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

Program → Symbolic Execution
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 Probabilistic Symbolic Execution

Model Counting
- Boolean Logic
- Strings
- Linear Integer Arithmetic
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
Software Verification

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

- Never divide by 0
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
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
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**!
Programs can have infinitely many behaviors. Even simple programs can have exponentially many behaviors. Feasible software verification techniques must deal with state space explosion.
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
- 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
- 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
- Cousot. Systematic Design of Program Analysis Frameworks. POPL 1979
Software Verification Tools

A small sample:

- Yang. Using Model Checking to Find Serious File System Errors. OSDI 2004
Outline

Symbolic Execution
  Software Verification
Symbolic Execution
  Probabilistic Symbolic Execution
  SMT Solvers

Side Channel Analysis
  Background and Information Theory
  Via Probabilistic Symbolic Execution

Model Counting
  Boolean Logic
  Strings
  Linear Integer Arithmetic
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
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
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
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
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
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
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
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
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
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
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
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
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
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
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
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

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 Integer Arithmetic
Question

How likely is a certain program behavior?
## Question

How likely is a certain program behavior?

What is 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:

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

Question

How likely is a certain program behavior?

What is 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 probability of a particular program execution path?

### Path Constraint Probability

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

Assuming $D$ is uniformly distributed:

$$p(\text{PC}_i) = \frac{|PC_i|}{|D|}$$
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 $D$. 
### Probabilistic Symbolic Execution

#### Question

How likely is a certain program behavior?

What is 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  

\[ P[0] \neq G[0] \rightarrow \text{false} \]

\[ P[0] \neq G[0] \]

\( P \): 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

<table>
<thead>
<tr>
<th>i</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>$PC_i$</td>
<td>$P[0] \neq G[0]$</td>
<td>$P[0] = G[0]$</td>
<td>$P[0] = G[0]$</td>
<td>$P[0] = G[0]$</td>
<td>$P[0] = G[0]$</td>
</tr>
<tr>
<td>$</td>
<td>PC_i</td>
<td>$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$p_i$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Probabilistic Symbolic Execution

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

<table>
<thead>
<tr>
<th>( i )</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>(</td>
<td>PC_i</td>
<td>)</td>
<td>( )</td>
<td>( )</td>
<td>( )</td>
</tr>
<tr>
<td>( p_i )</td>
<td>( )</td>
<td>( )</td>
<td>( )</td>
<td>( )</td>
<td>( )</td>
</tr>
</tbody>
</table>
Probabilistic Symbolic Execution

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

<table>
<thead>
<tr>
<th>i</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>$PC_i$</td>
<td>$P[0] \neq G[0]$</td>
<td>$P[0] = G[0]$</td>
<td>$P[0] = G[0]$</td>
<td>$P[0] = G[0]$</td>
<td>$P[0] = G[0]$</td>
</tr>
<tr>
<td>$</td>
<td>PC_i</td>
<td>$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$p_i$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[
p_i = \frac{|PC_i|}{|D|}
\]
Probabilistic Symbolic Execution

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

<table>
<thead>
<tr>
<th>i</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>$PC_i$</td>
<td>$P[0] \neq G[0]$</td>
<td>$P[0] = G[0]$</td>
<td>$P[0] = G[0]$</td>
<td>$P[0] = G[0]$</td>
<td>$P[0] = G[0]$</td>
</tr>
<tr>
<td>$</td>
<td>PC_i</td>
<td>$</td>
<td>??????</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$p_i$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

$$p_i = \frac{|PC_i|}{|D|}$$
Probabilistic Symbolic Execution

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

<table>
<thead>
<tr>
<th>$i$</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>$</td>
<td>PC_i</td>
<td>$</td>
<td>128</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$p_i$</td>
<td>????</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

$$p_i = \frac{|PC_i|}{|D|}$$
Probabilistic Symbolic Execution

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

<table>
<thead>
<tr>
<th>i</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>$PC_i$</td>
<td>$P[0] \neq G[0]$</td>
<td>$P[0] = G[0]$</td>
<td>$P[0] = G[0]$</td>
<td>$P[0] = G[0]$</td>
<td>$P[0] = G[0]$</td>
</tr>
<tr>
<td>$</td>
<td>PC_i</td>
<td>$</td>
<td>128</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$p_i$</td>
<td>1/2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

$$p_i = \frac{|PC_i|}{|D|}$$
Probabilistic Symbolic Execution

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

<table>
<thead>
<tr>
<th>$i$</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>$</td>
<td>PC_i</td>
<td>$</td>
<td>128</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$p_i$</td>
<td>1/2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

$p_i = \frac{|PC_i|}{|D|}$
Probabilistic Symbolic Execution

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

<table>
<thead>
<tr>
<th>$i$</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>$</td>
<td>PC_i</td>
<td>$</td>
<td>128</td>
<td>256</td>
<td>512</td>
</tr>
<tr>
<td>$p_i$</td>
<td>$1/2$</td>
<td>$1/4$</td>
<td>$1/8$</td>
<td>$1/16$</td>
<td>$1/32$</td>
</tr>
</tbody>
</table>

$p_i = \frac{|PC_i|}{|D|}$
Probabilistic Symbolic Execution

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

<table>
<thead>
<tr>
<th>( i )</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>( PC_i )</td>
<td>( P[0] \neq G[0] )</td>
<td>( P[0] = G[0] )</td>
<td>( P[0] = G[0] )</td>
<td>( P[0] = G[0] )</td>
<td>( P[0] = G[0] )</td>
</tr>
<tr>
<td>(</td>
<td>PC_i</td>
<td>)</td>
<td>128</td>
<td>????</td>
<td></td>
</tr>
<tr>
<td>( p_i )</td>
<td>( 1/2 )</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[
p_i = \frac{|PC_i|}{|D|}
\]
Probabilistic Symbolic Execution

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

<table>
<thead>
<tr>
<th>( i )</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>( p_i )</td>
<td>( 1/2 )</td>
<td>( ??? )</td>
<td>( ??? )</td>
<td>( ??? )</td>
<td>( ??? )</td>
</tr>
</tbody>
</table>

\[
p_i = \frac{|PC_i|}{|D|}
\]
## Probabilistic Symbolic Execution

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

<table>
<thead>
<tr>
<th>( i )</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>( PC_i )</td>
<td>( P[0] \neq G[0] )</td>
<td>( P[0] = G[0] )</td>
<td>( P[0] = G[0] )</td>
<td>( P[0] = G[0] )</td>
<td>( P[0] = G[0] )</td>
</tr>
<tr>
<td>(</td>
<td>PC_i</td>
<td>)</td>
<td>128</td>
<td>64</td>
<td></td>
</tr>
<tr>
<td>( p_i )</td>
<td>1/2</td>
<td>1/4</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[
p_i = \frac{|PC_i|}{|D|}
\]
Probabilistic Symbolic Execution

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

<table>
<thead>
<tr>
<th>$i$</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>$PC_i$</td>
<td>$P[0] \neq G[0]$</td>
<td>$P[0] = G[0]$</td>
<td>$P[0] = G[0]$</td>
<td>$P[0] = G[0]$</td>
<td>$P[0] = G[0]$</td>
</tr>
<tr>
<td>$</td>
<td>PC_i</td>
<td>$</td>
<td>128</td>
<td>64</td>
<td></td>
</tr>
<tr>
<td>$p_i$</td>
<td>1/2</td>
<td>1/4</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

$$p_i = \frac{|PC_i|}{|D|}$$
## Probabilistic Symbolic Execution

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

<table>
<thead>
<tr>
<th>i</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>$</td>
<td>PC_i</td>
<td>$</td>
<td>128</td>
<td>64</td>
<td>32</td>
</tr>
<tr>
<td>$p_i$</td>
<td>1/2</td>
<td>1/4</td>
<td>1/8</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

$$p_i = \frac{|PC_i|}{|D|}$$
Assume binary 4 digit PIN. $P$ has 4 bits, $G$ has 4 bits. $|D| = 2^8 = 256$.

