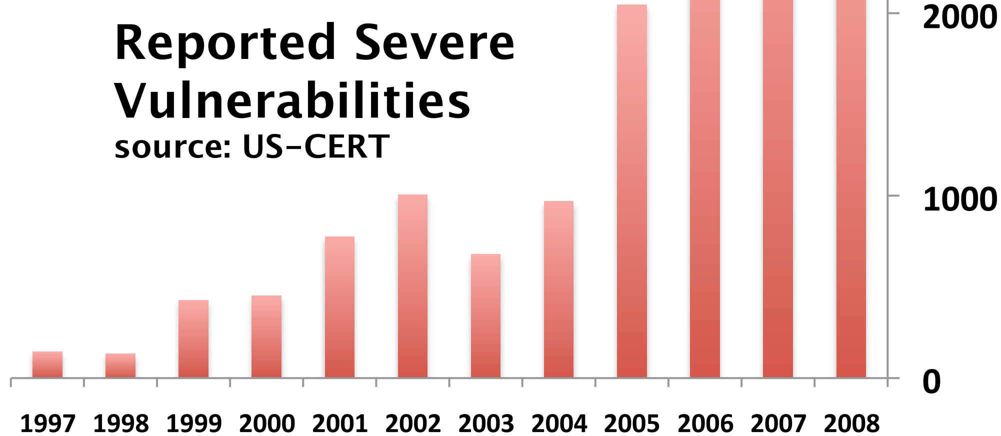
Effective Software Testing with a String-Constraint Solver

Adam Kiezun
MIT

Software Testing Aims To Find Errors Before Users (Or Hackers) Do

Goals of software testing

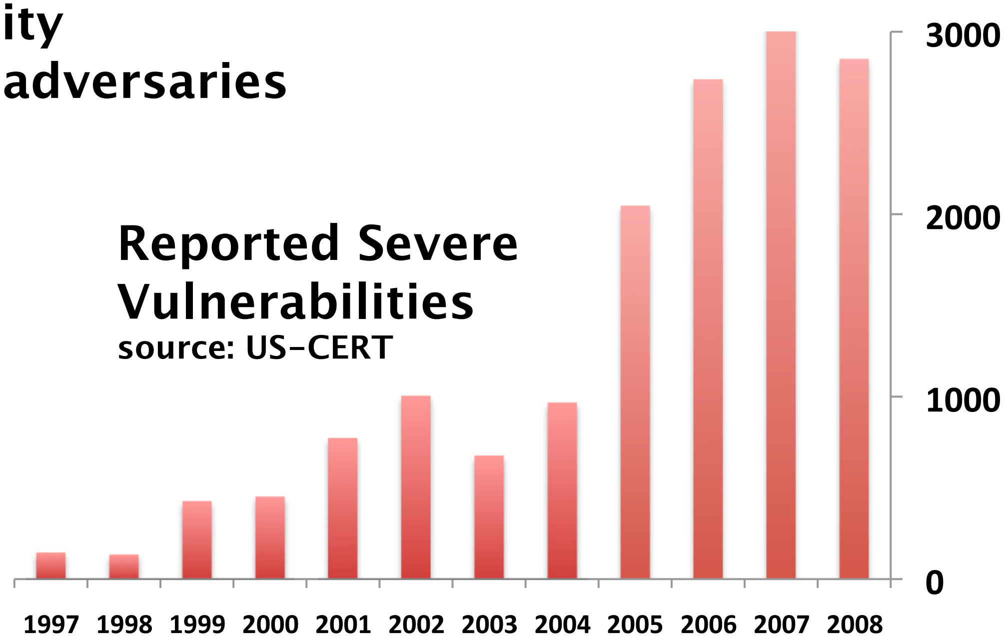
- improve quality
- protect from adversaries



Software Testing Aims To Find Errors Before Users (Or Hackers) Do

Goals of software testing

- improve quality
- protect from adversaries



Research → Tech transfer → Hackers

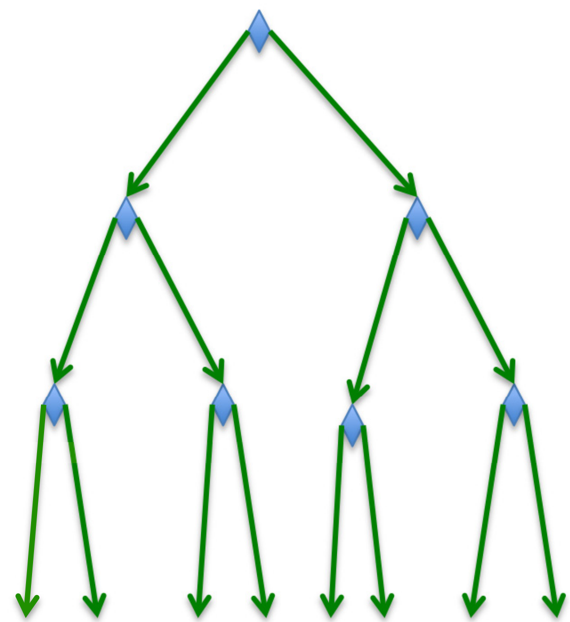
Goal: help find errors by improving testing tools

Concolic Testing Is An Effective Software Testing Methodology

Implementation-based: exploit knowledge of program code

Dynamic: observe running program using combined <u>conc</u>rete and symb<u>olic</u> execution

Constraint solver systematically enumerate execution paths



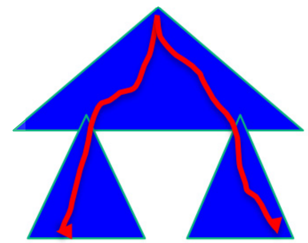
Tools: DART, CUTE, CREST, SAGE, EXE, Klee, Apollo, jFuzz

Key idea: improve effectiveness, applicability of concolic testing with a string-constraint solver

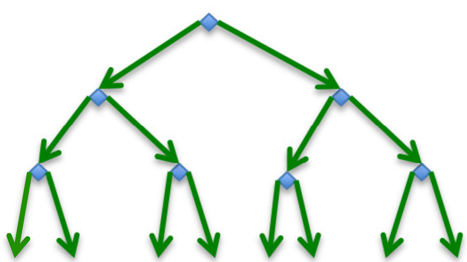
Effective Software Testing With A String-Constraint Solver

Concolic Security Testing
[ICSE'09]

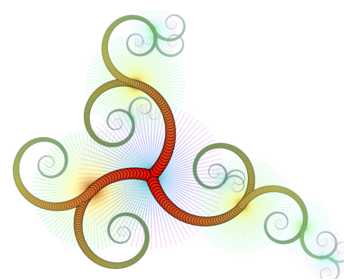
Grammar-based Concolic Testing
[PLDI'08]



Concolic Testing

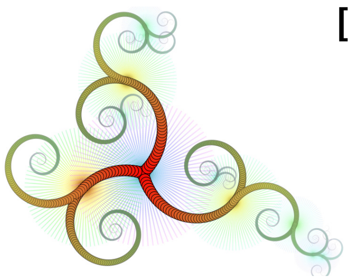


Hampi: String-Constraint Solver
[ISSTA'09]



Results Summary: String-Constraint Solver

Hampi: String-Constraint Solver [ISSTA'09]



- ✔ Novel solver for string constraints
- ✔ Supports context-free grammars, regular constraints
- ✔ Effective in concolic testing, program analysis
- ✔ Efficient: ~7x faster than a comparable solver

Results Summary: Concolic Security Testing

Concolic Security Testing [ICSE'09]

✔ Novel technique for creating SQL injection and XSS attacks on Web applications

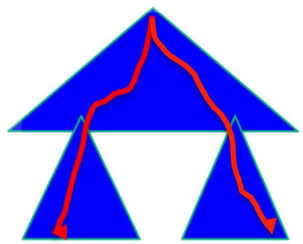
✔ Uses Hampi for grammar constraints to construct attack inputs

✔ First to create damaging second-order cross-site scripting (XSS) attacks

✔ 60 attacks (23 SQL injection, 37 XSS) on 5 PHP applications, 0 false positives

Results Summary: Grammar-based Concolic Testing

Grammar-based Concolic Testing [PLDI'08]



✔ Novel technique for testing programs with structured inputs

✔ Uses Hampi for input-format grammar constraints

✔ Improves coverage by 30-100%

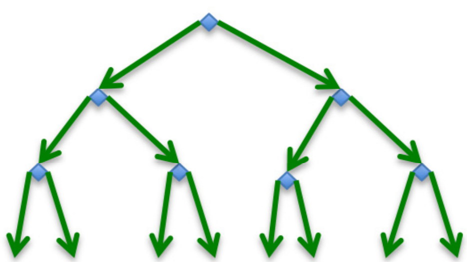
✔ 3 new infinite-loop errors

Effective Software Testing With A String-Constraint Solver

Concolic Security Testing
[ICSE'09]

Grammar-based Concolic Testing
[PLDI'08]

Concolic Testing



Hampi: String-Constraint Solver
[ISSTA'09]

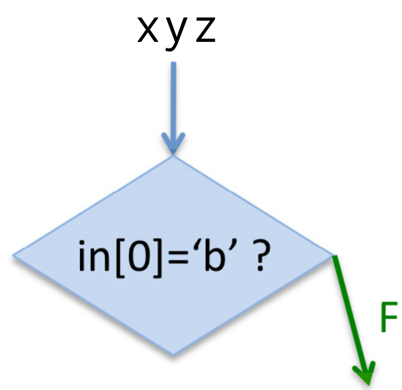
Concolic Testing Combines Dynamic Symbolic Execution, Path Enumeration

```
void main(char[] in){
  int count=0;
  if (in[0] == 'b')
    count++;
  if (in[1] == 'a')
    count++;
  if (in[2] == 'd')
    count++;
  if (count == 3)
    ERROR;
}
```

xyz

Concolic Testing Combines Dynamic Symbolic Execution, Path Enumeration

```
void main(char[] in){
  int count=0;
  if (in[0] == 'b')
    count++;
  if (in[1] == 'a')
    count++;
  if (in[2] == 'd')
    count++;
  if (count == 3)
    ERROR;
}
```

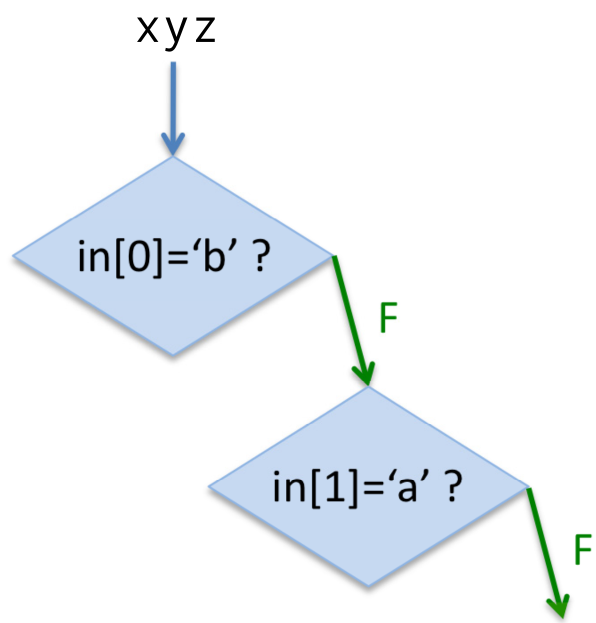


describes inputs that execute same path prefix

Path constraint: `(in[0]≠'b')`

Concolic Testing Combines Dynamic Symbolic Execution, Path Enumeration

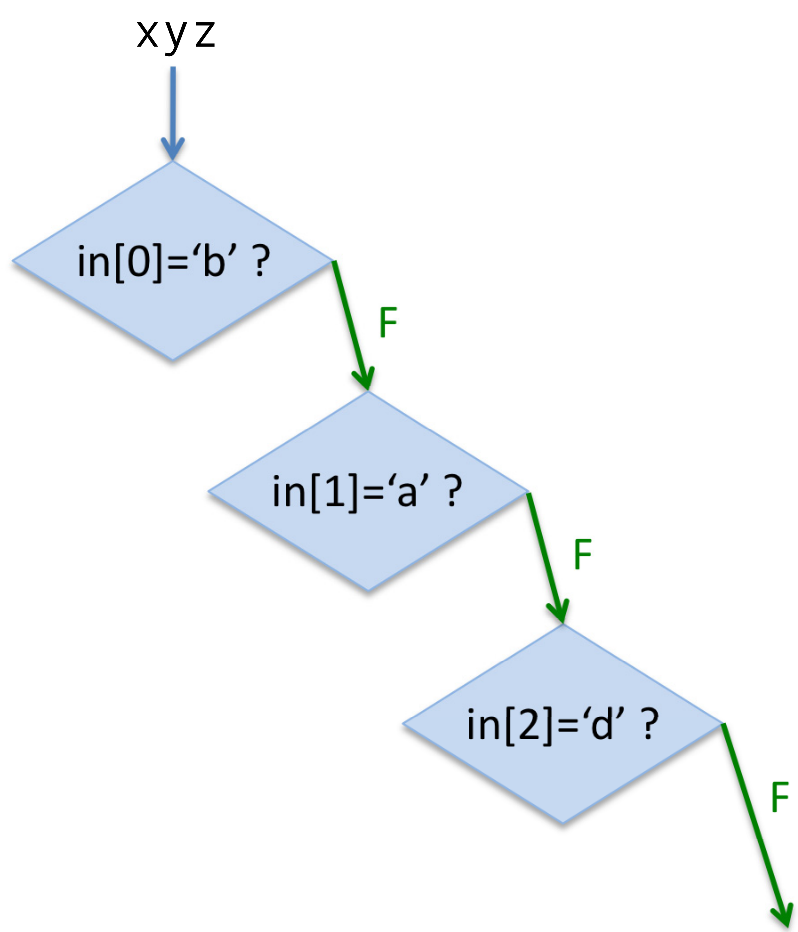
```
void main(char[] in){
  int count=0;
  if (in[0] == 'b')
    count++;
  if (in[1] == 'a')
    count++;
  if (in[2] == 'd')
    count++;
  if (count == 3)
    ERROR;
}
```



Path constraint: $(in[0] \neq 'b') \wedge (in[1] \neq 'a')$

Concolic Testing Combines Dynamic Symbolic Execution, Path Enumeration

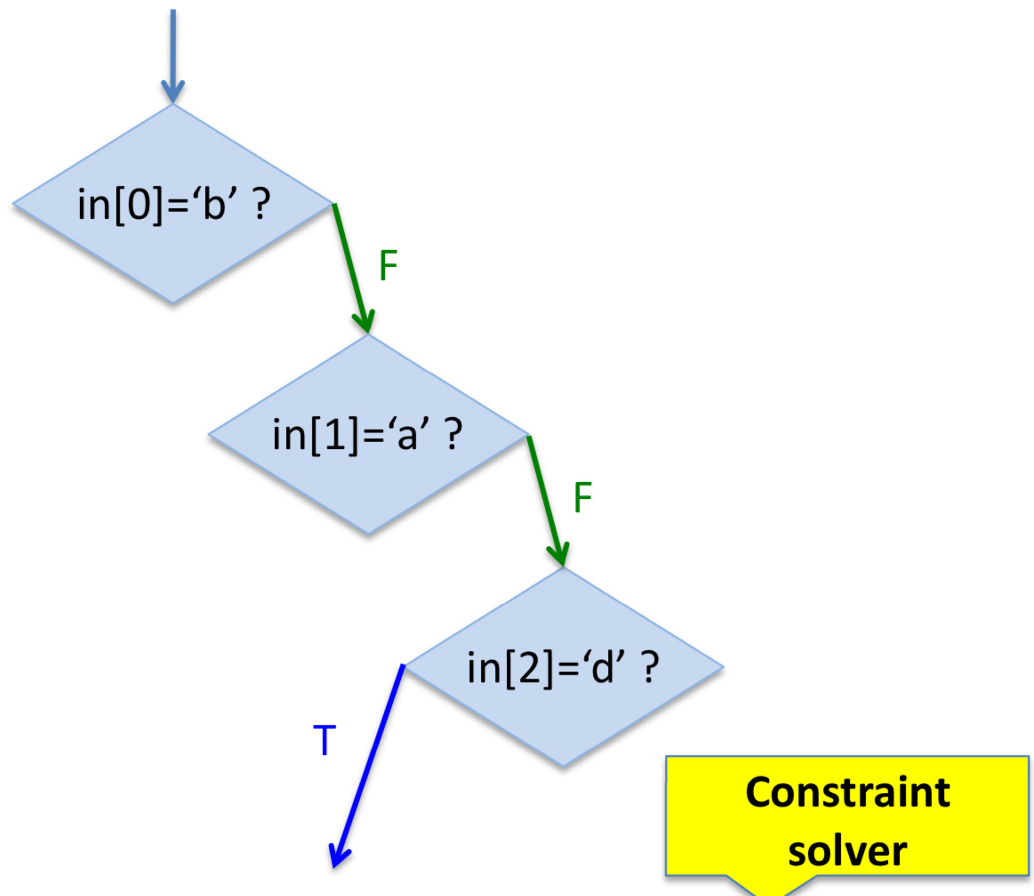
```
void main(char[] in){
  int count=0;
  if (in[0] == 'b')
    count++;
  if (in[1] == 'a')
    count++;
  if (in[2] == 'd')
    count++;
  if (count == 3)
    ERROR;
}
```



