automata-based model counting for strings".
- FSE 2015: "Automatically computing path complexity of programs".
- Internship Summer 2015 Carnegie: Mellon University / NASA
  - Integration of string model counter with Java Symbolic Path Finder(SPF)
- 2015-2016: Side channel analysis using SPF.
- FSE 2016: "Side channel analysis of segmented oracles." (Submitted)

Questions?

Thank you.