<table>
<thead>
<tr>
<th>$i$</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>$PC_i$</td>
<td>$P[0] \neq G[0]$</td>
<td>$P[0] = G[0]$</td>
<td>$P[0] = G[0]$</td>
<td>$P[0] = G[0]$</td>
<td>$P[0] = G[0]$</td>
</tr>
<tr>
<td>$</td>
<td>PC_i</td>
<td>$</td>
<td>128</td>
<td>64</td>
<td>32</td>
</tr>
<tr>
<td>$p_i$</td>
<td>$1/2$</td>
<td>$1/4$</td>
<td>$1/8$</td>
<td>$1/16$</td>
<td></td>
</tr>
</tbody>
</table>

\[
p_i = \frac{|PC_i|}{|D|}
\]
Probabilistic Symbolic Execution

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

<table>
<thead>
<tr>
<th>$i$</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>$PC_i$</td>
<td>$P[0] \neq G[0]$</td>
<td>$P[0] = G[0]$</td>
<td>$P[0] = G[0]$</td>
<td>$P[0] = G[0]$</td>
<td>$P[0] = G[0]$</td>
</tr>
<tr>
<td>$</td>
<td>PC_i</td>
<td>$</td>
<td>128</td>
<td>64</td>
<td>32</td>
</tr>
<tr>
<td>$p_i$</td>
<td>$1/2$</td>
<td>$1/4$</td>
<td>$1/8$</td>
<td>$1/16$</td>
<td>$1/16$</td>
</tr>
</tbody>
</table>

\[ p_i = \frac{|PC_i|}{|D|} \]
Probabilistic Symbolic Execution

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

<table>
<thead>
<tr>
<th>( i )</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>(</td>
<td>PC_i</td>
<td>)</td>
<td>128</td>
<td>64</td>
<td>32</td>
</tr>
<tr>
<td>( p_i )</td>
<td>1/2</td>
<td>1/4</td>
<td>1/8</td>
<td>1/16</td>
<td>1/16</td>
</tr>
</tbody>
</table>

\[
p_i = \frac{|PC_i|}{|D|}
\]
Probabilistic Symbolic Execution

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

<table>
<thead>
<tr>
<th>$i$</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>$PC_i$</td>
<td>$P[0] \neq G[0]$</td>
<td>$P[0] = G[0]$</td>
<td>$P[0] = G[0]$</td>
<td>$P[0] = G[0]$</td>
<td>$P[0] = G[0]$</td>
</tr>
<tr>
<td>$</td>
<td>PC_i</td>
<td>$</td>
<td>128</td>
<td>64</td>
<td>32</td>
</tr>
<tr>
<td>$p_i$</td>
<td>1/2</td>
<td>1/4</td>
<td>1/8</td>
<td>1/16</td>
<td>1/16</td>
</tr>
</tbody>
</table>

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

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

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

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

Model Counting
- Boolean Logic
- Strings
- Linear Integer Arithmetic
Problem: how to solve path constraints?
Problem: how to solve path constraints?
Satisfiability Modulo Theories (SMT) Solvers
Problem: how to solve path constraints?

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, ...
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
- Kroening. Decision Procedures - an algorithmic point of view. TCS 2008
- Barrett. CVC4. CAV 2011
### Davis-Putnam-Logemann-Loveland (DPLL) Algorithm

A decision procedure for satisfiability of Boolean formulas in conjunctive normal form (CNF-SAT).
## 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.
**Function**: DPLL($\phi$)

**Input**: CNF formula $\phi$ over $n$ variables

**Output**: true or false, the satisfiability of $\phi$

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

Input : CNF formula $\phi$ over $n$ variables

Output : true or false, the satisfiability of $F$

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

\[ \phi = \{ x \lor y \neg x \lor z, z \lor w, x, y \lor v \} \]
DPLL uses **Unit Propagation**.

\[
\phi = \{x \lor y \neg x \lor z, z \lor w, x, y \lor v\}
\]

\[
\phi' = \{z, x, y \lor v\}
\]
DPLL Execution Example

\{z, x, y \lor v\}
DPLL Execution Example

\[
\{z, x, y \lor v\}
\]

\[
\Downarrow
\]

\[
x \mapsto F
\]

\[
\Downarrow
\]

UNSAT \ \{z, F, y \lor v\}
DPLL Execution Example

\[
\{z, x, y \lor v\} \\
\left\{\begin{array}{c}
x \mapsto F \\
 x \mapsto T
\end{array}\right. \\
\text{UNSAT} \quad \{z, F, y \lor v\} \quad \{z, T, y \lor v\}
\]
DPLL Execution Example

\{z, x, y \lor v\}
\xrightarrow{\ x \mapsto F\ }
\{z, F, y \lor v\} \quad \{z, T, y \lor v\}
\xrightarrow{\ z \mapsto F\ }
\text{UNSAT} \quad \{F, T, y \lor v\}

Result: $\phi$ is satisfiable.
DPLL Execution Example

\[
\begin{align*}
\{z, x, y \lor v\} \\
/ \\
x \leftrightarrow F & \quad x \leftrightarrow T \\
/ \\
\text{UNSAT} & \quad \{z, F, y \lor v\} \quad \{z, T, y \lor v\} \\
/ \\
z \leftrightarrow F & \quad z \leftrightarrow T \\
/ \\
\text{UNSAT} & \quad \{F, T, y \lor v\} \quad \{T, T, y \lor v\}
\end{align*}
\]

Result: \(\phi\) is satisfiable.
DPLL Execution Example

\[
\{z, x, y \lor v\}
\]

\[
\begin{array}{c}
\text{UNSAT} \\
x \mapsto F \\
\text{unsat} \\
x \mapsto T \\
\end{array}
\]

\[
\begin{array}{c}
\text{unsat} \\
z \mapsto F \\
z \mapsto T \\
\end{array}
\]

\[
\begin{array}{c}
\text{unsat} \\
y \mapsto F \\
\end{array}
\]

\[
\{T, T, F \lor v\}
\]

Result: \(\phi\) is satisfiable.
DPLL Execution Example

\[ \{z, x, y \vee v\} \]

\[ x \mapsto F \quad x \mapsto T \]

\[ \text{UNSAT} \quad \{z, F, y \vee v\} \quad \{z, T, y \vee v\} \]

\[ z \mapsto F \quad z \mapsto T \]

\[ \text{UNSAT} \quad \{F, T, y \vee v\} \quad \{T, T, y \vee v\} \]

\[ y \mapsto F \]

\[ \{T, T, F \vee v\} \]

\[ v \mapsto F \]

\[ \text{UNSAT} \quad \{T, T, F \vee F\} \]