Path constraint: $(in[0] \neq 'b') \land (in[1] \neq 'a') \land (in[2] \neq 'd')$

Concolic Testing Combines Dynamic Symbolic Execution, Path Enumeration

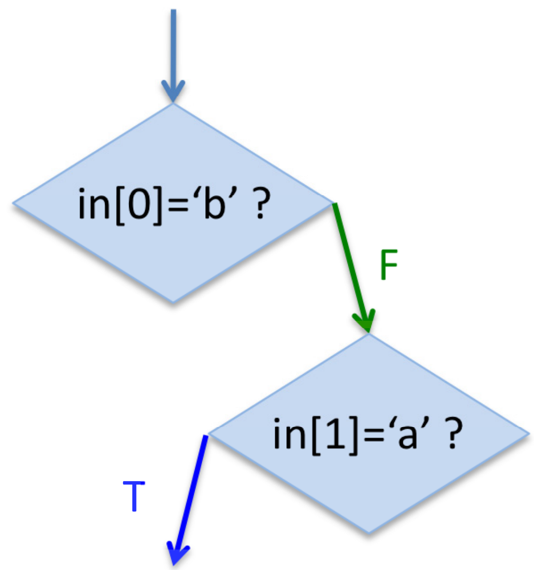
```
void main(char[] in){
  int count=0;
  if (in[0] == 'b')
    count++;
  if (in[1] == 'a')
    count++;
  if (in[2] == 'd')
    count++;
  if (count == 3)
    ERROR;
}
```



Path constraint: (in[0]≠'b')∧(in[1]≠'a')∧(in[2]='d')➔xyd

Concolic Testing Combines Dynamic Symbolic Execution, Path Enumeration

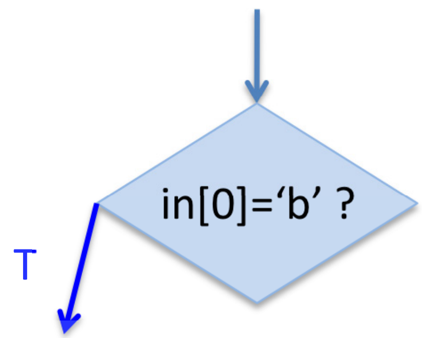
```
void main(char[] in){
  int count=0;
  if (in[0] == 'b')
    count++;
  if (in[1] == 'a')
    count++;
  if (in[2] == 'd')
    count++;
  if (count == 3)
    ERROR;
}
```



Path constraint: $(in[0] \neq 'b') \wedge (in[1] = 'a') \rightarrow$ xaz

Concolic Testing Combines Dynamic Symbolic Execution, Path Enumeration

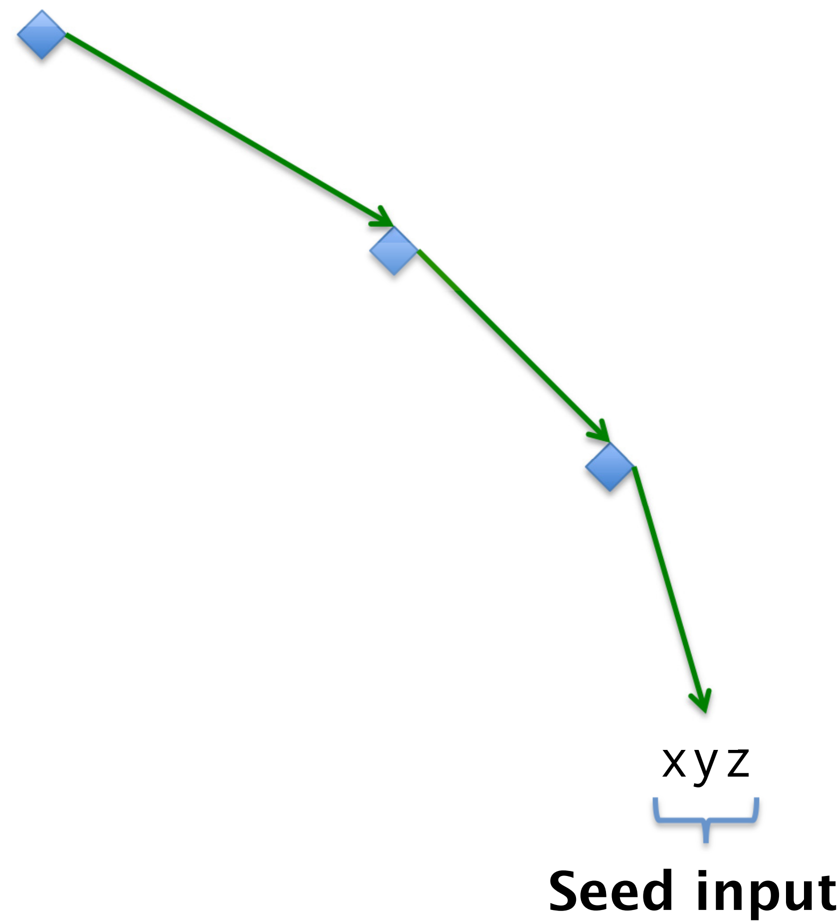
```
void main(char[] in){
  int count=0;
  if (in[0] == 'b')
    count++;
  if (in[1] == 'a')
    count++;
  if (in[2] == 'd')
    count++;
  if (count == 3)
    ERROR;
}
```



Path constraint: `(in[0]='b')`➔`byz`

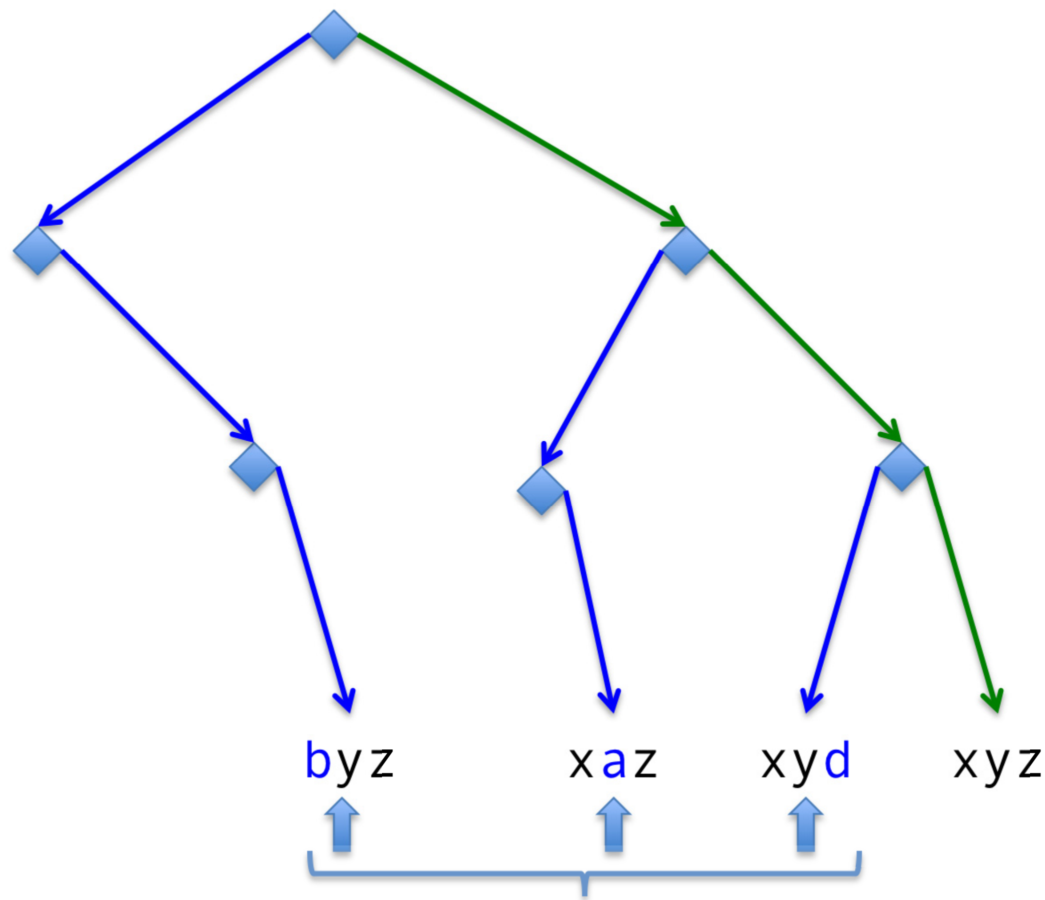
Concolic Testing Systematically Enumerates All Paths In The Program

```
void main(char[] in){
  int count=0;
  if (in[0] == 'b')
    count++;
  if (in[1] == 'a')
    count++;
  if (in[2] == 'd')
    count++;
  if (count == 3)
    ERROR;
}
```



Concolic Testing Systematically Enumerates All Paths In The Program

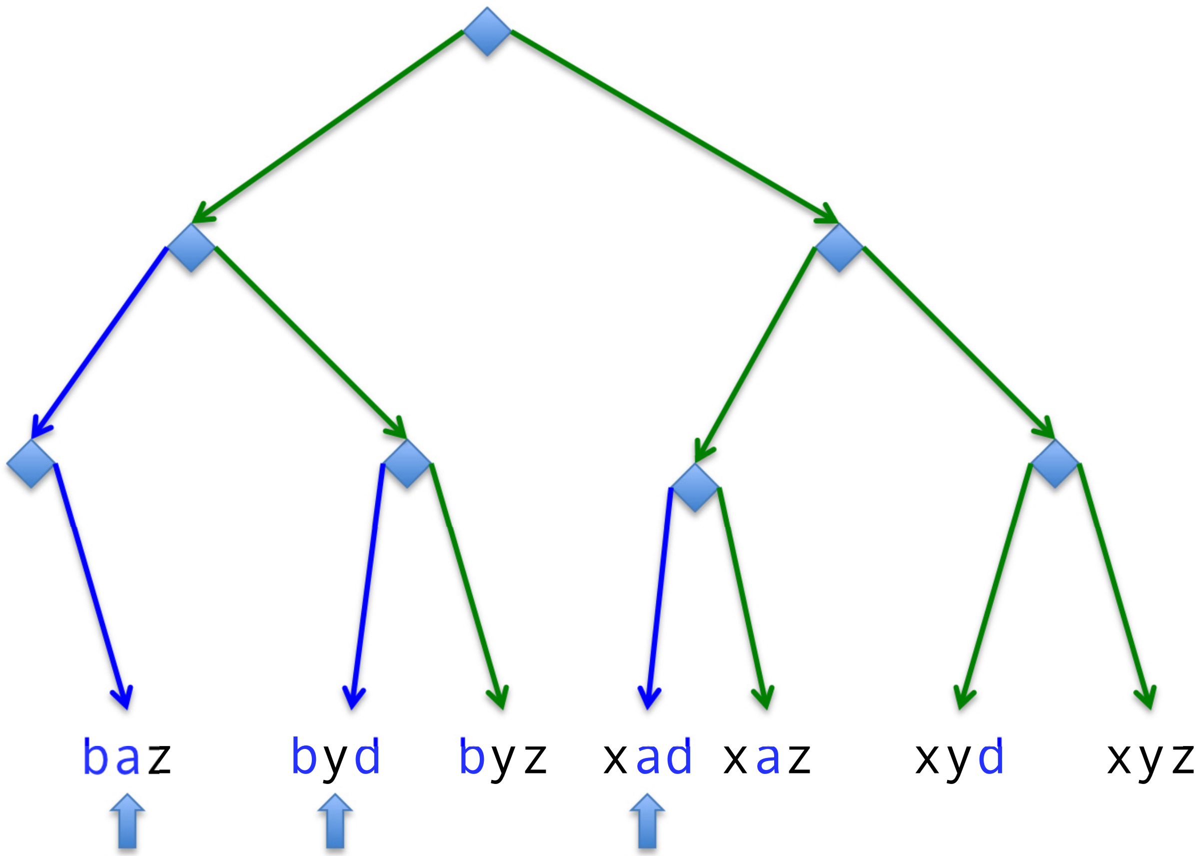
```
void main(char[] in){
  int count=0;
  if (in[0] == 'b')
    count++;
  if (in[1] == 'a')
    count++;
  if (in[2] == 'd')
    count++;
  if (count == 3)
    ERROR;
}
```



Generated inputs
(each covers a new path)

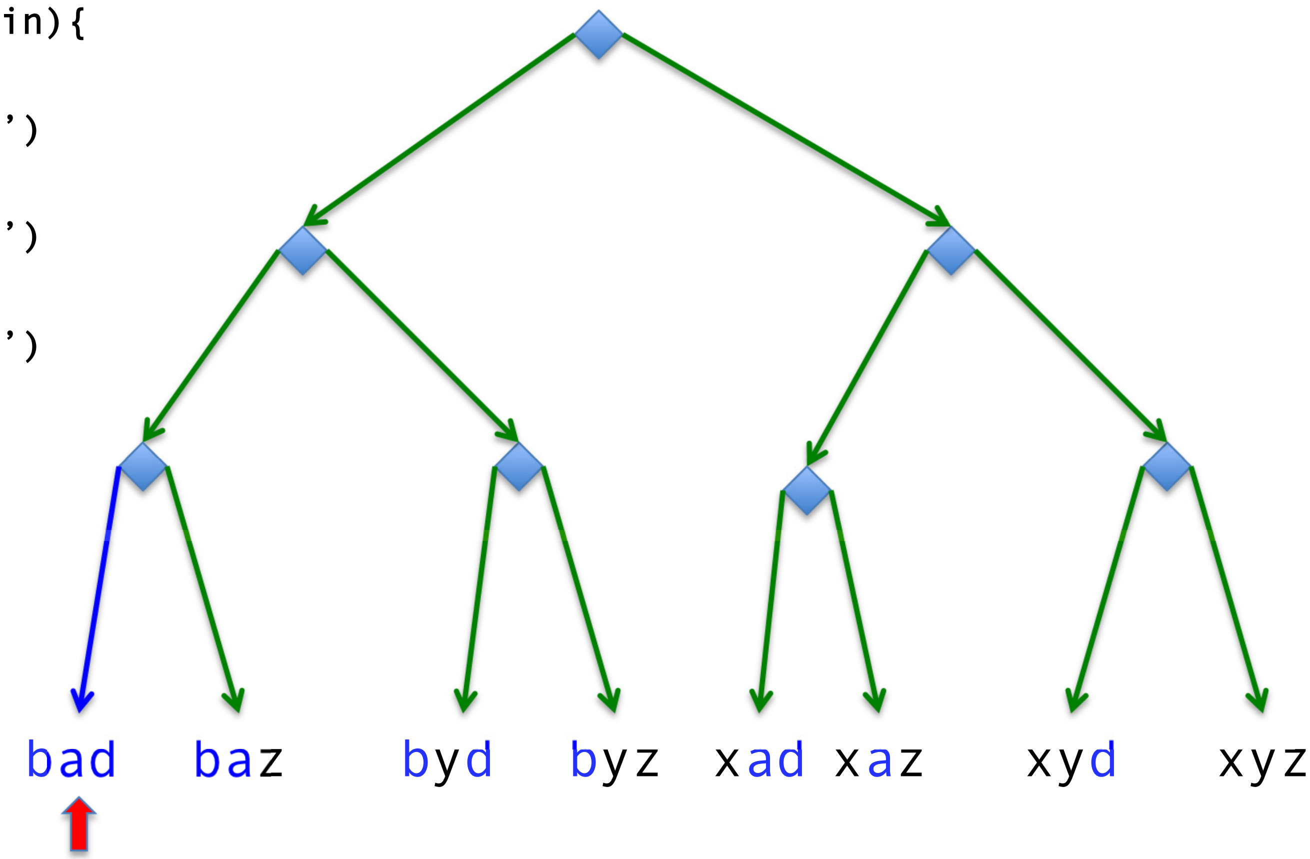
Concolic Testing Systematically Enumerates All Paths In The Program

```
void main(char[] in){
  int count=0;
  if (in[0] == 'b')
    count++;
  if (in[1] == 'a')
    count++;
  if (in[2] == 'd')
    count++;
  if (count == 3)
    ERROR;
}
```



Concolic Testing Systematically Enumerates All Paths In The Program

```
void main(char[] in){
  int count=0;
  if (in[0] == 'b')
    count++;
  if (in[1] == 'a')
    count++;
  if (in[2] == 'd')
    count++;
  if (count == 3)
    ERROR;
}
```



Concolic testing creates inputs for all program paths.

