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

Reported Severe Vulnerabilities
source: US-CERT
Software Testing Aims To Find Errors Before Users (Or Hackers) Do

Goals of software testing
- improve quality
- protect from adversaries

Reported Severe Vulnerabilities
source: US-CERT

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 **concrete** and **symbolic** 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]
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;
}
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:  \( (\text{in}[0] \neq 'b') \)
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'))
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**

```c
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:** \((in[0] \neq 'b') \land (in[1] \neq 'a') \land (in[2] = 'd') \Rightarrow xyd\)
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: \((\text{in}[0] \neq 'b') \land (\text{in}[1] = 'a') \rightarrow \text{xaz}\)
Concolic Testing Combines Dynamic Symbolic Execution, Path Enumeration

```c
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: \((\text{in}[0]=='b') \Rightarrow \text{byz}\)
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;
}
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

Generated inputs (each covers a new path)
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;
}
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 $\Sigma$ (e.g., ASCII characters)

String variables over $\Sigma^*$

```plaintext
var v
```

String constraints – language membership:

```plaintext
assert v \in L
```

String operations

```plaintext
concat("foo", v, "bar")
```
Hampi Uses Length Bounding To Support Context-Free Constraints

more expressive

context-free \mid L_1 \cap \ldots \cap L_N \quad \text{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

<table>
<thead>
<tr>
<th>Type</th>
<th>Expression</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>context-free</td>
<td>$L_1 \cap \ldots \cap L_N$</td>
<td>Undecidable</td>
</tr>
<tr>
<td>regular</td>
<td>$R_1 \cap \ldots \cap R_N$</td>
<td>PSPACE-complete</td>
</tr>
<tr>
<td>bounded regular</td>
<td>$r_1 \cap \ldots \cap r_N$</td>
<td>NP-complete</td>
</tr>
</tbody>
</table>

more expressive

more tractable
Hampi Uses Length Bounding To Support Context-Free Constraints

- **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

\( \text{bound(any language)} \rightarrow \text{bounded regular} \)
**Hampi Uses Length Bounding To Support Context-Free Constraints**

<table>
<thead>
<tr>
<th></th>
<th>$L_1 \cap \ldots \cap L_N$</th>
<th>Undecidable</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>context-free</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>regular</strong></td>
<td>$R_1 \cap \ldots \cap R_N$</td>
<td>PSPACE-complete</td>
</tr>
<tr>
<td><strong>bounded regular</strong></td>
<td>$r_1 \cap \ldots \cap r_N$</td>
<td>NP-complete</td>
</tr>
</tbody>
</table>

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

```plaintext
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;

Context-free grammar

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

\[
\text{var } v : 4;
\]

\[
\text{cfg } E := "()" \mid E \ E \mid "(" \ E \ "")";
\]

\[
\text{reg } E_{\text{bounded}} := \text{bound}(E, 6);
\]

\[
\text{val } q := \text{concat}(\ "(" , v , ", " , "")\"\ );
\]

\[
\text{assert } q \text{ in } E_{\text{bounded}};
\]

\[
\text{assert } q \text{ contains } "()()";
\]

"Find a 4-character string \( v \), such that:
\begin{itemize}
  \item \( v \) has balanced parentheses, and
  \item \( v \) contains substring \( ()() \)
\end{itemize}"
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 "()()";
```

“Find a 4–character string v, such that:
• (v) has balanced parentheses, and
• (v) contains substring ( )()”
Hampi Can Solve Context-Free and Regular Constraints

```plaintext
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 finds satisfying assignment v = )())(}
# Hampi Supports Rich String Constraints

<table>
<thead>
<tr>
<th></th>
<th>Hampi</th>
<th>CFGAnalyzer</th>
<th>Wassermann</th>
<th>Bjorner</th>
<th>Hooijmeier</th>
<th>Emmi</th>
<th>MONA</th>
<th>Caballero</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>context-free grammars</strong></td>
<td>🟢</td>
<td>🟢</td>
<td>🟢</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>regular expressions</strong></td>
<td>🟢</td>
<td>🟢</td>
<td>🟢</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>string concatenation</strong></td>
<td>🟢</td>
<td>🟢</td>
<td>🟢</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>stand-alone tool</strong></td>
<td>🟢</td>
<td>🟢</td>
<td>🟢</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>unbounded length</strong></td>
<td>🟢</td>
<td>🟢</td>
<td>🟢</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Hampi Encodes String Constraints In Bit-Vector Logic

string constraints

HAMPI

Normalizer

core string constraints

Encoder

bit-vector constraints

Decoder

Bit-vector Solver

bit-vector solution

string solution
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

\[
\text{cfg } E := "( )" \mid E E \mid "()";
\]

\[
\text{bound}(E, 6)
\]
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 \( E \) := "(" \( E \) ")" | \( E \) \( E \) | "()";
```
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

\[
\text{cfg } E := "\( E \)" \mid E \ E \mid "(\)"
\]

\[
\begin{align*}
\text{( } E \text{ )} & + \quad E E & + \quad ( \\
6 & \quad 6 & \quad 6
\end{align*}
\]
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 | "()";
```
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

\[
\text{cfg } E := "( " E " )" | E E | "( )";
\]

4

( E ) + 6

E E
Core string constraint have only regular expressions

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

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

( 4 0 6 1 5 2 4 3 3 4 2 5 1 6 0 )

Hampi Normalizer Converts String Constraints To Core Form
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 | "()";
```

(bit-vector solution)
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

\[
\text{cfg} \quad E := "(" \ E \ "")" \ | \ E \ E \ | \ "()";
\]

\[
\left( \begin{array}{c} 4 \\ E \end{array} \right) + \left( \begin{array}{c} 2 \\ E \end{array} \right) \ E + \left( \begin{array}{c} 4 \\ E \end{array} \right) \ E
\]
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 | "()";