Result: \( \phi \) is satisfiable.
DPLL Execution Example

\[
\begin{align*}
\{z, x, y \lor v\} & \\
x \mapsto F & x \mapsto T & \\
\text{UNSAT} & \{z, F, y \lor v\} & \{z, T, y \lor v\} & \\
z \mapsto F & z \mapsto T & \\
\text{UNSAT} & \{F, T, y \lor v\} & \{T, T, y \lor v\} & \\
y \mapsto F & \\
\{T, T, F \lor v\} & \\
\text{v \mapsto F} & \text{v \mapsto T} & \\
\text{UNSAT} & \{T, T, F \lor v\} & \{T, T, F \lor T\} & \text{SAT}
\end{align*}
\]
DPLL Execution Example

{z, x, y \lor v} \\
\quad / \\
x \mapsto F \quad x \mapsto T \\
\quad / \\
\text{UNSAT} \quad \{z, F, y \lor v\} \quad \{z, T, y \lor v\} \\
\quad / \\
z \mapsto F \quad z \mapsto T \\
\quad / \\
\text{UNSAT} \quad \{F, T, y \lor v\} \quad \{T, T, y \lor v\} \\
\quad / \\
y \mapsto F \quad y \mapsto T \\
\quad / \\
\quad {T, T, F \lor v} \quad \{T, T, T \lor v\} \text{ SAT} \\
\quad / \\
v \mapsto F \quad v \mapsto T \\
\quad / \\
\text{UNSAT} \quad \{T, T, F \lor F\} \quad \{T, T, F \lor T\} \text{ SAT}
DPLL Execution Example

\[ \{z, x, y \lor v\} \]

\[ x \mapsto F \quad x \mapsto T \]

\[ \text{UNSAT} \quad \{z, F, y \lor v\} \quad \{z, T, y \lor v\} \]

\[ z \mapsto F \quad z \mapsto T \]

\[ \text{UNSAT} \quad \{F, T, y \lor v\} \quad \{T, T, y \lor v\} \]

\[ y \mapsto F \quad y \mapsto T \]

\[ \{T, T, F \lor v\} \quad \{T, T, T \lor v\} \quad \text{SAT} \]

\[ v \mapsto F \quad v \mapsto T \]

\[ \text{UNSAT} \quad \{T, T, F \lor F\} \quad \{T, T, F \lor T\} \quad \text{SAT} \]

Result: \( \phi \) is satisfiable.
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
- Probabilistic
  - Geldenhuys. Probabilistic symbolic execution. ISSTA 2012
Symbolic Execution

- Summarizes program executions with path constraints.
## Symbolic Execution

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

- Probabilistic
  - Geldenhuys. Probabilistic symbolic execution. ISSTA 2012
Outline

Symbolic Execution
  Software Verification
  Symbolic Execution
  Probabilistic Symbolic Execution
  SMT Solvers

Side Channel Analysis
  Background and Information Theory
  Via Probabilistic Symbolic Execution

Model Counting
  Boolean Logic
  Strings
  Linear Integer Arithmetic
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?

<table>
<thead>
<tr>
<th>How’s the weather?</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Direct Channel:</strong> Go outside and look up.</td>
</tr>
<tr>
<td>But, I’m too busy working on my MAE.</td>
</tr>
<tr>
<td><strong>Side Channel:</strong> Did Bo ride his bike today?</td>
</tr>
</tbody>
</table>
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

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

```c
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.

```c
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.

```c
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.