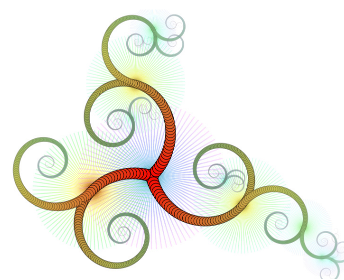
Effective Software Testing With A String-Constraint Solver

Concolic Security Testing
[ICSE'09]

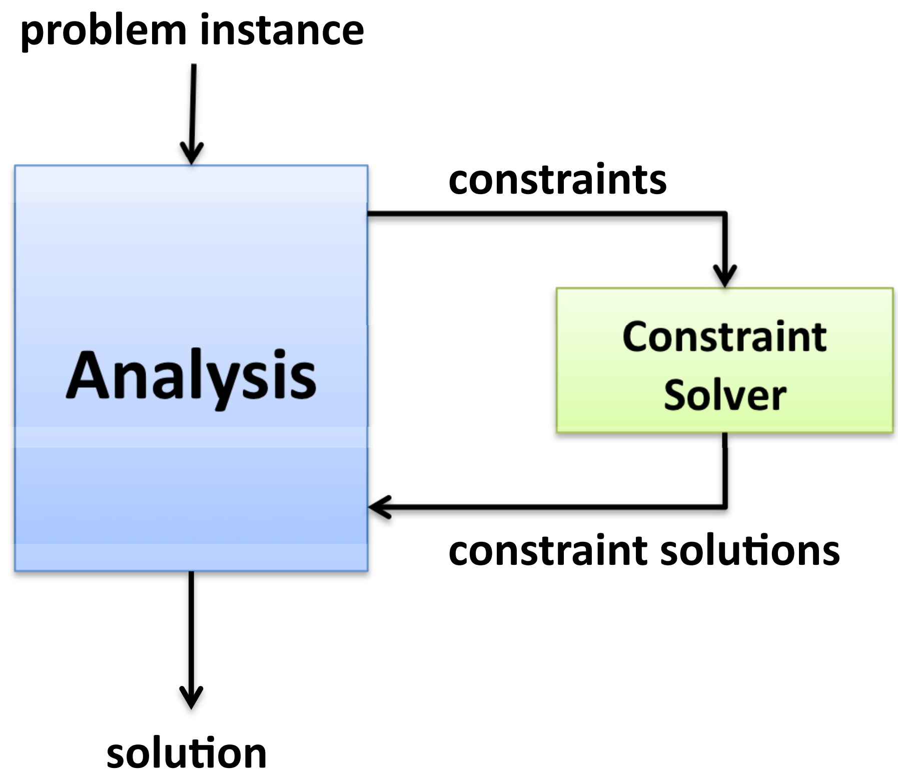
Grammar-based Concolic Testing
[PLDI'08]

Concolic Testing

Hampi: String-Constraint Solver
[ISSTA'09]



Many Program Analyses Reduce To Constraint Generation And Solving



Benefits

+ declarative formulation
+ better modularity
+ efficiency improvements

Downsides

– limited by solver's theory

Hampi: constraint solver for a theory of strings

String-Constraint Solver Finds Assignments For String Variables

Finite alphabet Σ (e.g., ASCII characters)

String variables over Σ*

```
var v
```

String constraints – language membership:

```
assert v ∈ L
```

Context-free, regular, etc.

String operations

```
concat("foo", v, "bar")
```

Hampi Uses Length Bounding To Support Context-Free Constraints

more expressive

context-free	$L_1 \cap \ldots \cap L_N$	**Undecidable**

more tractable

Hampi Uses Length Bounding To Support Context-Free Constraints

more expressive

context-free	$L_1 \cap \ldots \cap L_N$	Undecidable
regular	$R_1 \cap \ldots \cap R_N$	PSPACE-complete

more tractable

Hampi Uses Length Bounding To Support Context-Free Constraints

more expressive

context-free	$L_1 \cap \ldots \cap L_N$	Undecidable
regular	$R_1 \cap \ldots \cap R_N$	PSPACE-complete
bounded regular	$r_1 \cap \ldots \cap r_N$	NP-complete

more tractable

Hampi Uses Length Bounding To Support Context-Free Constraints

more expressive

context-free	$L_1 \cap \ldots \cap L_N$	Undecidable
regular	$R_1 \cap \ldots \cap R_N$	PSPACE-complete
bounded regular	$r_1 \cap \ldots \cap r_N$	NP-complete

more tractable

bound(any language) → bounded regular

Hampi Uses Length Bounding To Support Context-Free Constraints

more expressive

context-free	$L_1 \cap \ldots \cap L_N$	Undecidable
regular	$R_1 \cap \ldots \cap R_N$	PSPACE-complete
bounded regular	$r_1 \cap \ldots \cap r_N$	NP-complete

more tractable

bound(any language) ➔ bounded regular

Key Hampi idea: bound length of strings for high expressiveness, efficiency

Hampi Can Solve Context-Free and Regular Constraints

"Find a 4-character string `v`, such that:

- `(v)` has balanced parentheses, and
- `(v)` contains substring `()()`"

Hampi Can Solve Context-Free and Regular Constraints

String variable →

```
var v:4;

cfg E := "()" | E E | "(" E ")";

reg Ebounded := bound(E, 6);

val q := concat( ”(" , v , “)" );

assert q in Ebounded;
assert q contains “()()";
```

“Find a 4-character string v, such that:

- **(v) has balanced parentheses, and**
- **(v) contains substring ()()”**

Hampi Can Solve Context-Free and Regular Constraints

Context-free grammar

```
var v:4;

cfg E := "()" | E E | "(" E ")";

reg Ebounded := bound(E, 6);

val q := concat( ”(" , v , “)" );

assert q in Ebounded;
assert q contains “()()";
```

“Find a 4-character string v, such that:
- **(v) has balanced parentheses, and**
- **(v) contains substring ()()”**

Hampi Can Solve Context-Free and Regular Constraints

```
var v:4;

cfg E := "()" | E E | "(" E ")";

reg Ebounded := bound(E, 6);

val q := concat( ”(" , v , “)" );

assert q in Ebounded;
assert q contains “()()";
```

Regular lang. declaration (arrow to the `reg` line)

“Find a 4-character string v, such that:

- **(v) has balanced parentheses, and**
- **(v) contains substring ()()”**

Hampi Can Solve Context-Free and Regular Constraints

```
var v:4;

cfg E := "()" | E E | "(" E ")";

reg Ebounded := bound(E, 6);

val q := concat( ”(" , v , “)" );

assert q in Ebounded;
assert q contains “()()";
```

Declaration of (v)

“Find a 4-character string v, such that:

- **(v) has balanced parentheses, and**
- **(v) contains substring ()()”**

Hampi Can Solve Context-Free and Regular Constraints

```
var v:4;

cfg E := "()" | E E | "(" E ")";

reg Ebounded := bound(E, 6);

val q := concat( ”(" , v , “)" );

assert q in Ebounded;
assert q contains “()()";
```

Constraints →

“Find a 4-character string v, such that:
- **(v) has balanced parentheses, and**
- **(v) contains substring ()()”**

Hampi finds satisfying assignment v =)()(

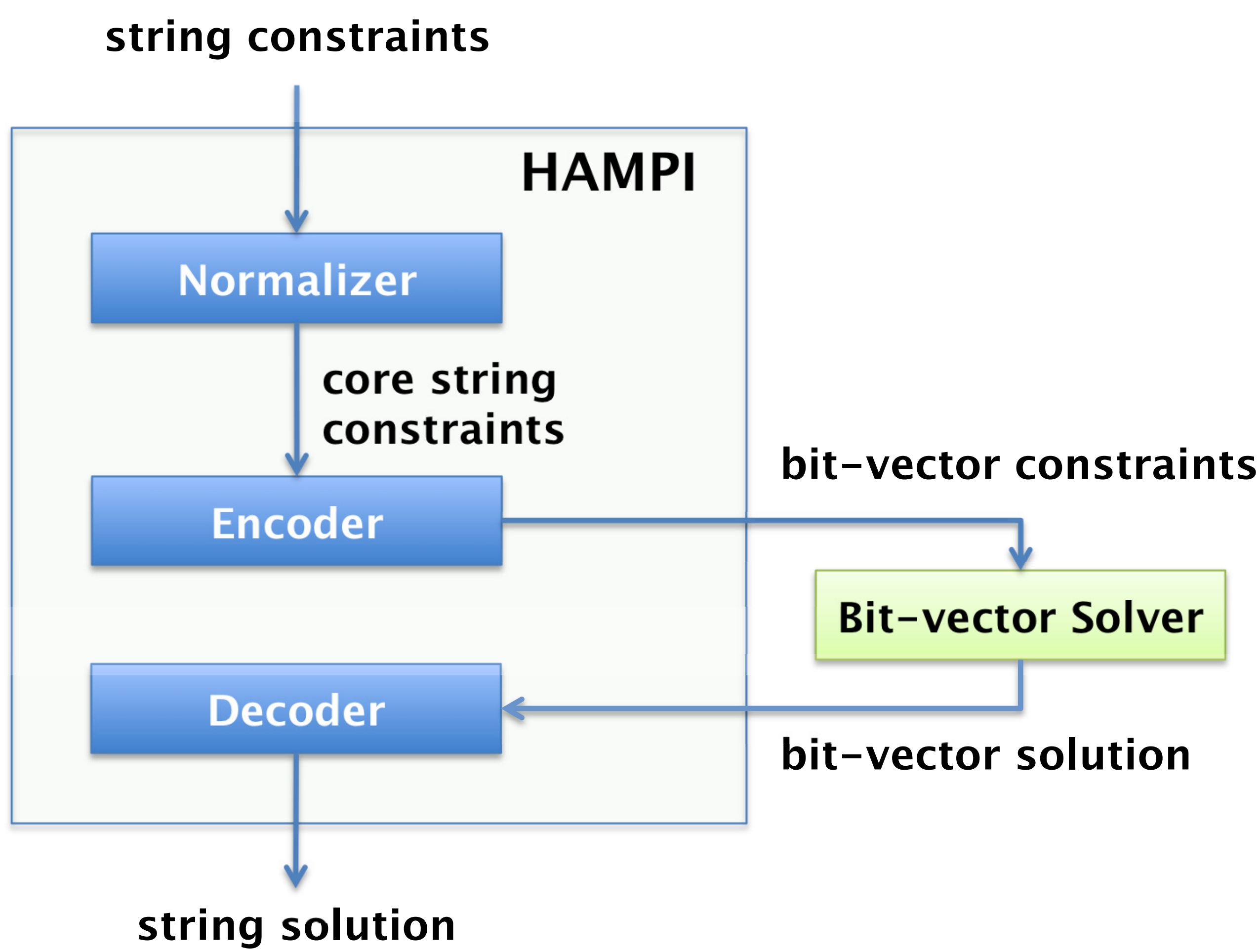
Hampi Supports Rich String Constraints

● full support

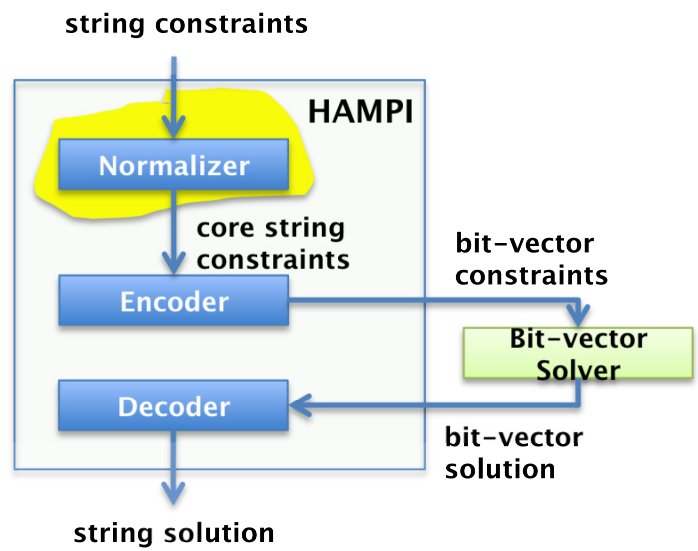
◒ partial support

	Hampi	CFGAnalyzer	Wassermann	Bjorner	Hooijmeier	Emmi	MONA	Caballero
context-free grammars	●	●	◒					
regular expressions	●	●	◒		●	●	◒	
string concatenation	●			●	●		●	●
stand-alone tool	●	●					●	
unbounded length			●		●	●	●	

Hampi Encodes String Constraints In Bit-Vector Logic



Hampi Normalizer Converts String Constraints To Core Form



Core string constraint have only regular expressions

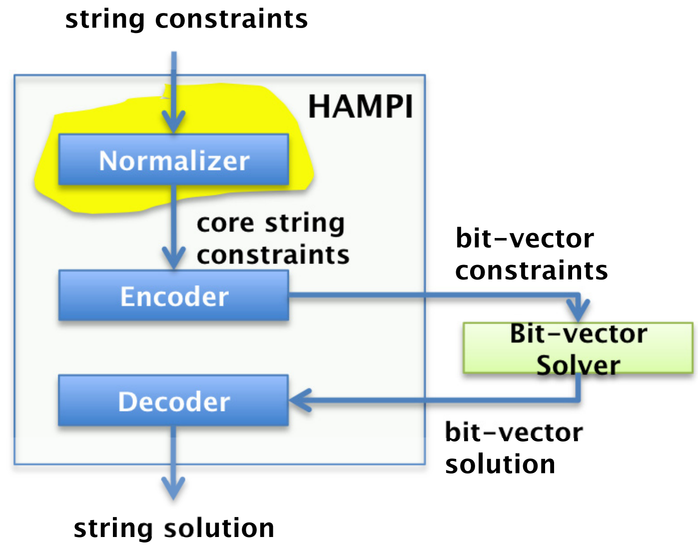
Expand grammars to regexps

- **expand nonterminals**
- **eliminate inconsistencies**
- **enumerate choices exhaustively**