```c
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.

```c
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

- Pasquale Malacaria. Assessing security threats of looping constructs. POPL 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
- Boris Kopf. Automatically deriving information-theoretic bounds for adaptive side-channel attacks. JCS 2011
- Thomas S. Messerges. Power Analysis Attacks of Modular Exponentiation in Smartcards, CHES 2002
Quantitative Information Flow

A Conceptual Framework

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

A Conceptual Framework

Let C be a program with inputs $I \in \mathcal{I}$ and observables $O \in \mathcal{O}$

- C is deterministic.
- $\mathcal{I} \sim U(\text{min}, \text{max})$

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

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

Example:

- $f$ outputs last 4 digits of $C\#$
- $f(n) = n \mod 10000$
- $f(0000\ 0000\ 0000\ 6789) = 6789 = f(1111\ 1111\ 1111\ 6789)$
- $0000\ 0000\ 0000\ 6789 \sim 1111\ 1111\ 1111\ 6789$
Quantitative Information Flow

A Conceptual Framework

Let $C$ be a program with inputs $I \in \mathcal{I}$ and observables $O \in \mathcal{O}$

$C$ is deterministic.

$\mathcal{I} \sim U(\text{min}, \text{max})$

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 $CC#$

$f(n) = n \mod 10000$

$f(0000\ 0000\ 0000\ 6789) = 6789 = f(1111\ 1111\ 1111\ 6789)$

$0000\ 0000\ 0000\ 6789 \sim 1111\ 1111\ 1111\ 6789$
Let C be a program with inputs $I \in \mathcal{I}$ and observables $O \in \mathcal{O}$

C is deterministic.

$I \sim U(\min, \max)$

Then there exists a function $f : \mathcal{I} \rightarrow \mathcal{O}$ 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 Conceptual Framework

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

Then there exists a function \( f : \mathcal{I} \rightarrow \mathcal{O} \) 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 \( CC\# \)

- \( f(n) = n \mod 10000 \)
- \( f(0000 \ 0000 \ 0000 \ 6789) = 6789 \)
Let C be a program with inputs $I \in \mathcal{I}$ and observables $O \in \mathcal{O}$

- C is deterministic.
- $I \sim U(\min, \max)$

Then there exists a function $f : \mathcal{I} \rightarrow \mathcal{O}$ 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 Conceptual Framework

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

Then there exists a function $f : \mathcal{I} \rightarrow \mathcal{O}$ 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

\[ H = \sum p_i \log \frac{1}{p_i} \]
Information Theory

Claude Shannon

\[ H = \sum p_i \log \frac{1}{p_i} \]
Information Theory

Claude Shannon

Information Theory

Claude Shannon


\[ H = \sum p_i \log \frac{1}{p_i} \]
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

<table>
<thead>
<tr>
<th>S = {0, 1, 2, 3, ..., 254, 255}</th>
</tr>
</thead>
<tbody>
<tr>
<td>How many bits needed to distinguish x, y ∈ S?</td>
</tr>
</tbody>
</table>
Logarithm gives the necessary number of bits

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

How many bits needed to distinguish \( x, y \in S \)? \( \log_2(256) = 8 \)
Information Theory Intuition

Logarithm gives the necessary number of bits

<table>
<thead>
<tr>
<th>$S = {0, 1, 2, 3, \ldots, 254, 255}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>How many bits needed to distinguish $x, y \in S$? $\log_2(256) = 8$</td>
</tr>
</tbody>
</table>

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 $S_i, S_j \subseteq S$?

$$\log_2(32) = \log_2(8) = 3$$

$$\log_2(32) = \log_2\left(\frac{|S_i|}{|S_j|}\right) - 1 = \log_2\left(\frac{1}{p(S_i)}\right)$$
Information Theory Intuition

Logarithm gives the necessary number of bits

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

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

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 distinguish \( x, y \in S \)? \( \log_2(256) = 8 \)

What about a partition?

\[ S_0 = \{0, \ldots, 31\}, \quad S_1 = \{32, \ldots, 63\}, \]
**Information Theory Intuition**

Logarithm gives the necessary number of bits

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

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

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 distinguish $$x, y \in S$$? $$\log_2(256) = 8$$

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 distinguish \(x, y \in S\)? \(\log_2(256) = 8\)

**What about a partition?**

\[ S_0 = \{0, \ldots, 31\}, \quad S_1 = \{32, \ldots, 63\}, \ldots, \quad S_8 = \{224, \ldots, 255\} \]

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 distinguish \( x, y \in S? \) \( \log_2(256) = 8 \)

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 distinguish \( x, y \in S \)? \( \log_2(256) = 8 \)

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 distinguish \( x, y \in S? \) \( \log_2(256) = 8 \)

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 distinguish \( x, y \in S? \log_2(256) = 8 \)

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

<table>
<thead>
<tr>
<th>City</th>
<th>Weather Condition</th>
<th>Probability</th>
<th>Entropy (H)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Seattle</td>
<td>Always Raining</td>
<td>p_rain = 1, p_son = 0</td>
<td>H = 0</td>
</tr>
<tr>
<td>Seattle</td>
<td>Coin Flip</td>
<td>p_rain = 1/2, p_son = 1/2</td>
<td>H = 1</td>
</tr>
<tr>
<td>Santa Barbara</td>
<td>Almost Always Beautiful!</td>
<td>p_rain = 1/10, p_son = 9/10</td>
<td>H = 0.4960</td>
</tr>
</tbody>
</table>
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

\[ p_{\text{rain}} = 1, \ p_{\text{sun}} = 0 \]
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

\( p_{\text{rain}} = 1, \ p_{\text{sun}} = 0 \quad H = 0 \)
# 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 \textit{"surprise"}.

## Seattle Weather, Always Raining

\( p_{\text{rain}} = 1, \; p_{\text{sun}} = 0 \)

\( H = 0 \)

## Costa Rica Weather, Coin Flip

\( p_{\text{rain}} = \frac{1}{2}, \; p_{\text{sun}} = \frac{1}{2} \)
# 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

\( p_{\text{rain}} = 1, \ p_{\text{sun}} = 0 \quad \quad \quad \quad \quad \quad H = 0 \)

### Costa Rica Weather, Coin Flip

\( p_{\text{rain}} = \frac{1}{2}, \ p_{\text{sun}} = \frac{1}{2} \quad \quad \quad \quad \quad \quad H = 1 \)
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

\( p_{\text{rain}} = 1, p_{\text{sun}} = 0 \quad H = 0 \)

Costa Rica Weather, Coin Flip

\( p_{\text{rain}} = \frac{1}{2}, p_{\text{sun}} = \frac{1}{2} \quad H = 1 \)

Santa Barbara Weather, Almost Always Beautiful!

\( p_{\text{rain}} = \frac{1}{10}, p_{\text{sun}} = \frac{9}{10} \)
Information Theory Intuition

### Information Entropy

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

\( p_{\text{rain}} = 1, p_{\text{sun}} = 0 \) \( H = 0 \)

### Costa Rica Weather, Coin Flip

\( p_{\text{rain}} = \frac{1}{2}, p_{\text{sun}} = \frac{1}{2} \) \( H = 1 \)

### Santa Barbara Weather, Almost Always Beautiful!

\( p_{\text{rain}} = \frac{1}{10}, p_{\text{sun}} = \frac{9}{10} \) \( H = 0.4960 \)
Outline

Symbolic Execution
  Software Verification
  Symbolic Execution
  Probabilistic Symbolic Execution
  SMT Solvers

Side Channel Analysis
  Background and Information Theory
  Via Probabilistic Symbolic Execution

Model Counting
  Boolean Logic
  Strings
  Linear Integer Arithmetic
### 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:
    \[
    H = \sum p_i \log \frac{1}{p_i}
    \]
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, ...
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.
## 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[4])
for (i = 0; i < 4; i++)
    if (guess[i] != PIN[i])
        return false
return true

$P$: PIN, $G$: guess

$o_i = \text{lines of code}$
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
<table>
<thead>
<tr>
<th>i</th>
<th>PC&lt;sub&gt;i&lt;/sub&gt;</th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$P[0] \neq G[0]$</td>
<td>$P[0] = G[0]$</td>
<td>$P[0] = G[0]$</td>
<td>$P[0] = G[0]$</td>
<td>$P[0] = G[0]$</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>true</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>return</th>
<th>false</th>
<th>false</th>
<th>false</th>
<th>false</th>
<th>true</th>
</tr>
</thead>
<tbody>
<tr>
<td>$</td>
<td>PC&lt;sub&gt;i&lt;/sub&gt;</td>
<td>$</td>
<td>128</td>
<td>64</td>
<td>32</td>
</tr>
<tr>
<td>$p_i$</td>
<td>1/2</td>
<td>1/4</td>
<td>1/8</td>
<td>1/16</td>
<td>1/16</td>
</tr>
<tr>
<td>$o_i$</td>
<td>3</td>
<td>5</td>
<td>7</td>
<td>9</td>
<td>10</td>
</tr>
<tr>
<td>$i$</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>-----</td>
<td>-------</td>
<td>-------</td>
<td>-------</td>
<td>-------</td>
<td>-------</td>
</tr>
<tr>
<td>return</td>
<td>false</td>
<td>false</td>
<td>false</td>
<td>false</td>
<td>true</td>
</tr>
<tr>
<td>$</td>
<td>PC_i</td>
<td>$</td>
<td>128</td>
<td>64</td>
<td>32</td>
</tr>
<tr>
<td>$p_i$</td>
<td>1/2</td>
<td>1/4</td>
<td>1/8</td>
<td>1/16</td>
<td>1/16</td>
</tr>
<tr>
<td>$o_i$</td>
<td>3</td>
<td>5</td>
<td>7</td>
<td>9</td>
<td>10</td>
</tr>
</tbody>
</table>

A measure of program vulnerability $H = \frac{1}{2} \log p_i = 1.875$. The expected amount of information that an adversary can gain in 1 guess.
<table>
<thead>
<tr>
<th>i</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
</table>
| $PC_i$ | $P[0] \neq G[0]$ | $P[0] = G[0]$ \[
P[1] \neq G[1]
\] | $P[0] = G[0]$ \[
\]| $P[0] = G[0]$ \[
\]| $P[0] = G[0]$ \[
\] |
<p>| 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 |</p>
<table>
<thead>
<tr>
<th>i</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>$PC_i$</td>
<td>$P[0] \neq G[0]$</td>
<td>$P[0] = G[0]$</td>
<td>$P[0] = G[0]$</td>
<td>$P[0] = G[0]$</td>
<td>$P[0] = G[0]$</td>
</tr>
<tr>
<td>return</td>
<td>false</td>
<td>false</td>
<td>false</td>
<td>false</td>
<td>true</td>
</tr>
<tr>
<td>$</td>
<td>PC_i</td>
<td>$</td>
<td>128</td>
<td>64</td>
<td>32</td>
</tr>
<tr>
<td>$p_i$</td>
<td>1/2</td>
<td>1/4</td>
<td>1/8</td>
<td>1/16</td>
<td>1/16</td>
</tr>
<tr>
<td>$o_i$</td>
<td>3</td>
<td>5</td>
<td>7</td>
<td>9</td>
<td>10</td>
</tr>
<tr>
<td>i</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>----</td>
<td>----</td>
<td>----</td>
<td>----</td>
<td>----</td>
<td>----</td>
</tr>
<tr>
<td>$PC_i$</td>
<td>$P[0] \neq G[0]$</td>
<td>$P[0] = G[0]$</td>
<td>$P[0] = G[0]$</td>
<td>$P[0] = G[0]$</td>
<td>$P[0] = G[0]$</td>
</tr>
<tr>
<td>return</td>
<td>false</td>
<td>false</td>
<td>false</td>
<td>false</td>
<td>true</td>
</tr>
<tr>
<td>$</td>
<td>PC_i</td>
<td>$</td>
<td>128</td>
<td>64</td>
<td>32</td>
</tr>
<tr>
<td>$p_i$</td>
<td>$1/2$</td>
<td>$1/4$</td>
<td>$1/8$</td>
<td>$1/16$</td>
<td>$1/16$</td>
</tr>
<tr>
<td>$o_i$</td>
<td>3</td>
<td>5</td>
<td>7</td>
<td>9</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>$PC_i$</td>
<td>$P[0] \neq G[0]$</td>
<td>$P[0] = G[0]$</td>
<td>$P[0] = G[0]$</td>
<td>$P[0] = G[0]$</td>
<td>$P[0] = G[0]$</td>
</tr>
<tr>
<td>return</td>
<td>false</td>
<td>false</td>
<td>false</td>
<td>false</td>
<td>true</td>
</tr>
<tr>
<td>$</td>
<td>PC_i</td>
<td>$</td>
<td>128</td>
<td>64</td>
<td>32</td>
</tr>
<tr>
<td>$p_i$</td>
<td>$1/2$</td>
<td>$1/4$</td>
<td>$1/8$</td>
<td>$1/16$</td>
<td>$1/16$</td>
</tr>
<tr>
<td>$o_i$</td>
<td>3</td>
<td>5</td>
<td>7</td>
<td>9</td>
<td>10</td>
</tr>
</tbody>
</table>

\[
H = \sum p_i \log \frac{1}{p_i} = 1.8750
\]
<table>
<thead>
<tr>
<th>i</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>$PC_i$</td>
<td>$P[0] \neq G[0]$</td>
<td>$P[0] = G[0]$</td>
<td>$P[0] = G[0]$</td>
<td>$P[0] = G[0]$</td>
<td>$P[0] = G[0]$</td>
</tr>
<tr>
<td>return</td>
<td>false</td>
<td>false</td>
<td>false</td>
<td>false</td>
<td>true</td>
</tr>
<tr>
<td>$</td>
<td>PC_i</td>
<td>$</td>
<td>128</td>
<td>64</td>
<td>32</td>
</tr>
<tr>
<td>$p_i$</td>
<td>$1/2$</td>
<td>$1/4$</td>
<td>$1/8$</td>
<td>$1/16$</td>
<td>$1/16$</td>
</tr>
<tr>
<td>$o_i$</td>
<td>3</td>
<td>5</td>
<td>7</td>
<td>9</td>
<td>10</td>
</tr>
</tbody>
</table>

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

A measure of program vulnerability

$H = \text{expected amount of information that an adversary can gain in 1 guess.}$
Side Channel Analysis

A more secure 4 digit PIN verification function:

```java
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 observables:

- $o_0$ = perfect match,
- $o_1$ = not perfect match.

$p(o_0) = \frac{1}{16}$,
$p(o_1) = \frac{15}{16}$.

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

39/66
Side Channel Analysis

A more secure 4 digit PIN verification function:

```java
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 observables: $o_0$ = perfect match, $o_1$ = not perfect match.
Side Channel Analysis

A more secure 4 digit PIN verification function:

```java
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 observables: \( o_0 = \) perfect match, \( o_1 = \) not perfect match.

\[
p(o_0) = 1/16, p(o_1) = 15/16
\]
A more secure 4 digit PIN verification function:

```java
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 observables: \( o_0 = \) perfect match, \( o_1 = \) not perfect match.

\[
p(o_0) = \frac{1}{16}, \quad p(o_1) = \frac{15}{16}
\]

\[
H_{\text{secure}} = 0.33729
\]
Side Channel Analysis

A more secure 4 digit PIN verification function:

```java
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 observables: $o_0 = $ perfect match, $o_1 = $ not perfect match.

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

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

- Observe non-functional aspects of computation to learn information.
- Probabilistic 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 computation to learn information.
- Probabilistic 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.
Recall the classic (boolean) SAT problem

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

Example:

$$\phi = (x \lor y) \land (\neg x \lor z) \land (z \lor w) \land x \land (y \lor v)$$
Recall the classic (boolean) SAT problem

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

Example:

$$
\phi = (x \lor y) \land (\neg x \lor z) \land (z \lor w) \land x \land (y \lor v)
$$

$\phi$ is satisfiable by setting

$$(x, y, z, w, v) = (T, F, T, F, T).$$
Recall the classic (boolean) SAT problem

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

Example:

\[
\phi = (x \lor y) \land (\neg x \lor z) \land (z \lor w) \land x \land (y \lor v)
\]

\( \phi \) is satisfiable by setting

\[
(x, y, z, w, v) = (T, F, T, F, T).
\]

A satisfying assignment is called a model for \( \phi \).
The **model counting problem**

Given a formula $\phi$ over some theory (Boolean, LIA, Strings, ...)

how many models are there for $\phi$?
Model Counting

The **model counting problem**

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

how many models are there for \( \phi \)?

**Difficulty of Model Counting**

Model counting is “at least as hard” than satisfiability check.
The **model counting problem**

Given a formula $\phi$ over some theory (Boolean, LIA, Strings, ...)

how many models are there for $\phi$?

---

**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

- Pugh. Counting Solutions to Presburger Formulas: How and Why. 1994
- Parker. An Automata-Theoretic Algorithm for Counting Solutions to Presburger Formulas. Compiler Construction 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
- Birnbaum. The good old Davis-Putnam procedure helps counting models. JAIR 1999
Outline

Symbolic Execution
  Software Verification
  Symbolic Execution
  Probabilistic Symbolic Execution
  SMT Solvers

Side Channel Analysis
  Background and Information Theory
  Via Probabilistic Symbolic Execution

Model Counting
  Boolean Logic
  Strings
  Linear Integer Arithmetic
Model Counting Boolean SAT

$\varphi$ has 6 models.

Truth table method is $\theta(2^n)$.
Model Counting Boolean SAT

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>z</th>
<th>w</th>
<th>v</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>T</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>T</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>T</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>T</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
</tbody>
</table>

$\phi$ has 6 models.
### Model Counting Boolean SAT

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>z</th>
<th>w</th>
<th>v</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>T</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>T</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>T</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
</tbody>
</table>

\( \phi \) has 6 models.

Truth table method is \( \theta(2^n) \).
DPLL can be converted into a procedure for \#CNF-SAT.

**Function** : DPLL(\(\phi, t\))

**Input** : CNF formula \(\phi\) over \(n\) variables; \(t \in \mathbb{Z}\)

**Output** : \(\#\phi\), the model count of \(\phi\)

**begin**

UnitPropagate(\(\phi\))

if \(\phi\) has false clause **return** \(false\)

if all clauses of \(\phi\) satisfied **return** \(true\)

\(x \leftarrow\) SelectBranchVariable(\(\phi\))

**return** DPLL(\(\phi[x \mapsto true], t - 1\)) \(\lor\) DPLL(\(\phi[x \mapsto true], t - 1\))

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

**Function**: DPLL($\phi$, $t$)

**Input**: CNF formula $\phi$ over $n$ variables; $t \in \mathbb{Z}$

**Output**: $\#\phi$, the model count of $\phi$

**begin**

UnitPropagate($\phi$)

if $\phi$ has false clause then return false

if all clauses of $\phi$ satisfied then return true

x $\leftarrow$ SelectBranchVariable($\phi$)

return DPLL($\phi[x \mapsto true]$, $t - 1$) $\lor$ DPLL($\phi[x \mapsto true]$, $t - 1$)

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

**Function**: DPLL(\(\phi, t\))

**Input**: CNF formula \(\phi\) over \(n\) variables; \(t \in \mathbb{Z}\)

**Output**: \(\#\phi\), the model count of \(\phi\)

begin
    UnitPropagate(\(\phi\))
    if \(\phi\) has false clause then return false
    if all clauses of \(\phi\) satisfied then return true
    \(x \leftarrow\) SelectBranchVariable(\(\phi\))
    return DPLL(\(\phi[x \mapsto true]\), \(t - 1\)) \(\lor\) DPLL(\(\phi[x \mapsto true]\), \(t - 1\))
end
DPLL can be converted into a procedure for \#CNF-SAT.

**Function** : DPLL(\(\phi, t\))
**Input** : CNF formula \(\phi\) over \(n\) variables; \(t \in \mathbb{Z}\)
**Output** : \(#\phi\), the model count of \(\phi\)

begin
    UnitPropagate(\(\phi\))
    if \(\phi\) has false clause then return 0
    if all clauses of \(\phi\) satisfied then return true
    \(x \leftarrow\) SelectBranchVariable(\(\phi\))
    return DPLL(\(\phi[x \mapsto true]\), \(t - 1\)) \(\lor\) DPLL(\(\phi[x \mapsto true]\), \(t - 1\))
end
DPLL can be converted into a procedure for \#CNF-SAT.

**Function** : DPLL($\phi, t$)
**Input** : CNF formula $\phi$ over $n$ variables; $t \in \mathbb{Z}$
**Output** : $\#\phi$, the model count of $\phi$

begin
    UnitPropagate($\phi$)
    if $\phi$ has false clause then return 0
    if all clauses of $\phi$ satisfied then return $true$
    $x \leftarrow$ SelectBranchVariable($\phi$)
    return DPLL($\phi[x \mapsto true], t - 1$) $\lor$ DPLL($\phi[x \mapsto true], t - 1$)
end
DPLL can be converted into a procedure for \#CNF-SAT.

**Function** : DPLL($\phi, t$)

**Input** : CNF formula $\phi$ over $n$ variables; $t \in \mathbb{Z}$

**Output** : $\#\phi$, the model count of $\phi$

**begin**

UnitPropagate($\phi$)

if $\phi$ has false clause then return 0

if all clauses of $\phi$ satisfied then return true

$x \leftarrow$ SelectBranchVariable($\phi$)

return DPLL($\phi[x \mapsto true], t - 1$) $\lor$ DPLL($\phi[x \mapsto true], t - 1$)

**end**
DPLL can be converted into a procedure for \( \# \text{CNF-SAT} \).

**Function** : \( \text{DPLL}(\phi, t) \)

**Input** : CNF formula \( \phi \) over \( n \) variables; \( t \in \mathbb{Z} \)

**Output** : \( \#\phi \), the model count of \( \phi \)

begin

UnitPropagate(\( \phi \))

if \( \phi \) has false clause then return 0

if all clauses of \( \phi \) satisfied then return \( 2^t \)

\( x \leftarrow \text{SelectBranchVariable}(\phi) \)

return \( \text{DPLL}(\phi[x \mapsto true], t - 1) \lor \text{DPLL}(\phi[x \mapsto true], t - 1) \)

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

**Function** : DPLL(\(\phi, t\))

**Input** : CNF formula \(\phi\) over \(n\) variables; \(t \in \mathbb{Z}\)

**Output** : \#\(\phi\), the model count of \(\phi\)

begin
UnitPropagate(\(\phi\))
if \(\phi\) has false clause then return 0
if all clauses of \(\phi\) satisfied then return \(2^t\)
\(x \leftarrow\) SelectBranchVariable(\(\phi\))
return DPLL(\(\phi[x \mapsto true]\), \(t - 1\)) + DPLL(\(\phi[x \mapsto true]\), \(t - 1\))
end
Counting with DPLL

\[ \phi = \{ x \lor y, \neg x \lor z, z \lor w, x, y \lor v \}, \ n = 5 \]

\{ z, x, y \lor v \} t = 5
Counting with DPLL

\[ \phi = \{ x \lor y, \neg x \lor z, z \lor w, x, y \lor v \}, n = 5 \]

\[ \{ z, x, y \lor v \} t = 5 \]

\[ x \mapsto F \]

\[ 0 \quad \{ z, F, y \lor v \} t = 4 \]
Counting with DPLL

\[ \phi = \{ x \lor y, \neg x \lor z, z \lor w, x, y \lor v \}, \ n = 5 \]

\[ \{ z, x, y \lor v \} t = 5 \]

\[ x \mapsto F \quad x \mapsto T \]

\[ \{ z, F, y \lor v \} t = 4 \]

\[ \{ z, T, y \lor v \} t = 4 \]

Result: 0 + 0 + 0 + 2 + 4 = 6 models
Counting with DPLL

\[ \phi = \{ x \lor y, \neg x \lor z, z \lor w, x, y \lor v \}, \ n = 5 \]

\{z, x, y \lor v\} t = 5

\begin{align*}
&x \mapsto F \\
&x \mapsto T
\end{align*}

\begin{align*}
0 \ \{z, F, y \lor v\} t = 4 & & \{z, T, y \lor v\} t = 4 \\
\end{align*}

\begin{align*}
&z \mapsto F \\
&0 \ \{F, T, y \lor v\} t = 3
\end{align*}
Counting with DPLL

\[ \phi = \{x \lor y, \neg x \lor z, z \lor w, x, y \lor v\}, \ n = 5 \]

\[ \{z, x, y \lor v\} t = 5 \]

\[ x \mapsto F \]

\[ x \mapsto T \]

\[ \{z, F, y \lor v\} t = 4 \]

\[ \{z, T, y \lor v\} t = 4 \]

\[ z \mapsto F \]

\[ z \mapsto T \]

\[ \{F, T, y \lor v\} t = 3 \]

\[ \{T, T, y \lor v\} t = 3 \]

Result: 0 + 0 + 0 + 2 + 4 = 6 models
Counting with DPLL

\[ \phi = \{ x \lor y, \neg x \lor z, z \lor w, x, y \lor v \} \], \ n = 5

\{z, x, y \lor v\} t = 5

\xrightarrow{x \leftrightarrow F} \{z, F, y \lor v\} t = 4

\xrightarrow{x \leftrightarrow T} \{z, T, y \lor v\} t = 4

\{z \leftrightarrow F\}

\{F, T, y \lor v\} t = 3

\xrightarrow{y \leftrightarrow F} \{T, T, F \lor v\} t = 2

\{T, T, y \lor v\} t = 3

\{T, T, y \lor v\} t = 3

\{T, T, y \lor v\} t = 3
Counting with DPLL

\[ \phi = \{ x \lor y, \neg x \lor z, z \lor w, x, y \lor v \}, \ n = 5 \]

\[
\begin{align*}
\{z, x, y \lor v\} & t = 5 \\
x \mapsto F & \quad x \mapsto T \\
0 \ {z, F, y \lor v} & t = 4 \\
z \mapsto F & \quad z \mapsto T \\
0 \ {F, T, y \lor v} & t = 3 \\
y \mapsto F & \\
{z, T, y \lor v} & t = 4 \\
z \mapsto F & \quad z \mapsto T \\
0 \ {T, T, y \lor v} & t = 3 \\
y \mapsto F & \\
{z, T, F \lor v} & t = 2 \\
z \mapsto F & \\
{F, T, F \lor v} & t = 3 \\
0 \ {z, F, y \lor v} & t = 4 \\
z \mapsto F & \quad z \mapsto T \\
0 \ {T, T, F \lor v} & t = 2 \\
0 \ {T, T, F \lor v} & t = 1
\end{align*}
\]