`cfg` *E* `:= "(" ` *E* ` ")" | ` *E E* ` | "()";`

➡ `bound(`*E*`, 6)`

Hampi Normalizer Converts String Constraints To Core Form



Core string constraint have only regular expressions

Expand grammars to regexps

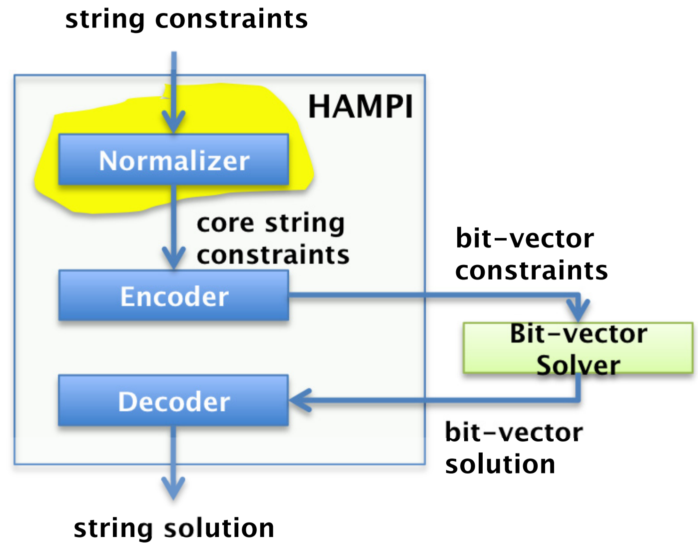
➡ **expand nonterminals**

- **eliminate inconsistencies**
- **enumerate choices exhaustively**

```
cfg E := "(" E ")” | E E | "()”;
```

$$\overbrace{E}^{6}$$

Hampi Normalizer Converts String Constraints To Core Form



Core string constraint have only regular expressions

Expand grammars to regexps

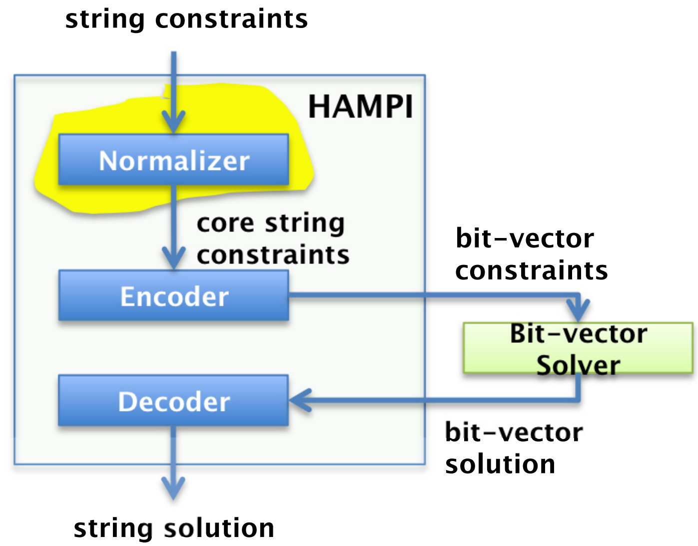
- expand nonterminals
- ➡ eliminate inconsistencies
- enumerate choices exhaustively

```
cfg E := "(" E ")" | E E | "()";
```

$$\overbrace{(\ E\ )}^{6} + \overbrace{E\ E}^{6} + \overbrace{(\,)}^{6}$$

Hampi Normalizer Converts String Constraints To Core Form

Core string constraint have only regular expressions



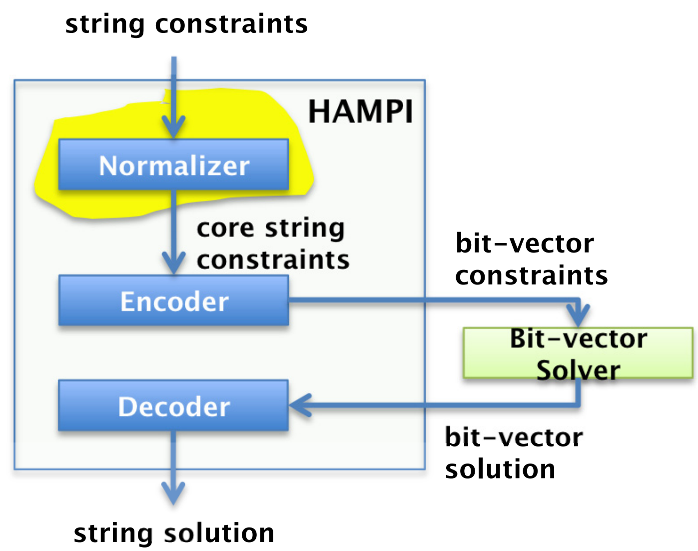
Expand grammars to regexps

- expand nonterminals
- ➡ eliminate inconsistencies
- enumerate choices exhaustively

cfg E := "(" E ")" | E E | "()";

$$\overbrace{(\ E\ )}^{6} + \overbrace{E\ E}^{6} + \overbrace{(\,)}^{6}$$

Hampi Normalizer Converts String Constraints To Core Form



Core string constraint have only regular expressions

Expand grammars to regexps

- **expand nonterminals**
- **eliminate inconsistencies**

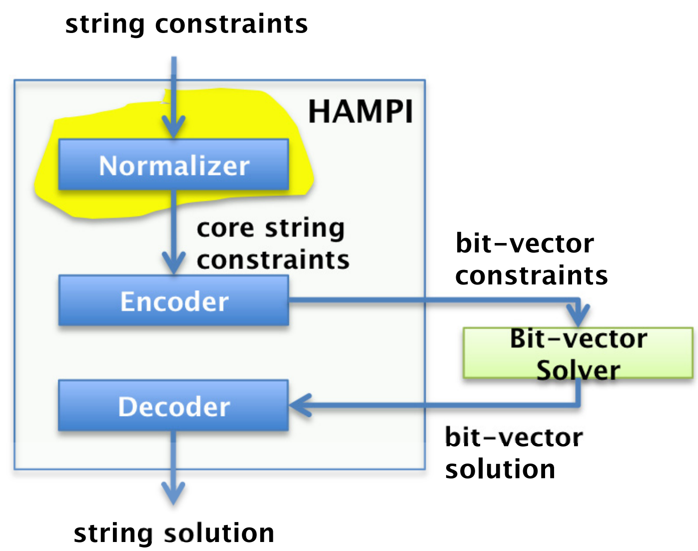
➡ **enumerate choices exhaustively**

```
cfg E := "(" E ")” | E E | "()”;
```

$$(\overbrace{E}^{4}) + \overbrace{E \ E}^{6}$$

Hampi Normalizer Converts String Constraints To Core Form

Core string constraint have only regular expressions



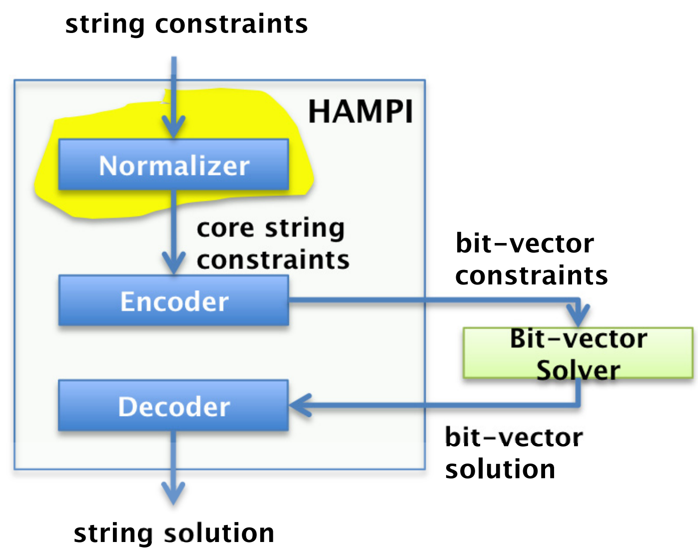
Expand grammars to regexps

- expand nonterminals
- ➡ eliminate inconsistencies
- enumerate choices exhaustively

cfg E := "(" E ")” | E E | "()”;

$$(\overbrace{E}^{4}) + \overbrace{E}^{0}\,\overbrace{E}^{6} + \overbrace{E}^{1}\,\overbrace{E}^{5} + \overbrace{E}^{2}\,\overbrace{E}^{4} + \overbrace{E}^{3}\,\overbrace{E}^{3} + \overbrace{E}^{4}\,\overbrace{E}^{2} + \overbrace{E}^{5}\,\overbrace{E}^{1} + \overbrace{E}^{6}\,\overbrace{E}^{0}$$

Hampi Normalizer Converts String Constraints To Core Form



Core string constraint have only regular expressions

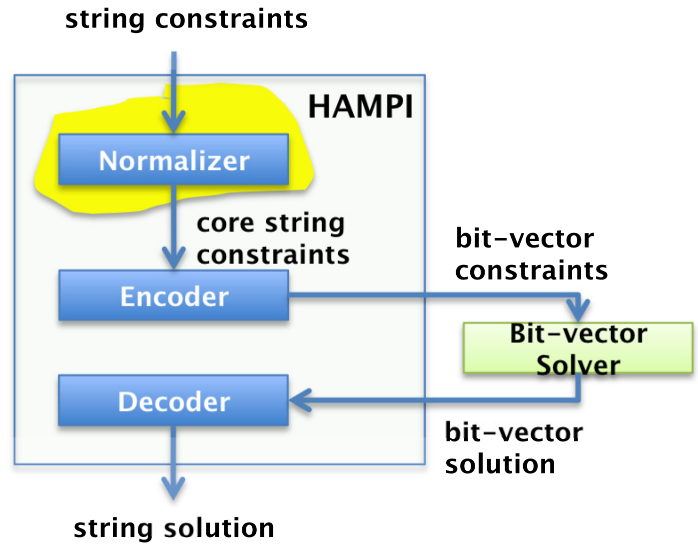
Expand grammars to regexps

- **expand nonterminals**
- ➡ **eliminate inconsistencies**
- **enumerate choices exhaustively**

```
cfg E := "(" E ")" | E E | "()";
```

$$(\overset{4}{E}) + \overset{0}{E}\ \overset{6}{E} + \overset{1}{E}\ \overset{5}{E} + \overset{2}{E}\ \overset{4}{E} + \overset{3}{E}\ \overset{3}{E} + \overset{4}{E}\ \overset{2}{E} + \overset{5}{E}\ \overset{1}{E} + \overset{6}{E}\ \overset{0}{E}$$

Hampi Normalizer Converts String Constraints To Core Form



Core string constraint have only regular expressions

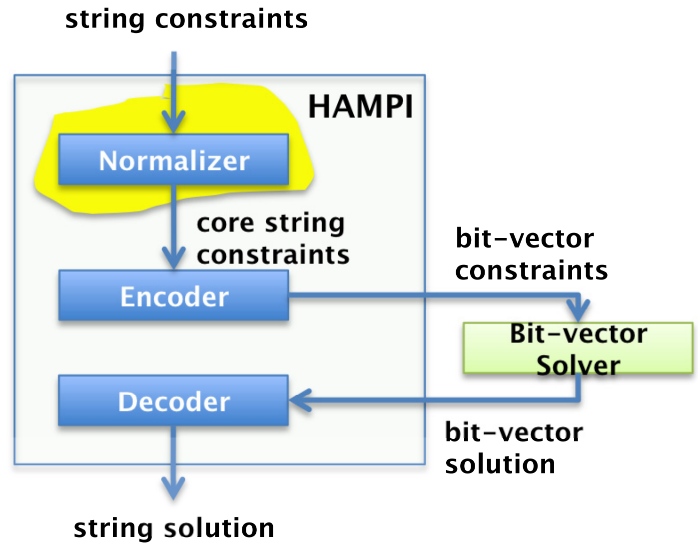
Expand grammars to regexps

- **expand nonterminals**
- **eliminate inconsistencies**
- **enumerate choices exhaustively**

```
cfg E := "(" E ")" | E E | "()";
```

$$\Rightarrow \; (\overbrace{E}^{4}) + \overbrace{E}^{2}\overbrace{E}^{4} + \overbrace{E}^{4}\overbrace{E}^{2}$$

Hampi Normalizer Converts String Constraints To Core Form



Core string constraint have only regular expressions

Expand grammars to regexps
- expand nonterminals
- eliminate inconsistencies
- enumerate choices exhaustively

cfg *E* := "(" *E* ")" | *E* *E* | "()";

([()() + (())]) +

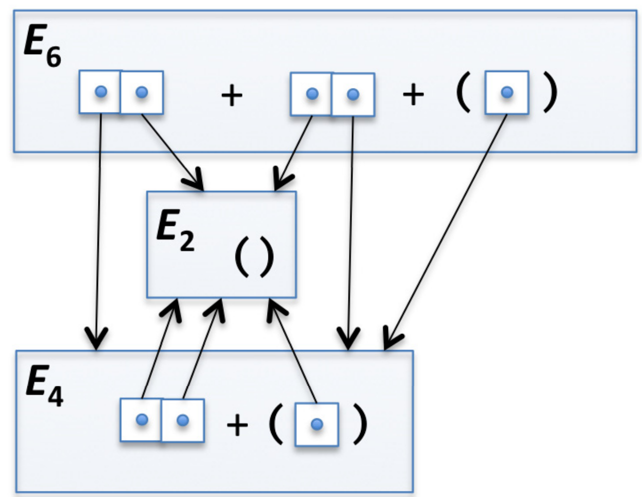
bound(*E*, 6) → () [()() + (())] +

[()() + (())] ()

Hampi Normalizer Uses Compact Representations Of Expressions

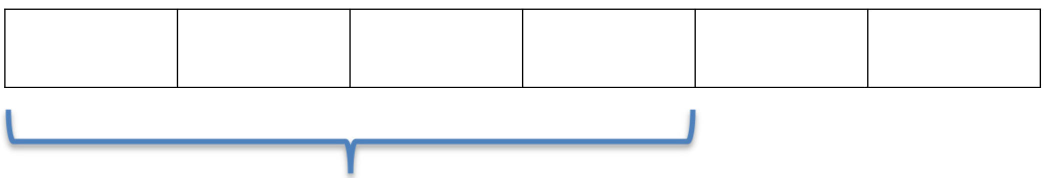
$$\left.\begin{array}{l} \overbrace{(\,)}^{E_2}\,\overbrace{[\,(\,)(\,) + ((\,))\,]}^{E_4} \;+ \\ \;[\,(\,)(\,) + ((\,))\,](\,) \;+ \\ (\,[\,(\,)(\,) + ((\,))\,]\,) \end{array}\right\} E_6$$

shared graph nodes for common subexpressions



Bit Vectors Are Ordered, Fixed-Size, Sets Of Bits

Bit vector B (length 6 bits)



$$(B[0:4] = B[2:4]) \land (B[1:3] = 101)$$

offset:length

Bit Vectors Are Ordered, Fixed-Size, Sets Of Bits

Bit vector B (length 6 bits)

0	1	0	1	0	1

$$(B[0:4] = B[2:4]) \wedge (B[1:3] = 101)$$

offset:length

Bit-vector solver finds the solution B = 010101

Hampi Encodes Core Constraints As Bit-Vector Constraints

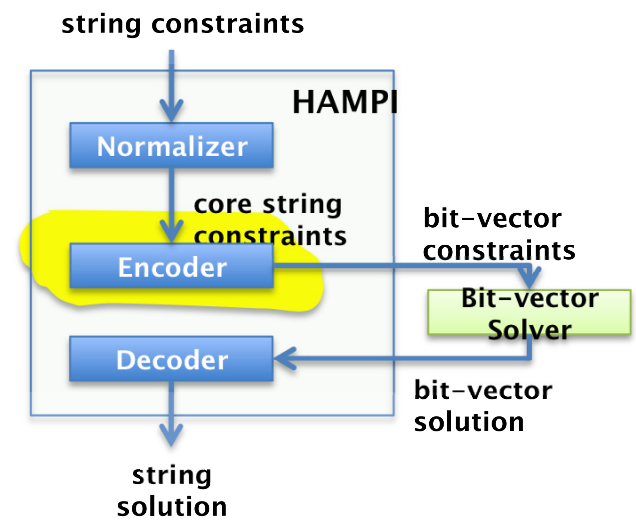
Map alphabet Σ to bit-vector constants:

(→ 0

) → 1

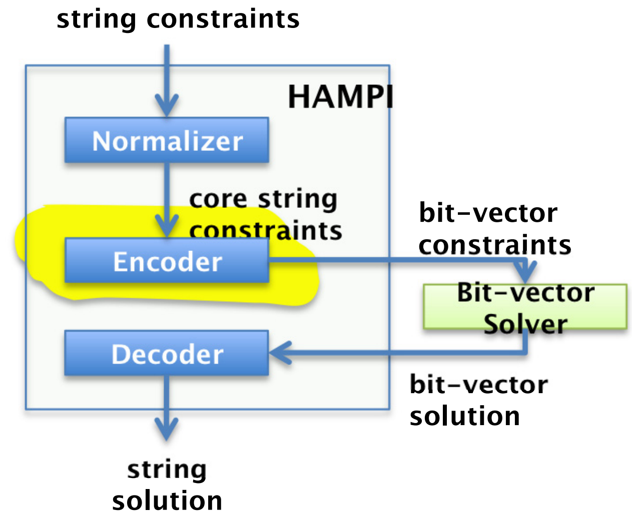
Compute size of bit-vector B:

(1+4+1) * 1 bit = 6 bits



(v) ∈ ()[()() + (())] + [()() + (())]() + ([()() + (())])

Hampi Encodes Regular Expressions Recursively



Encode regular expressions recursively

- union + → disjunction $\vee$
- concatenation → conjunction $\wedge$
- Kleene star * → conjunction $\wedge$
- constant → bit-vector constant

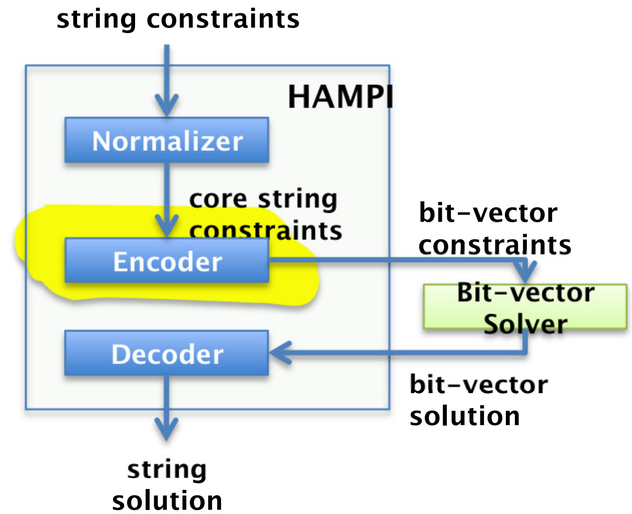
$$(v) \in ()[()() + (())] + [()() + (())]() + ([()() + (())])$$

$$\text{Formula } \Phi_1 \quad \vee \quad \text{Formula } \Phi_2 \quad \vee \quad \text{Formula } \Phi_3$$