Result: 0 + 0 + 0 + 2 + 4 = 6 models
Counting with DPLL

\[ \phi = \{ x \lor y, \neg x \lor z, z \lor w, x, y \lor v \} \], \ n = 5

\[
\begin{align*}
\{z, x, y \lor v\} t &= 5 \\
&\quad \downarrow \quad x \mapsto F \quad x \mapsto T \\
0 \quad \{z, F, y \lor v\} t &= 4 & \quad \{z, T, y \lor v\} t &= 4 \\
&\quad \downarrow \quad z \mapsto F \quad z \mapsto T \\
0 \quad \{F, T, y \lor v\} t &= 3 & \quad \{T, T, y \lor v\} t &= 3 \\
&\quad \downarrow \quad y \mapsto F \\
\{T, T, F \lor v\} t &= 2 \\
&\quad \downarrow \quad v \mapsto F \quad v \mapsto T \\
0 \quad \{T, T, F \lor F\} t &= 1 & 2^1 &= 2 \quad \{T, T, F \lor T\} t &= 1
\end{align*}
\]
Counting with DPLL

\[ \phi = \{ x \lor y, \neg x \lor z, z \lor w, x, y \lor v \}, \ n = 5 \]

\[ \{ z, x, y \lor v \} t = 5 \]

\[ x \mapsto F \]

\[ x \mapsto T \]

\[ 0 \ \{ z, F, y \lor v \} t = 4 \]

\[ \{ z, T, y \lor v \} t = 4 \]

\[ z \mapsto F \]

\[ z \mapsto T \]

\[ 0 \ \{ F, T, y \lor v \} t = 3 \]

\[ \{ T, T, y \lor v \} t = 3 \]

\[ y \mapsto F \]

\[ y \mapsto T \]

\[ \{ T, T, F \lor v \} t = 2 \]

\[ 2^2 = 4 \]

\[ \{ T, T, T \lor v \} t = 2 \]

\[ v \mapsto F \]

\[ v \mapsto T \]

\[ 0 \ \{ T, T, F \lor F \} t = 1 \]

\[ 2^1 = 2 \]

\[ \{ T, T, F \lor T \} t = 1 \]

Result: 0 + 0 + 0 + 2 + 4 = 6 models
Counting with DPLL

\( \phi = \{ x \lor y, \neg x \lor z, z \lor w, x, y \lor v \}, n = 5 \)

\( \{ z, x, y \lor v \} t = 5 \)
\( x \mapsto F \quad x \mapsto T \)

\( 0 \quad \{ z, F, y \lor v \} t = 4 \)
\( \{ z, T, y \lor v \} t = 4 \)
\( z \mapsto F \quad z \mapsto T \)

\( 0 \quad \{ F, T, y \lor v \} t = 3 \)
\( \{ T, T, y \lor v \} t = 3 \)
\( y \mapsto F \quad y \mapsto T \)

\( \{ T, T, F \lor v \} t = 2 \quad 2^2 = 4 \quad \{ T, T, T \lor v \} t = 2 \)
\( v \mapsto F \quad v \mapsto T \)

\( 0 \quad \{ T, T, F \lor F \} t = 1 \quad 2^1 = 2 \quad \{ T, T, F \lor T \} t = 1 \)

Result: \( 0 + 0 + 0 + 2 + 4 = 6 \) models
Generating functions are a way to compactly represent (possibly infinite) sequences. 

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

\[ g(z) = a_0 z^0 + a_1 z^1 + a_2 z^2 + a_3 z^3 + a_4 z^4 + \ldots \]
**Generating functions** are a way to compactly represent (possibly infinite) sequences.

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

\[ g(z) = a_0 z^0 + a_1 z^1 + a_2 z^2 + a_3 z^3 + \cdots \]
Generating functions are a way to compactly represent (possibly infinite) sequences.

\[ g(z) = \frac{1}{(1 - z)^3} \]
Generating functions are a way to compactly represent (possibly infinite) sequences.

\[ g(z) = \frac{1}{(1 - z)^3} = \sum_{k=0}^{\infty} a_k z^k \]
Generating functions are a way to compactly represent (possibly infinite) sequences.

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

\[ g(z) = 1z^0 + 3z^1 + 6z^2 + 10z^3 + 15z^4 + \ldots \]
Generating functions are a way to compactly represent (possibly infinite) sequences.

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

\[ g(z) = 1z^0 + 3z^1 + 6z^2 + 10z^3 + 15z^4 + \ldots \]

\[ g(z) = a_0 z^0 + a_1 z^1 + a_2 z^2 + a_3 z^3 + a_4 z^4 + \ldots \]
Outline

Symbolic Execution
  Software Verification
  Symbolic Execution
  Probabilistic Symbolic Execution
  SMT Solvers

Side Channel Analysis
  Background and Information Theory
  Via Probabilistic Symbolic Execution

Model Counting
  Boolean Logic
  Strings
  Linear Integer Arithmetic
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: $4 < \text{Length}(X) < 10$
Model Counting Strings

A formula over the theory of strings can involve

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

\[ X \in (0\mid(1(01^*0)^*1))^* \]

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 \(L\) encodes

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

\[ g(z) = \]

52 / 66
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

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

\[ g(z) = 1z^0 \]

<table>
<thead>
<tr>
<th>( k )</th>
<th>( X )</th>
<th>( a_k )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>( \varepsilon )</td>
<td>1</td>
</tr>
</tbody>
</table>
Regular Expressions

\[ X \in (0|(10^*0)^*1))^* \]

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

\[ a_k = \left| \{ s : s \in \mathcal{L}, \text{len}(s) = k \} \right| \]

\[ g(z) = 1z^0 + 1z^1 \]

<table>
<thead>
<tr>
<th>( k )</th>
<th>( X )</th>
<th>( a_k )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>( \varepsilon )</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
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 \( L \) encodes

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

\[ g(z) = 1z^0 + 1z^1 + 1z^2 \]

<table>
<thead>
<tr>
<th>( k )</th>
<th>( X )</th>
<th>( a_k )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>( \varepsilon )</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>11</td>
<td>1</td>
</tr>
</tbody>
</table>
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 \( L \) encodes

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

\[ g(z) = 1z^0 + 1z^1 + 1z^2 + 1z^3 \]

<table>
<thead>
<tr>
<th>( k )</th>
<th>( X )</th>
<th>( a_k )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>( \varepsilon )</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>11</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>110</td>
<td>1</td>
</tr>
</tbody>
</table>

52 / 66
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

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

\[ g(z) = 1z^0 + 1z^1 + 1z^2 + 1z^3 + 3z^4 \]

<table>
<thead>
<tr>
<th>( k )</th>
<th>( X )</th>
<th>( a_k )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>( \varepsilon )</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>11</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>110</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>1001, 1100, 1111</td>
<td>3</td>
</tr>
</tbody>
</table>
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

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

\[ g(z) = 1z^0 + 1z^1 + 1z^2 + 1z^3 + 3z^4 + 5z^5 + \ldots \]