Hampi Encoder Exploits Shift-Symmetry In Constraints

Shift-symmetric constraints = identical modulo offset in bit vector

Hampi reuses encoding templates for symmetric constraints



(v) ∈ () [()() + (())] + [()() + (())] () + ([()() + (())])

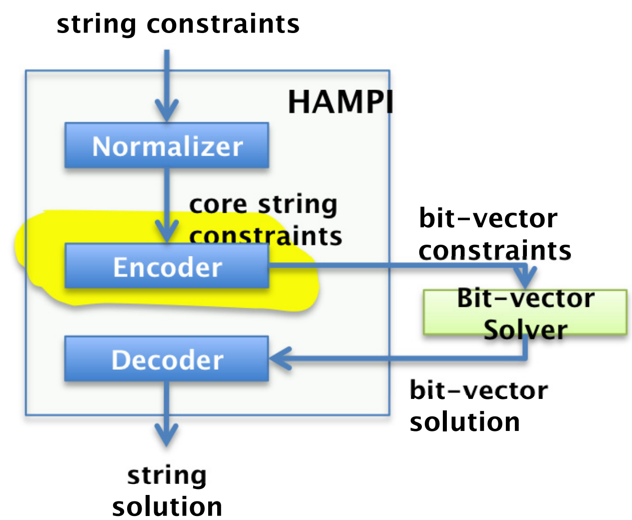
B[0:2] = 01

B[5:2] = 01

Shift-symmetric constraints

Hampi Encoder Exploits Shift-Symmetry In Constraints

Shift-symmetric constraints =
identical modulo offset in bit vector



Hampi reuses encoding templates for symmetric constraints

(v) ∈ () [()() + (())] + [()() + (())] () + ([()() + (())])

ϕ at offset 2 | ϕ at offset 0 | ϕ at offset 1

Shift-symmetric constraints

Hampi Uses Bit-Vector Solver And Decodes Solution

bit-vector constraints

Bit-vector Solver STP

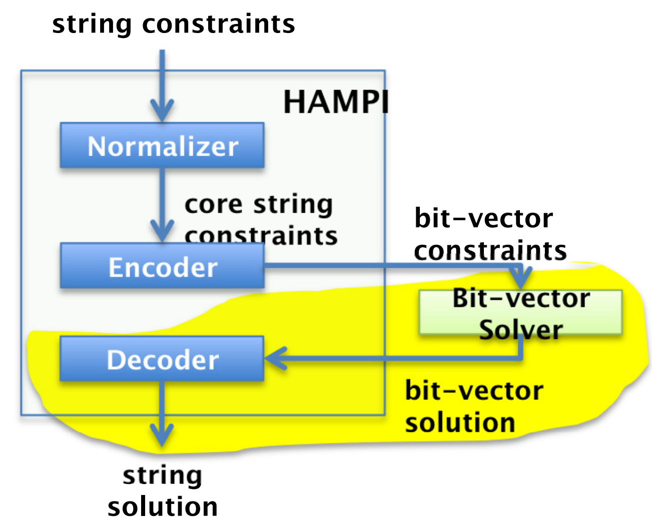
`B = 010101`

Decoder

Maps bits back to alphabet Σ

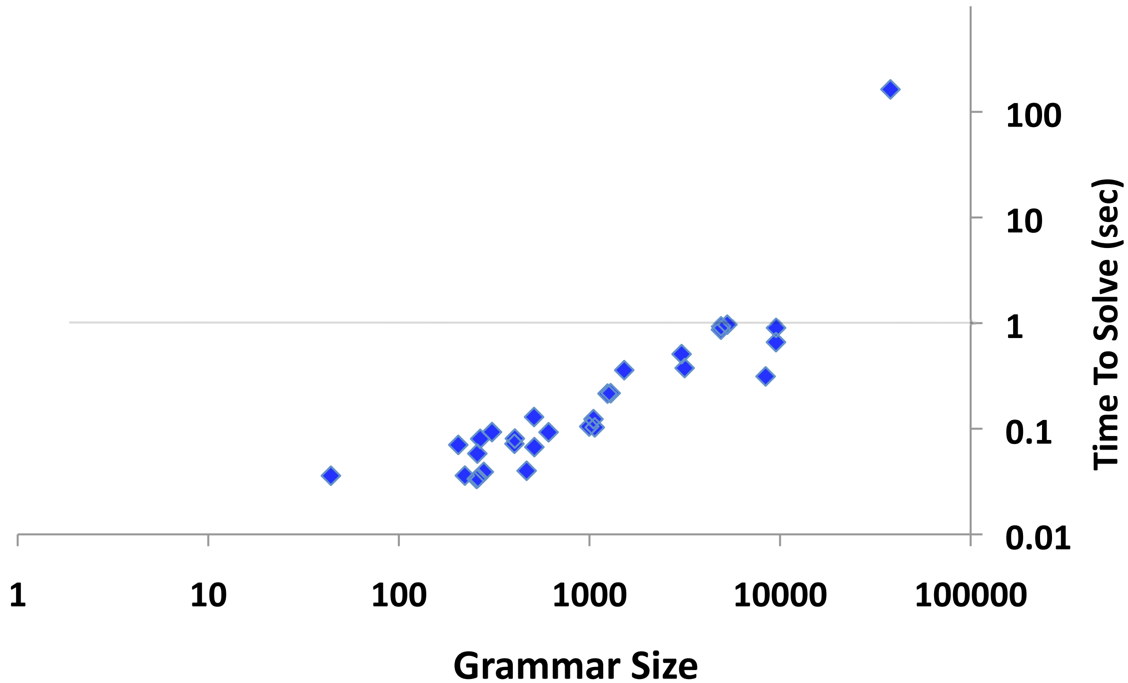
`B = ()()()`

`v =  )()(`



Result 1: Hampi Is Effective In Static SQL Injection Analysis

1367 string constraints from [Wassermann PLDI'07]



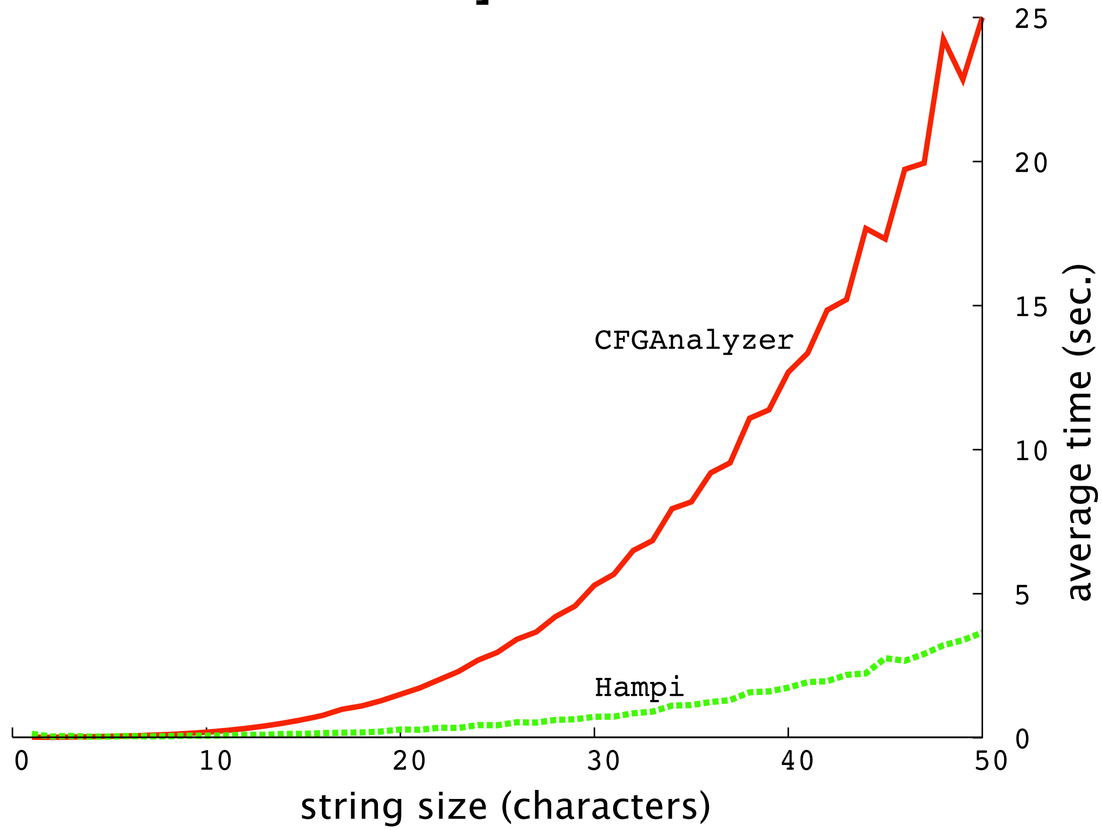
Hampi scales to large grammars

Hampi solved 99.7% of constraints in < 1 sec per constraint

All solvable constraints had short solutions $N \leq 4$

Result 2: Hampi Is Faster Than The CFGAnalyzer Solver

CFGAnalyzer encodes bounded grammar problems in SAT [Axelsson et al ICALP'08]



For size 50, Hampi is 6.8x faster on average (up to 3000x faster)

Effective Software Testing With A String-Constraint Solver

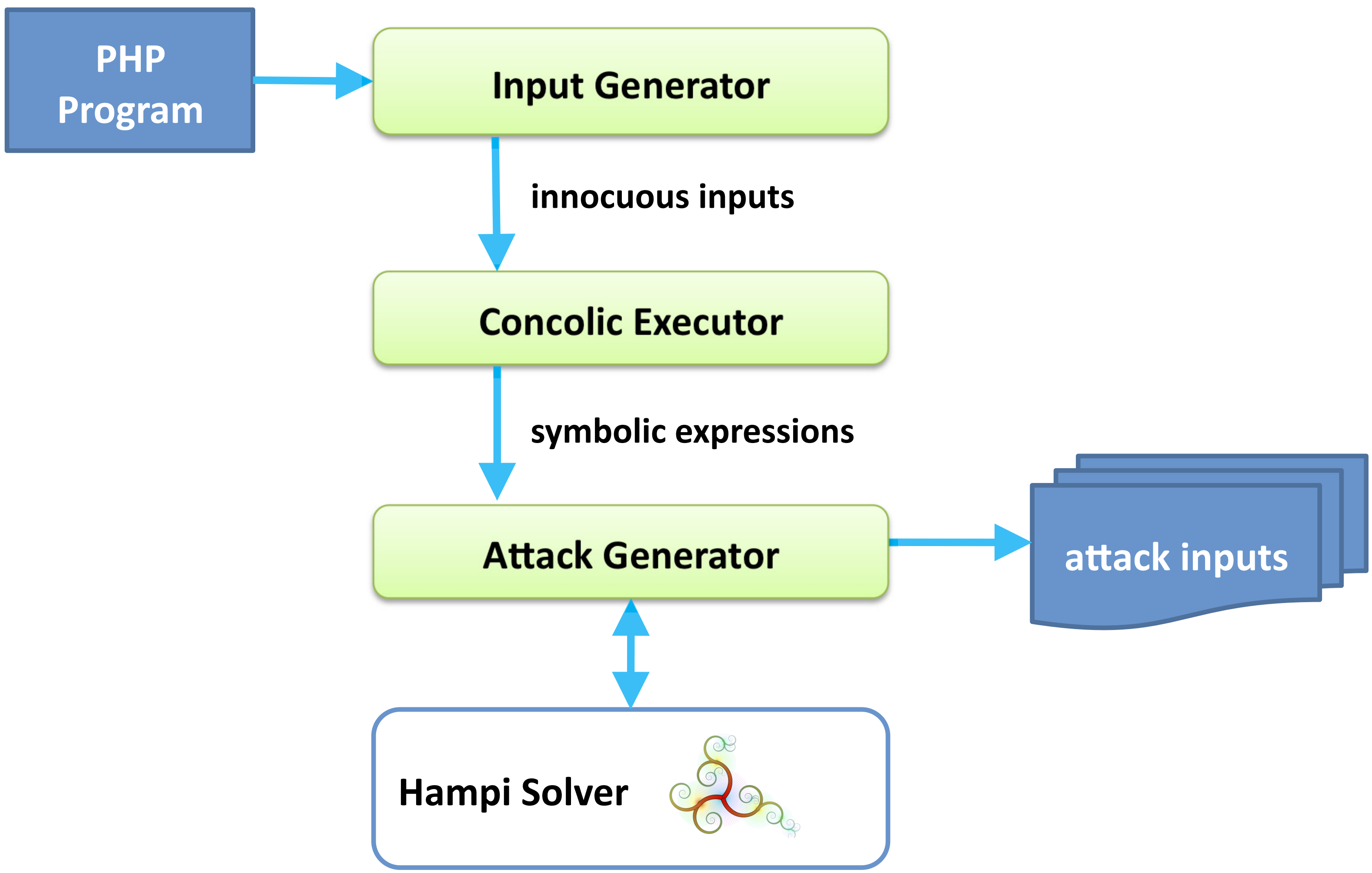
Concolic Security Testing
[ICSE'09]

Grammar-based Concolic Testing
[PLDI'08]

Concolic Testing

Hampi: String-Constraint Solver
[ISSTA'09]

Ardilla Mutates Generated Inputs To Construct Attacks



SQL Injection Attacks Modify Structure Of Database Queries

Innocuous input:
v → 1

```
SELECT msg FROM messages WHERE topicid='1'
```

SQL Injection Attacks Modify Structure Of Database Queries

Innocuous input:
v → 1

```
SELECT msg FROM messages WHERE topicid=‘1’
```

user input

Symbolic expression for SQL query

```
concat(SELECT msg FROM messages WHERE topicid=‘ v ’)
```

SQL Injection Attacks Modify Structure Of Database Queries

Innocuous input:
v → 1

```
SELECT msg FROM messages WHERE topicid=‘1’
```

user input

Symbolic expression for SQL query

```
concat(SELECT msg FROM messages WHERE topicid=‘ v ’)
```

Attack input:
v → 1’ OR ‘0’=‘0

attack tautology

```
SELECT msg FROM messages WHERE topicid=‘1’ OR ‘0’=‘0’
```

Attacker gets access to all messages

Example: Hampi Constraints That Create SQL Injection Attacks

user input string

```
var v : 12;
```

SQL grammar

```
cfg SqlSmall := "SELECT ” [a-z]+ " FROM ” [a-z]+ " WHERE " Cond;
cfg Cond := Val "=" Val | Cond " OR " Cond;
cfg Val := [a-z]+ | "'” [a-z0-9]* "'" | [0-9]+;
```

bounded SQL grammar

```
reg SqlSmallBounded := bound(SqlSmall, 53);
```

SQL query

```
val q := concat("SELECT msg FROM messages WHERE topicid='", v, "'");
```

SQLI attack conditions

```
assert q in SqlSmallBounded;
assert q contains "OR ‘0'=‘0'";
```

“q is a valid SQL query”

“q contains an attack tautology”

Example: Hampi Constraints That Create SQL Injection Attacks

user input string

```
var v : 12;
```

SQL grammar

```
cfg SqlSmall := "SELECT ” [a-z]+ " FROM ” [a-z]+ " WHERE " Cond;
cfg Cond := Val "=" Val | Cond " OR " Cond;
cfg Val := [a-z]+ | "'” [a-z0-9]* "'" | [0-9]+;
```

bounded SQL grammar

```
reg SqlSmallBounded := bound(SqlSmall, 53);
```

SQL query

```
val q := concat("SELECT msg FROM messages WHERE topicid='", v, "'");
```

SQLI attack conditions

```
assert q in SqlSmallBounded;
assert q contains "OR ‘0'=‘0'";
```

“q is a valid SQL query”

“q contains an attack tautology”

Hampi finds an attack input: v → 1’ OR ‘0’=‘0

Result: Ardilla Finds New Attacks

60 attacks on 5 PHP applications

23 SQL injection

4 cases of data corruption
19 cases of information leak

29 XSS first order

8 XSS second order

0 false positives

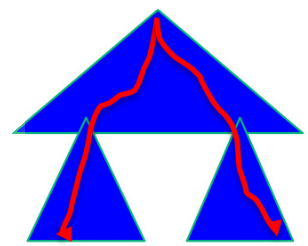
216 Hampi constraints solved

- 46% of constraints in < 1 second per constraint
- 100% of constraints in < 10 seconds per constraint

Effective Software Testing With A String-Constraint Solver

Concolic Security Testing
[ICSE'09]

Grammar-based Concolic Testing
[PLDI'08]



Concolic Testing

Hampi: String-Constraint Solver
[ISSTA'09]

Sometimes Concolic Testing Is Not Much Better Than Random Fuzzing

Randomly mutates bytes seed inputs

Random Fuzz Testing

50 well-formed seed inputs

Concolic Testing

50 well-formed seed inputs

Program under test: JavaScript interpreter

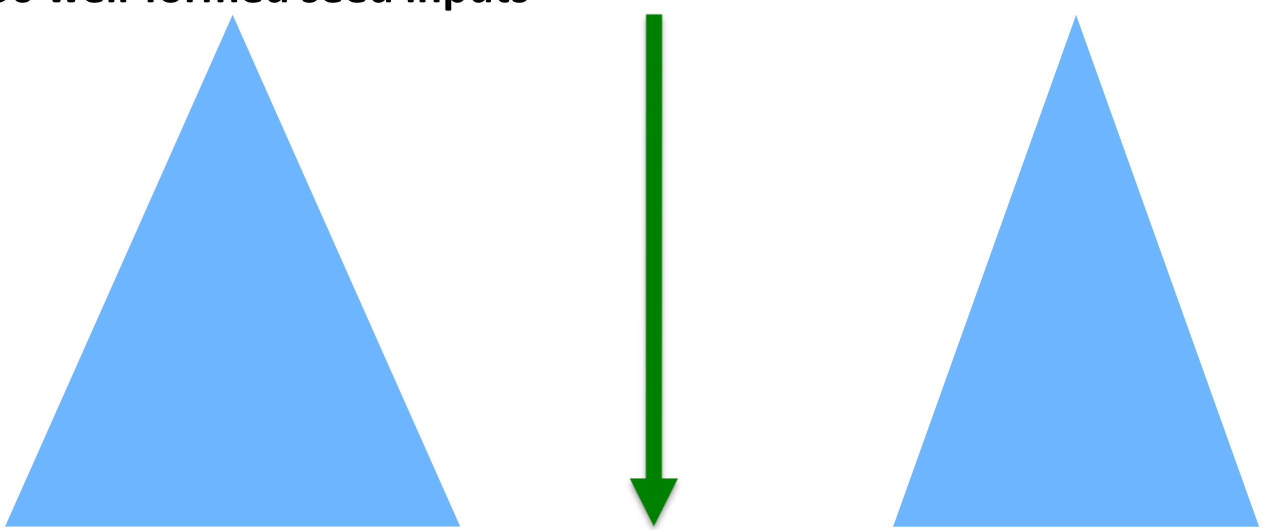
Sometimes Concolic Testing Is Not Much Better Than Random Fuzzing

Random Fuzz Testing

50 well-formed seed inputs

Concolic Testing

50 well-formed seed inputs



8658 inputs

2 hours

6883 inputs

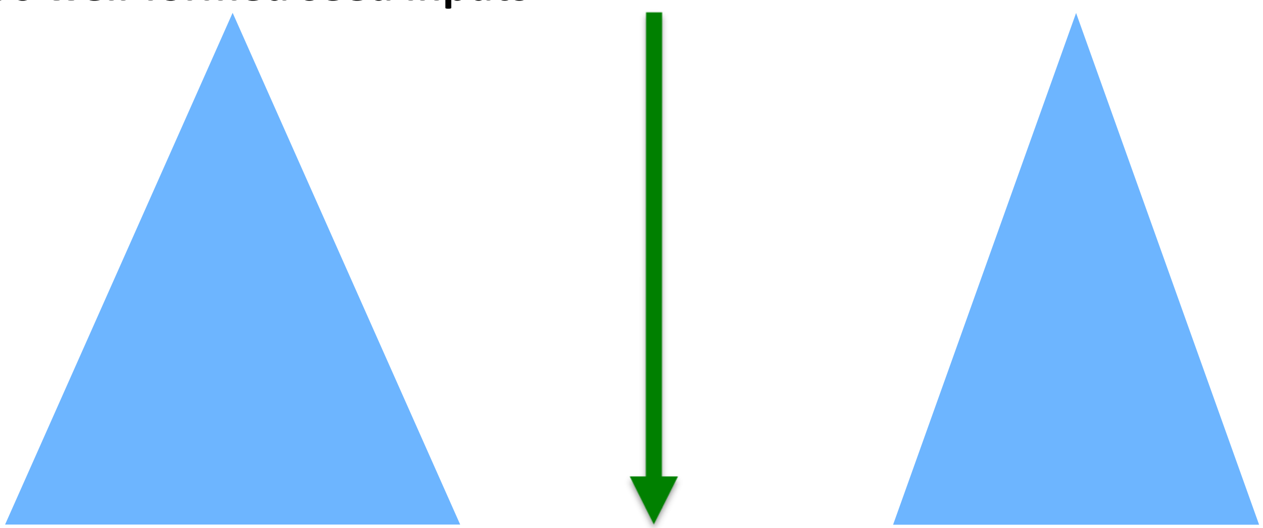
Sometimes Concolic Testing Is Not Much Better Than Random Fuzzing

Random Fuzz Testing

50 well-formed seed inputs

Concolic Testing

50 well-formed seed inputs



8658 inputs

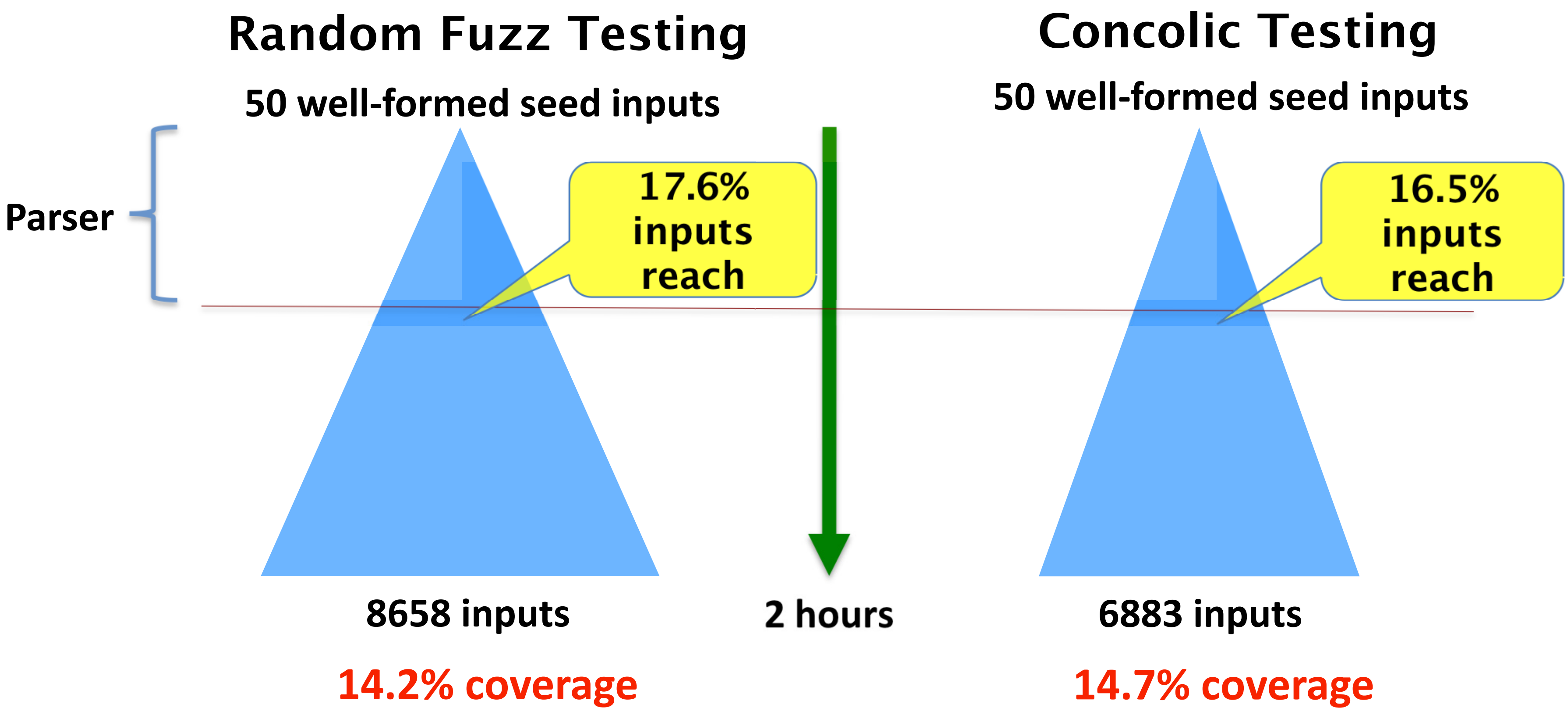
14.2% coverage

2 hours

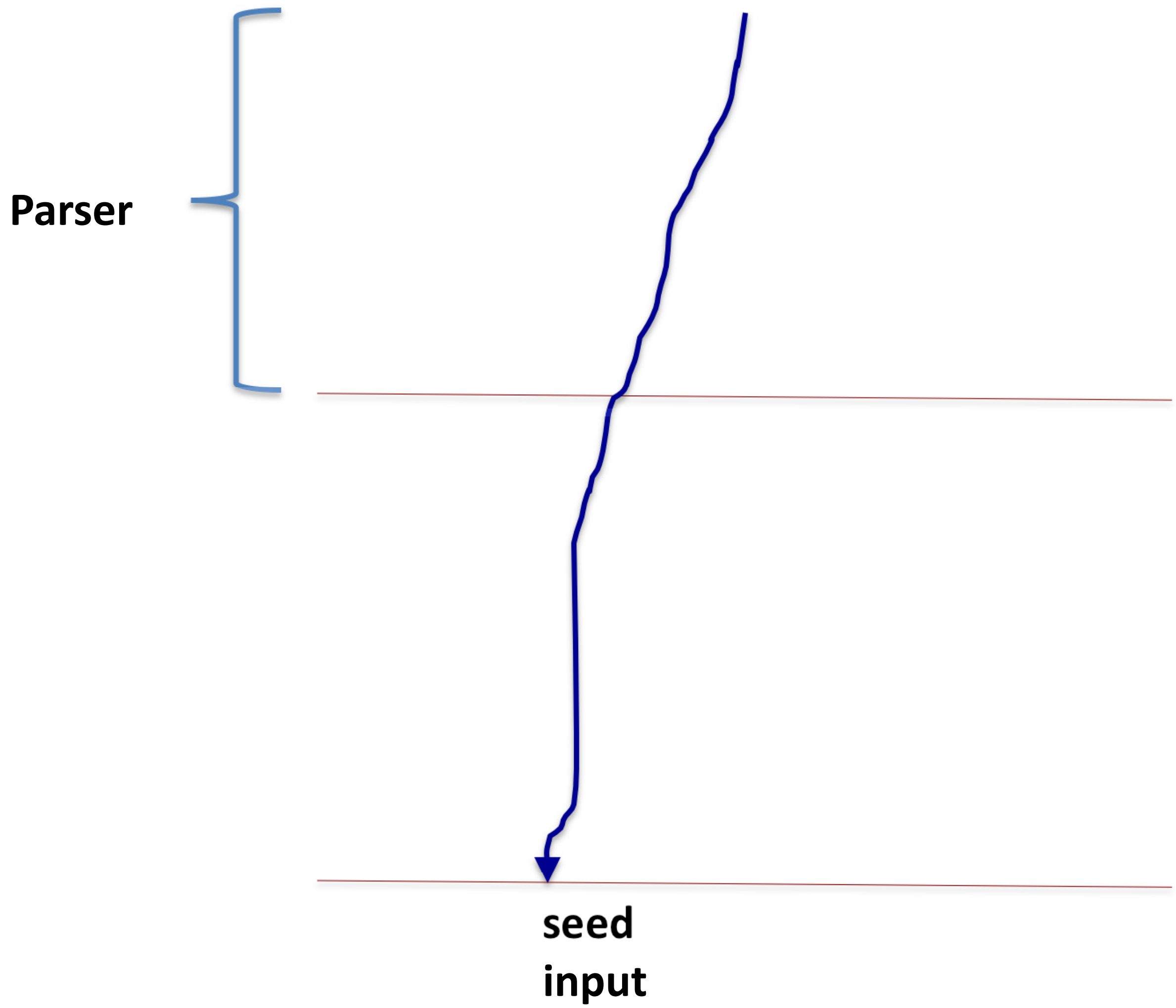
6883 inputs

14.7% coverage

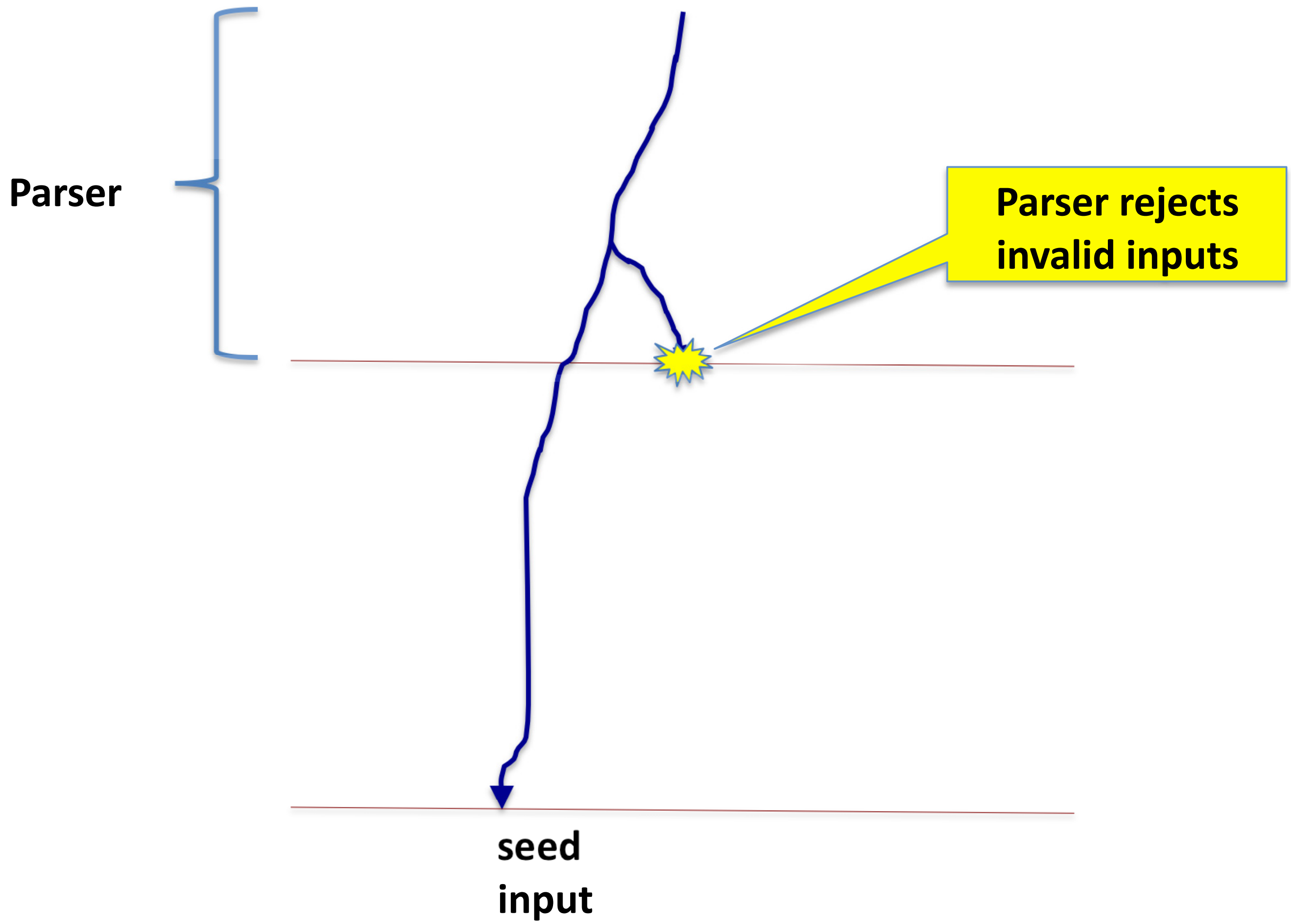
Sometimes Concolic Testing Is Not Much Better Than Random Fuzzing