<table>
<thead>
<tr>
<th>( k )</th>
<th>( X )</th>
<th>( a_k )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>( \varepsilon )</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>11</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>110</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>1001, 1100, 1111</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>10010, 10101, 11000, 11011, 11110</td>
<td>5</td>
</tr>
</tbody>
</table>
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.

\[
\begin{align*}
\varepsilon & \mapsto 1z^0 \\
c & \mapsto 1z^1
\end{align*}
\]
Regular Expressions

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

\[
\begin{align*}
\varepsilon & \mapsto 1z^0 \\
\mathcal{C} & \mapsto 1z^1 \\
A|B & \mapsto A(z) + B(z)
\end{align*}
\]
For a regular expression constraint, GF can be derived recursively.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Correspondence</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\varepsilon$</td>
<td>$1z^0$</td>
</tr>
<tr>
<td>$c$</td>
<td>$1z^1$</td>
</tr>
<tr>
<td>$A</td>
<td>B$</td>
</tr>
<tr>
<td>$A \circ B$</td>
<td>$A(z) \times B(z)$</td>
</tr>
</tbody>
</table>
For a regular expression constraint, GF can be derived recursively.

<table>
<thead>
<tr>
<th>Expression</th>
<th>GF</th>
</tr>
</thead>
<tbody>
<tr>
<td>ε</td>
<td>$1z^0$</td>
</tr>
<tr>
<td>c</td>
<td>$1z^1$</td>
</tr>
<tr>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>A ⋄ B</td>
<td>$A(z) \times B(z)$</td>
</tr>
<tr>
<td>A*</td>
<td>$1/(1 - A(z))$</td>
</tr>
</tbody>
</table>
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

\[ X \in (0|(1(01^*0)^*1))^* \]
Regular Expressions

\[ X \in (0|1(01*0)*1))* \]
Regular Expressions

\[ X \in (0|(1(01^*0)^*1))^* \]

Generating Function:

\[ g(z) = \frac{1}{1-z-\frac{z^2}{1-z}} \]
Regular Expressions

\[ X \in (0|((1(0^*0)^*1))^* \]

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:

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

\[
g(z) = 1z^0 + 1z^1 + 1z^2 + 1z^3 + 3z^4 + 5z^5 + \ldots
\]
Deterministic Finite Automata
Deterministic Finite Automata

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

\[ X \in (0|1(01^*0)^*1))^* \]

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

\text{String counting} \equiv \text{path counting}
Deterministic Finite Automata

How to count paths of length \( k \)?
How to count paths of length $k$?

Dynamic Programming
Deterministic Finite Automata

How to count paths of length $k$?

Dynamic Programming

$$\eta_s(k)$$
How to count paths of length $k$?

Dynamic Programming

$$\eta_s(k) = \sum_{s' \rightarrow s} \eta_{s'}(k - 1)$$
Deterministic Finite Automata

How to count paths of length $k$?

Dynamic Programming

Matrix Exponentiation

$$
\eta_s(k) = \sum_{s' \rightarrow s} \eta_{s'}(k - 1)
$$
How to count paths of length $k$?

**Dynamic Programming**

**Matrix Exponentiation**

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

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

How to count paths of length $k$?

Dynamic Programming

Matrix Exponentiation

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

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

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

How to count paths of length $k$?

**Dynamic Programming**

- State $s_1$
- State $s_2$
- State $s_3$
- State $s$

**Matrix Exponentiation**

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

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

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

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

How to count paths of length $k$?

**Dynamic Programming**

**Matrix Exponentiation**

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

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

**Generating Functions**

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

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

How to count paths of length $k$?

<table>
<thead>
<tr>
<th>Dynamic Programming</th>
<th>Matrix Exponentiation</th>
<th>Generating Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s_1'$</td>
<td>$s$</td>
<td>$s_2'$</td>
</tr>
</tbody>
</table>

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

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

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

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

$(A^4)_{0,0} = 3$
### Deterministic Finite Automata

![Automaton Diagram]

**How to count paths of length \( k \)?**

<table>
<thead>
<tr>
<th>Dynamic Programming</th>
<th>Matrix Exponentiation</th>
<th>Generating Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \eta_s(k) = \sum_{s' \rightarrow s} \eta_{s'}(k - 1) )</td>
<td>( A = \begin{pmatrix} 1 &amp; 1 &amp; 0 \ 1 &amp; 0 &amp; 1 \ 0 &amp; 1 &amp; 1 \end{pmatrix} )</td>
<td>( g(z) = \frac{\det(I - zA : i, j)}{(-1)^n \det(I - zA)} )</td>
</tr>
<tr>
<td>( (A^k)_{i,j} )</td>
<td>( A = \begin{pmatrix} 1 &amp; 1 &amp; 0 \ 1 &amp; 0 &amp; 1 \ 0 &amp; 1 &amp; 1 \end{pmatrix} )</td>
<td></td>
</tr>
<tr>
<td>( (A^4)_{0,0} = 3 )</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Deterministic Finite Automata

How to count paths of length $k$?

**Dynamic Programming**

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

**Matrix Exponentiation**

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

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

**(Generating Functions**

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

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

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

Symbolic Execution
   Software Verification
   Symbolic Execution
   Probabilistic Symbolic Execution
   SMT Solvers

Side Channel Analysis
   Background and Information Theory
   Via Probabilistic Symbolic Execution

Model Counting
   Boolean Logic
   Strings
   Linear Integer Arithmetic
What is this language?

\[ X \in (0 | (101 \ast 0) \ast 1) \ast L(X) = \{ s \mid s \text{ is a binary number divisible by } 3 \} \]

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

\[ X \in (0|(1(01^*0)^*1)))*\]
What is this language?

\[ X \in (0|(1(01^*0)^*1))^* \]

\[ L(X) = \{ s | s \text{ is a binary number divisible by 3} \} \]
What is this language?

\[ X \in (0|(1(01^*0)^*1)))^* \]

\[ L(X) = \{ s | s \text{ is a binary number divisible by 3} \} \]
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.
Quantifier-Free Linear Integer Arithmetic ($\mathbb{Z}, +, <$).
Quantifier-Free Linear Integer Arithmetic ($\mathbb{Z}, +, <$).

Constraints of the form:

$$Ax < B, x \in \mathbb{Z}^n$$
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: $\left( \begin{array}{c} b_x \\ b_y \end{array} \right)$

Solution DFA for the constraint $x > y$.

\[
\begin{align*}
(0, 0), (1, 1) & \\
(0, 1) & \\
(1, 0) & \\
(0, 0) & \\
(0, 1) & \\
(1, 0) & \\
(1, 1) & \\
(1, 1) & \\
\end{align*}
\]
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 $x > y$.

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**
  - 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
## Counting Techniques for Different Theories

- **Boolean**
  - Truth Table (Brute Force)
  - DPLL
## Model Counting Summary

<table>
<thead>
<tr>
<th>Counting Techniques for Different Theories</th>
</tr>
</thead>
<tbody>
<tr>
<td>▶ Boolean</td>
</tr>
<tr>
<td>▶ Truth Table (Brute Force)</td>
</tr>
<tr>
<td>▶ DPLL</td>
</tr>
<tr>
<td>▶ Strings</td>
</tr>
<tr>
<td>▶ Regular Expression with GFs</td>
</tr>
<tr>
<td>▶ DFA with Dynamic Programming, Matrix Multiplication, GFs</td>
</tr>
</tbody>
</table>
# 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
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.