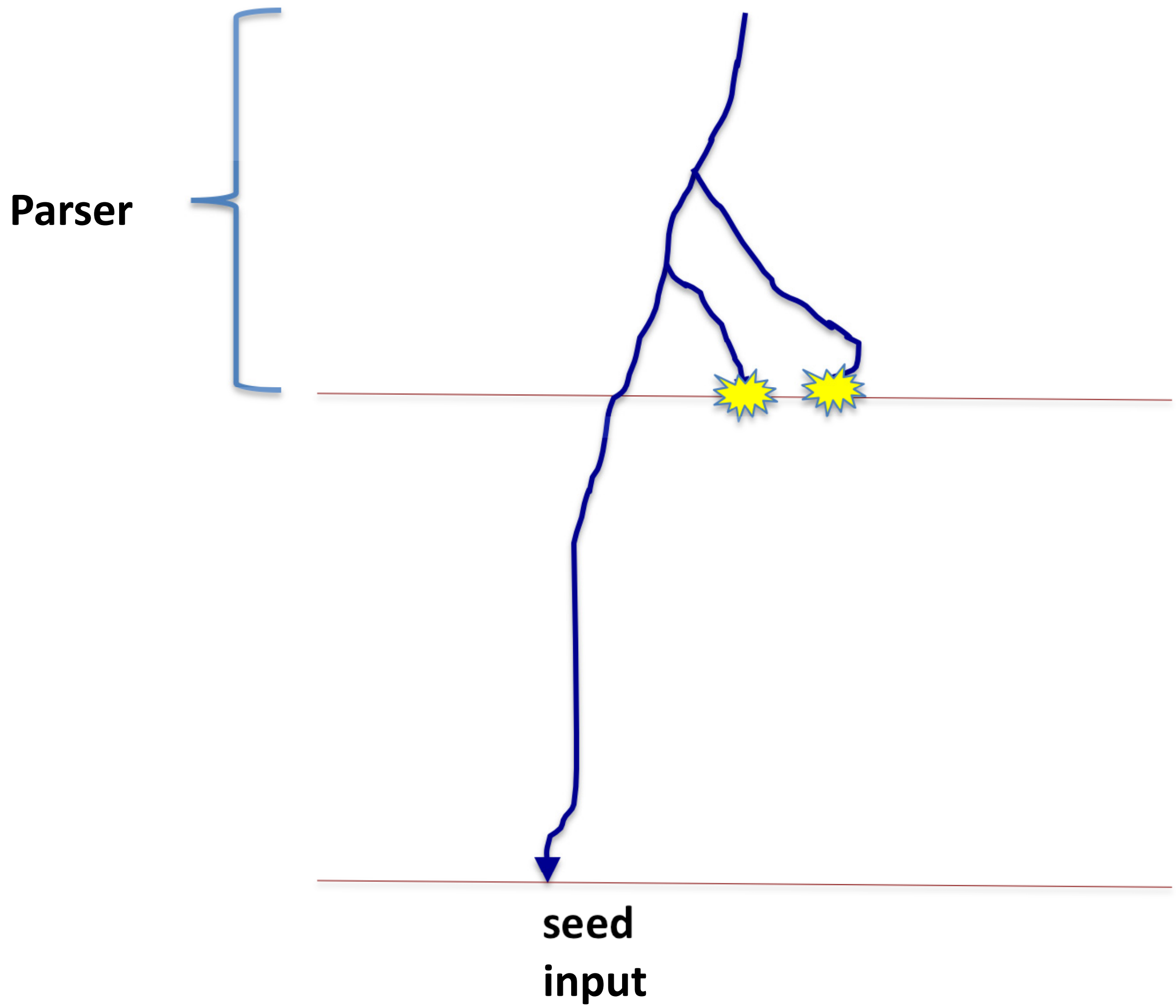
Most Generated Inputs Get Rejected Quickly



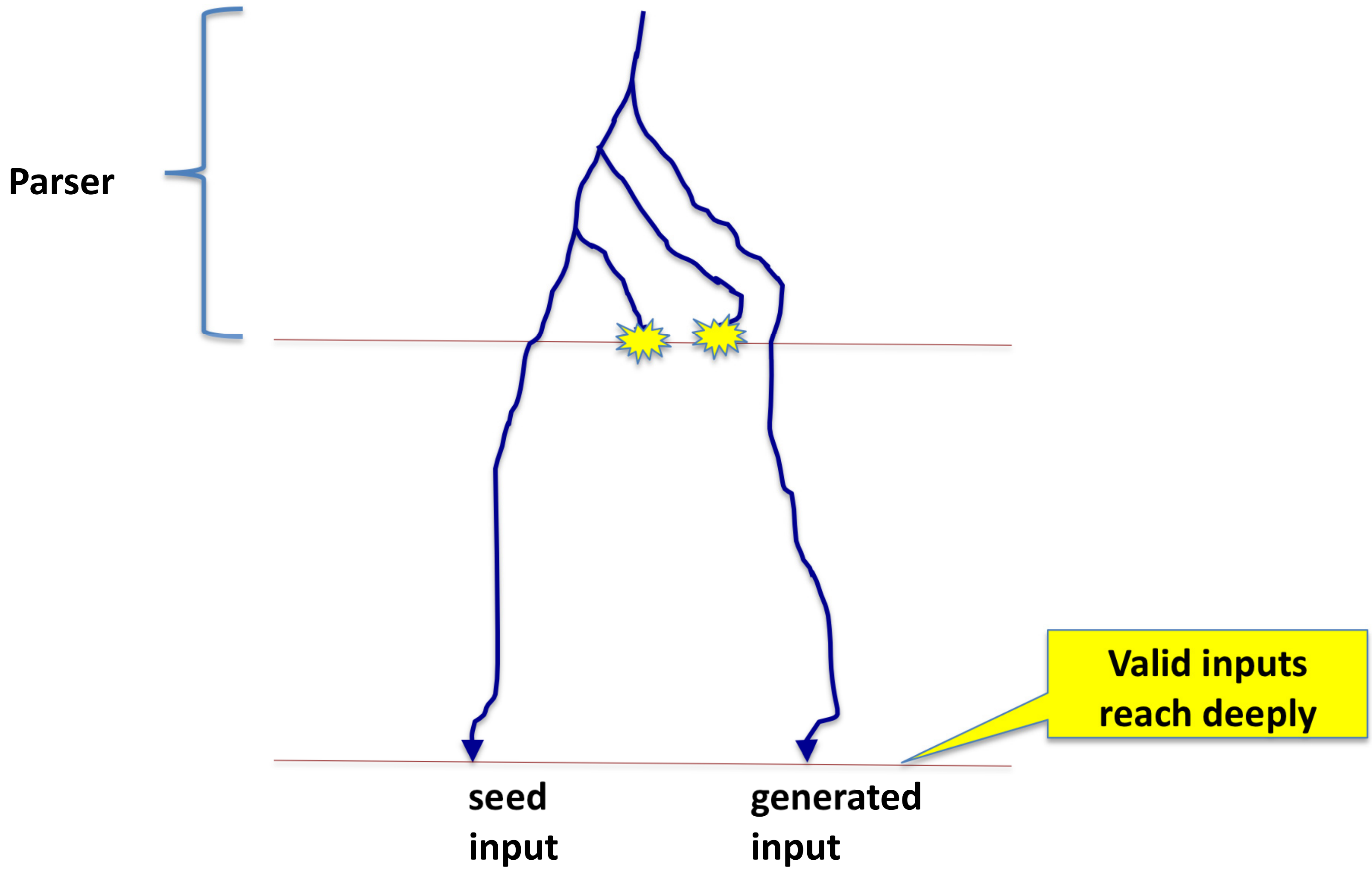
Most Generated Inputs Get Rejected Quickly



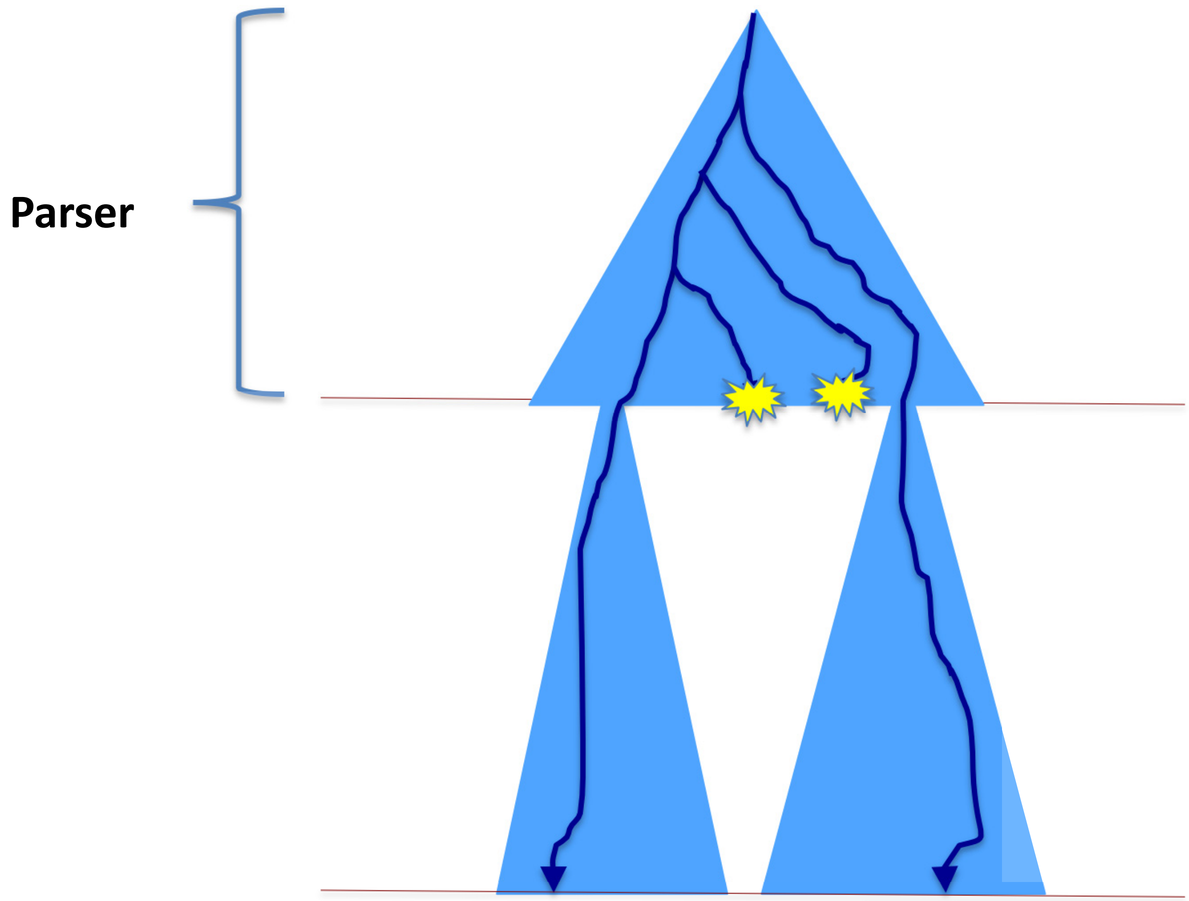
Most Generated Inputs Get Rejected Quickly



Most Generated Inputs Get Rejected Quickly

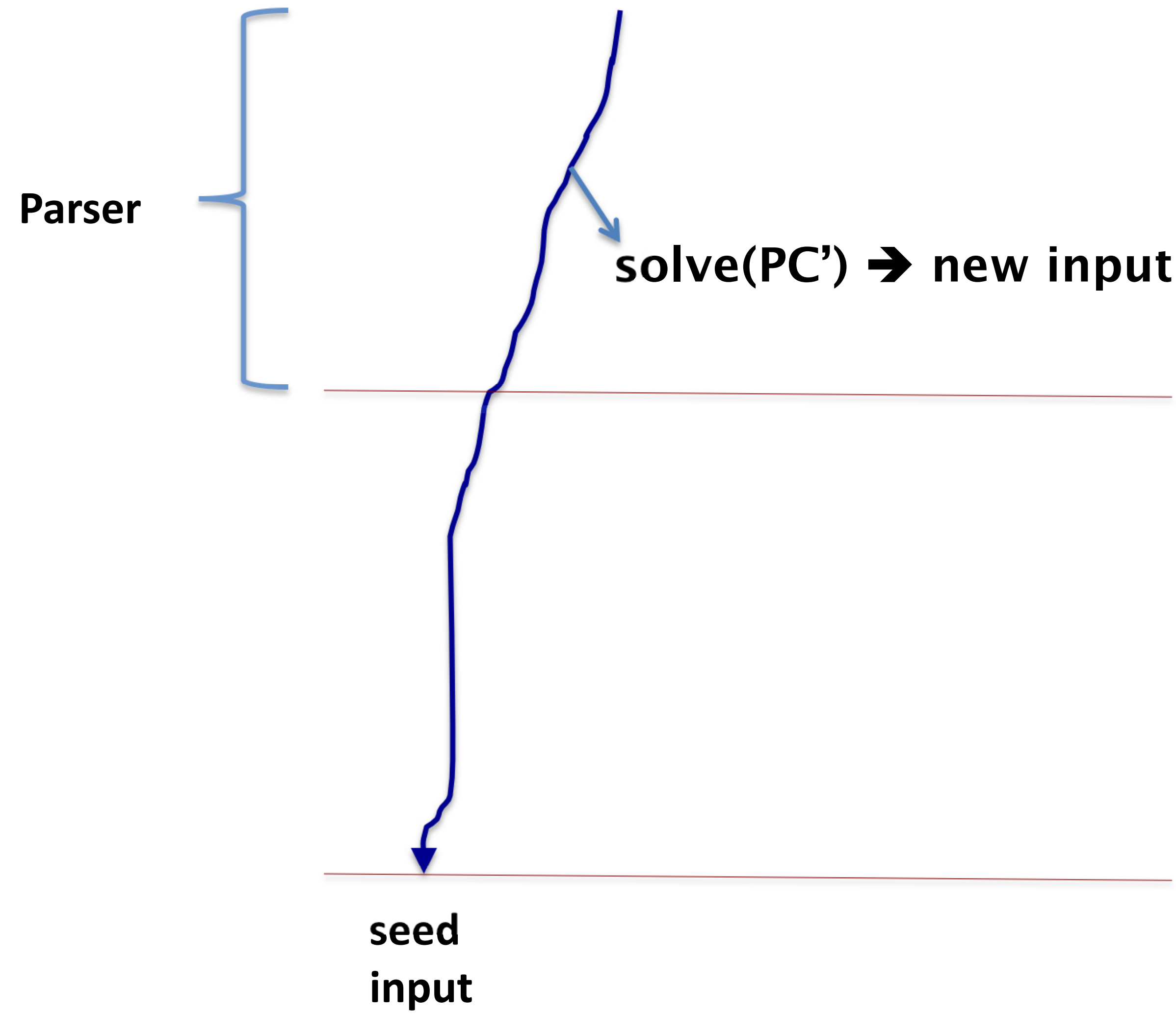


Most Generated Inputs Get Rejected Quickly

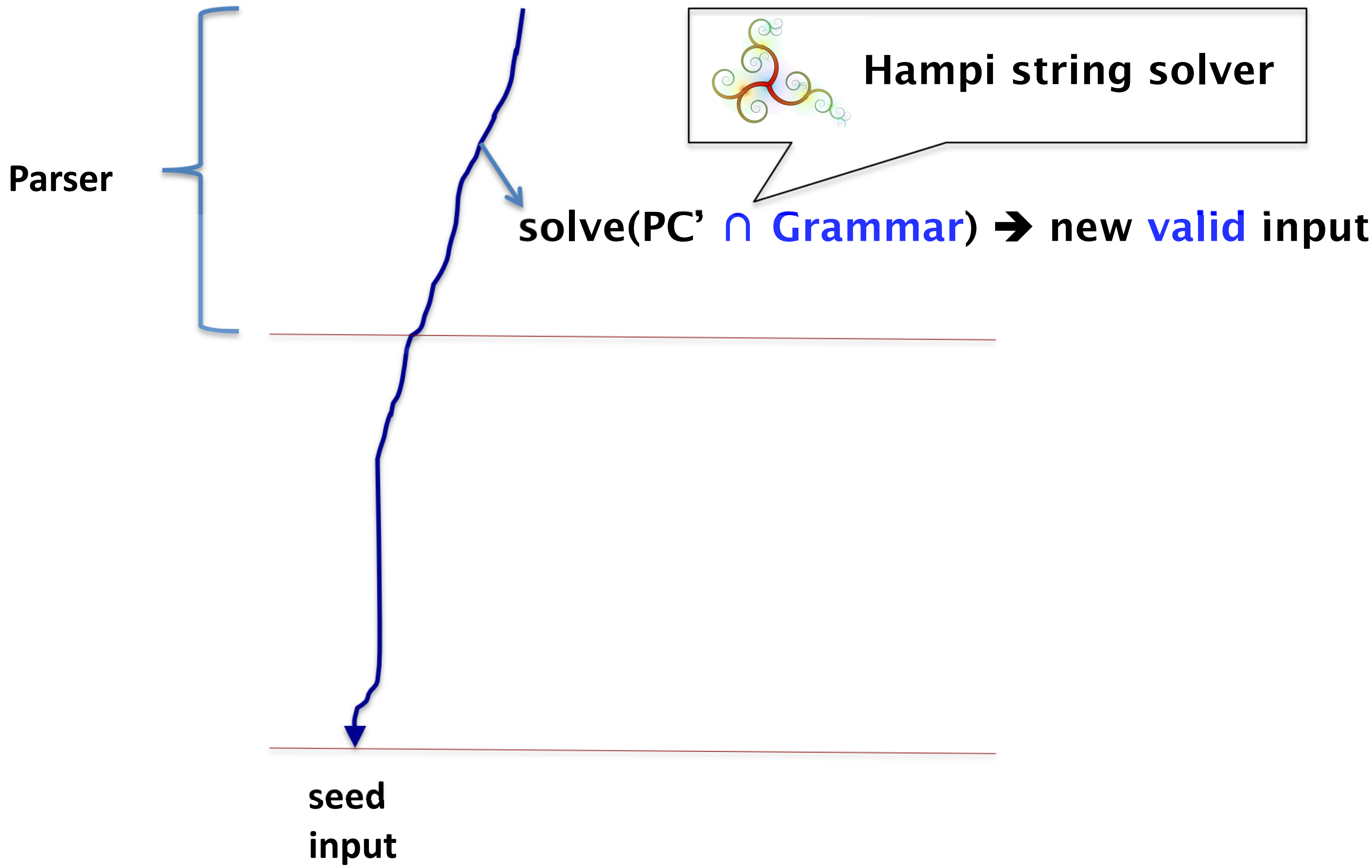


Key idea: generate only valid inputs

Input-Format Grammar Guides Creation Of Effective Inputs



Input-Format Grammar Guides Creation Of Effective Inputs



String-Constraint Solver Helps Create Valid Inputs

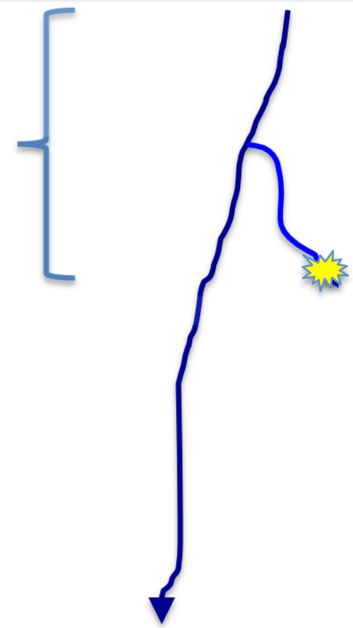
Seed input (for JavaScript interpreter):

```
function f(){ var v = 3; }
```

Constraints on tokens (created during execution)

token$_0$ = function
token$_1$ = id
token$_2$ = (
token$_3$ =)
token$_4$ = {
token$_5$ = var
…

String-Constraint Solver Helps Create Valid Inputs



Seed input (for JavaScript interpreter):

```
function f(){ var v = 3; }
```

Constraints on tokens
(created during execution)

$\texttt{token}_0 = \texttt{function}$
$\texttt{token}_1 = \texttt{id}$
$\texttt{token}_2 = \texttt{(}$
$\texttt{token}_3 = \texttt{)}$
$\texttt{token}_4 = \texttt{\{}$
$\texttt{token}_5 \neq \texttt{var}$

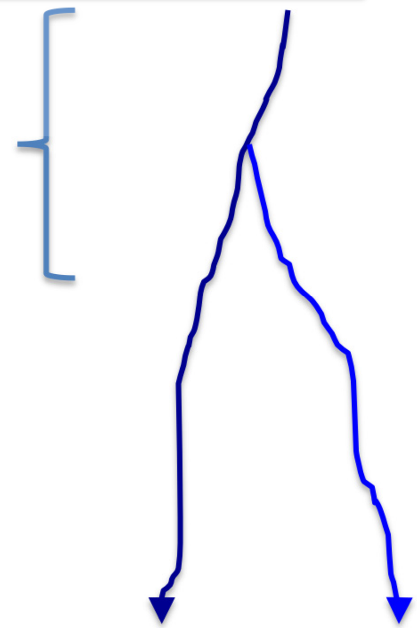
Normal solver → nonparsable input

```
function f(){ try v = 3; }
```

String-Constraint Solver Helps Create Valid Inputs

Seed input (for JavaScript interpreter):

```
function f(){ var v = 3; }
```



Constraints on tokens
(created during execution)

token$_0$ = function
token$_1$ = id
token$_2$ = (
token$_3$ =)
token$_4$ = {
token$_5$ ≠ var

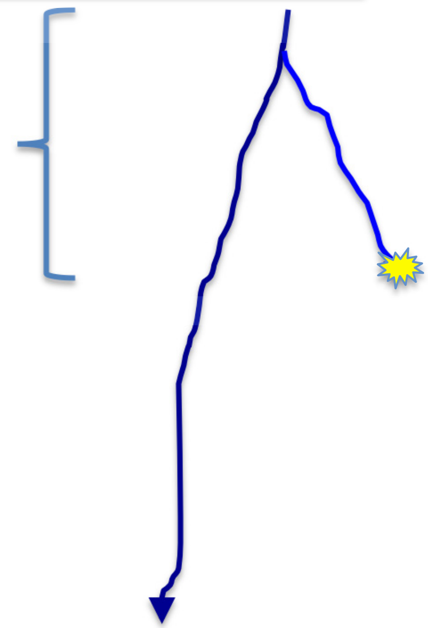
Normal solver → nonparsable input

```
function f(){ try v = 3; }
```

Hampi solver → complete parsable input

```
function f(){ try { } catch ( id ) { } finally { }; }
```

String-Constraint Solver Helps Avoid Dead-End Inputs



Seed input (for JavaScript interpreter):

```
function f(){ var v = 3; }
```

Constraints on tokens (created during execution)

$token_0$ = function

$token_1$ = id

$token_2$ = (

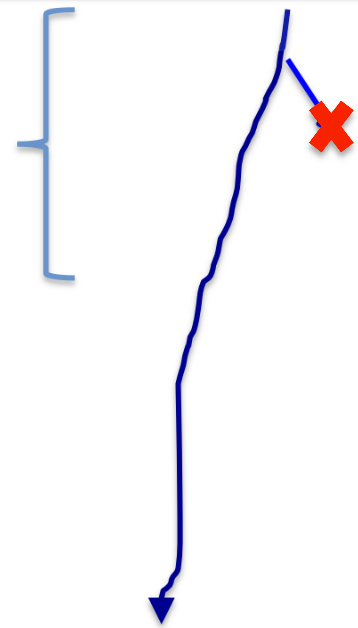
$token_3$ =)

$token_4 \neq$ {

Normal solver → nonparsable input

```
function f() var var v = 3; }
```

String-Constraint Solver Helps Avoid Dead-End Inputs



Seed input (for JavaScript interpreter):

```
function f(){ var v = 3; }
```

Constraints on tokens (created during execution)

$token_0$ = function

$token_1$ = id

$token_2$ = (

$token_3$ =)

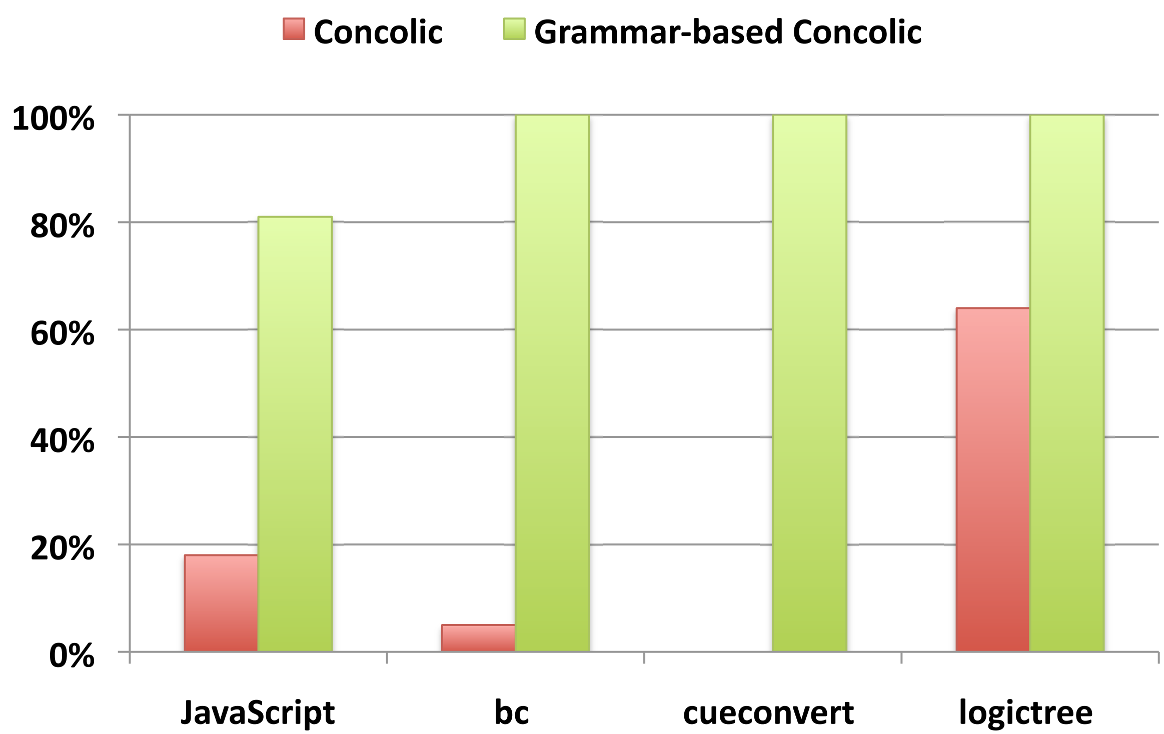
$token_4 \neq$ {

Normal solver → nonparsable input

```
function f() var var v = 3; }
```

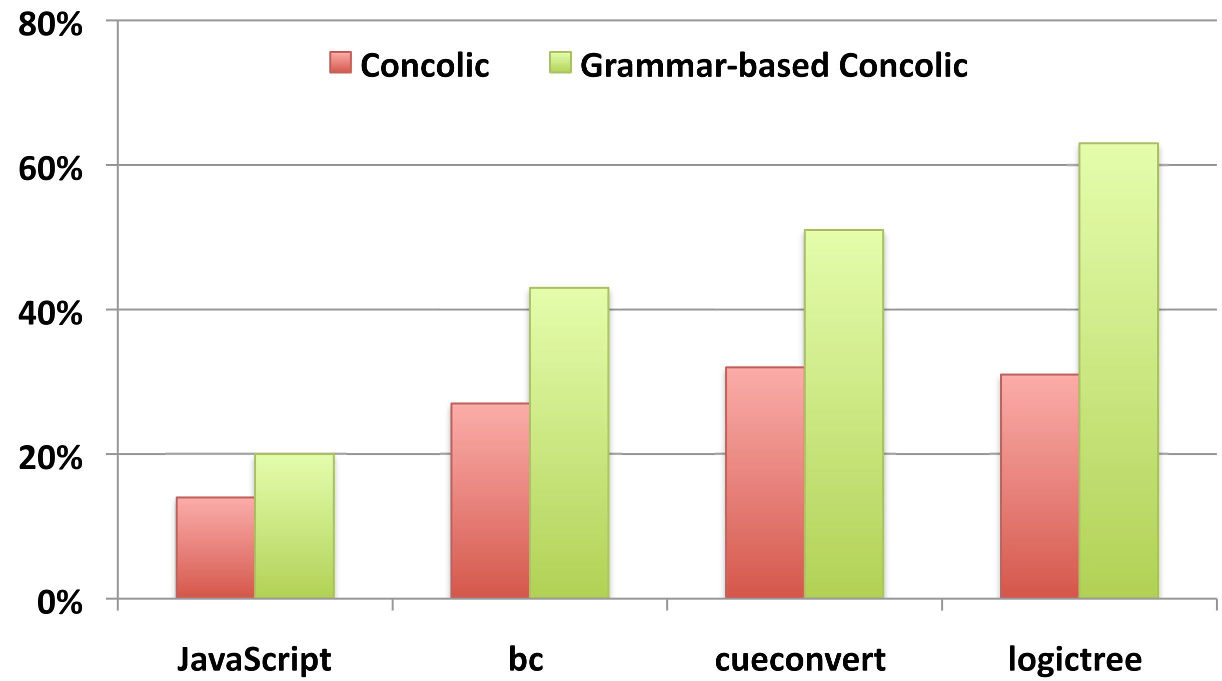
Hampi solver → no input tested, search tree pruned

Results: Grammar-Based Concolic Testing Improves Deep Reachability



Up to 20x deep reachability improvement: more generated inputs reach beyond the parser

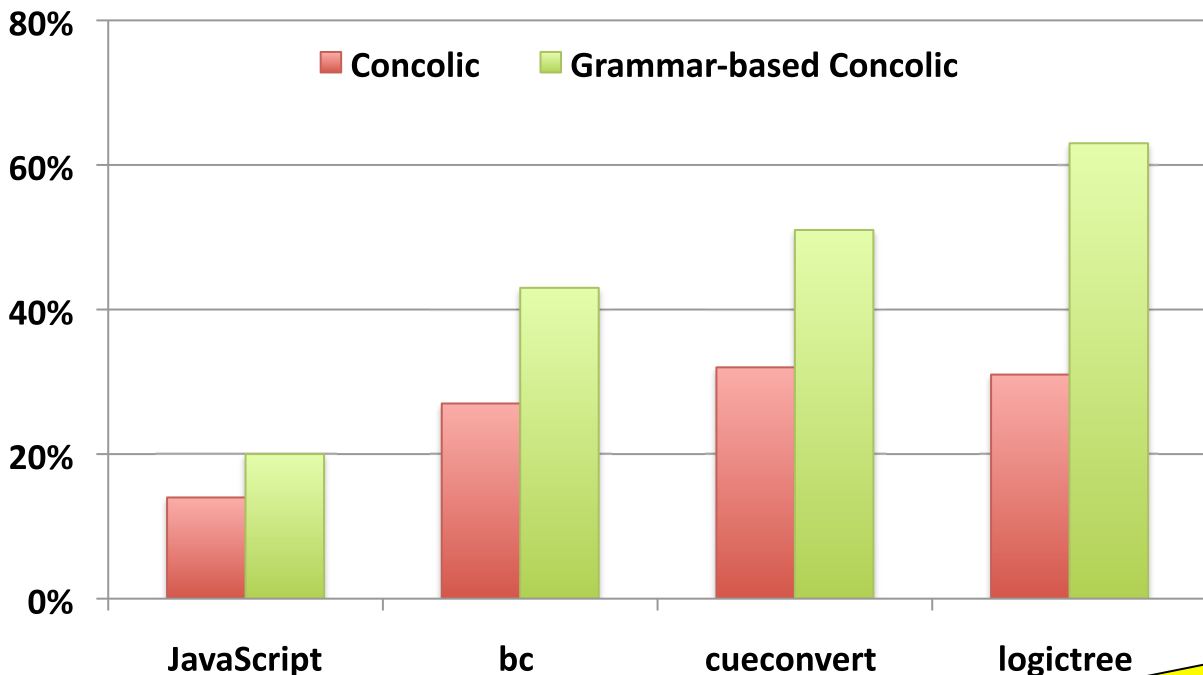
Results: Grammar-Based Concolic Testing Improves Coverage



Up to 2x coverage improvement

Results: Grammar-Based Concolic Testing Improves Coverage

and finds new bugs

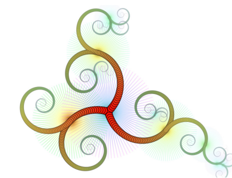


3 infinite-loop bugs

Up to 2x coverage improvement

Summary: Effective Software Testing With A String-Constraint Solver

Hampi String-Constraint Solver

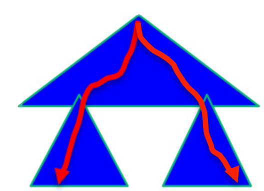


- expressive: supports context-free grammars
- efficient: solver real-world constraint quickly

Concolic Security Testing

- creates attacks on Web applications by input generation and mutation with Hampi string-constraint solver

Grammar-Based Concolic Testing



- effectively tests programs with structured inputs by using Hampi string-constraint solver and input grammars