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

bound(E, 6) \rightarrow ( )[ ()() + (()) ] +
 [ ()() + (()) ] ()
```
Hampi Normalizer Uses Compact Representations Of Expressions

\[ E_2 \quad E_4 \]
\[
() \left[ (()) + ((())) \right] + 
\left[ (()) + ((())) \right] () + 
\left[ ([()()] + ([()()]))) \right] 
\]

shared graph nodes for common subexpressions
Bit Vectors Are Ordered, Fixed-Size, Sets Of Bits

Bit vector B (length 6 bits)


offset:length
Bit Vectors Are Ordered, Fixed-Size, Sets Of Bits

Bit vector B (length 6 bits)

\[
\begin{array}{cccccc}
0 & 1 & 0 & 1 & 0 & 1 \\
\end{array}
\]


Bit-vector solver finds the solution \( B = 010101 \)
Hampi Encodes Core Constraints As Bit-Vector Constraints

Map alphabet $\Sigma$ to bit-vector constants:

$($ → 0
$)$ → 1

Compute size of bit-vector $B$:

$(1+4+1) \times 1 \text{ bit} = 6 \text{ bits}$

$$(v) \in () [ () () + (()) ] + [ () () + (()) ] () + ( [ () () + (()) ] )$$
Hampi Encodes Regular Expressions Recursively

Encode regular expressions recursively
- union $+$ $\rightarrow$ disjunction $\lor$
- concatenation $\rightarrow$ conjunction $\land$
- Kleene star $\ast$ $\rightarrow$ conjunction $\land$
- constant $\rightarrow$ bit-vector constant

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

Formula $\Phi_1$ $\lor$ Formula $\Phi_2$ $\lor$ 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

\[(v) \in (\text{\color{red}{}})[(\text{\color{red}{}}) + (\text{\color{red}{}})] + [(\text{\color{red}{}}) + (\text{\color{red}{}})](\text{\color{red}{}}) + ([(\text{\color{red}{}}) + (\text{\color{red}{}})](\text{\color{red}{}}))\]

\[B[0:2] = 01\]
\[B[5:2] = 01\]

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

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

\[\phi \text{ at offset 2} \quad \phi \text{ at offset 0} \quad \phi \text{ at offset 1}\]

Shift-symmetric constraints
Hampi Uses Bit-Vector Solver And Decodes Solution

bit-vector constraints

Bit-vector Solver
STP

B = 010101

Decoder

B = ( ) ( )

v = ) ( ) ( } 

Maps bits back to alphabet \( \Sigma \)
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≤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 \textbf{6.8x} faster on average (up to \textbf{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**
[ISSST'09]
Ardilla Mutates Generated Inputs To Construct Attacks
SQL Injection Attacks Modify Structure Of Database Queries

Innocuous input:

\[ v \rightarrow 1 \]

\[
SELECT \text{msg}\ FROM\ \text{messages}\ WHERE\ \text{topicid}=\text{'1'}
\]
SQL Injection Attacks Modify Structure Of Database Queries

Innocuous input:

\[
\begin{align*}
v & \rightarrow 1 \\
\text{SELECT msg FROM messages WHERE topicid='1'}
\end{align*}
\]

Symbolic expression for SQL query

\[
\text{concat(SELECT msg FROM messages WHERE topicid='v')}
\]
SQL Injection Attacks Modify Structure Of Database Queries

Innocuous input:
\[ v \rightarrow 1 \]

SELECT msg FROM messages WHERE topicid='1'

Symbolic expression for SQL query
\[
\text{concat}(\text{SELECT msg FROM messages WHERE topicid='v'})
\]

Attack input:
\[ v \rightarrow 1' \text{ OR } '0'='0 \]

SELECT msg FROM messages WHERE topicid='1' \text{ OR } '0'='0'

Attacker gets access to all messages
Example: Hampi Constraints That Create SQL Injection Attacks

user input string

\[
\text{var } v \text{ : 12;} \\
\]

SQL grammar

\[
\text{cfg } SqlSmall := \text{"SELECT } [a-z]+ \text{ " FROM } [a-z]+ \text{ " WHERE } \text{ " Cond;} \\
\text{cfg } Cond := \text{Val } \text{"=} Val \text{ | Cond } \text{ " OR } \text{ " Cond;} \\
\text{cfg } Val := [a-z]+ \text{ | } "" [a-z0-9]* "" \text{ | } [0-9]+; \\
\]

bounded SQL grammar

\[
\text{reg } SqlSmallBounded := \text{bound}(SqlSmall, 53); \\
\]

SQL query

\[
\text{val } q := \text{concat("SELECT msg FROM messages WHERE topicid='"), v, ""');} \\
\]

SQLI attack conditions

\[
\text{assert } q \text{ in } SqlSmallBounded; \\
\text{assert } q \text{ contains "OR '0'='0'";} \\
\]

"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'";
```

Hampi finds an attack input:  \( v \rightarrow 1' \text{ OR } '0'='0 \)
Result: Ardilla Finds New Attacks

60 attacks on 5 PHP applications
23 SQL injection
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

<table>
<thead>
<tr>
<th></th>
<th>Random Fuzz Testing</th>
<th>Concolic Testing</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>50 well-formed seed inputs</td>
<td>50 well-formed seed inputs</td>
</tr>
</tbody>
</table>

Program under test: JavaScript interpreter
Sometimes Concolic Testing Is Not Much Better Than Random Fuzzing

Random Fuzz Testing
50 well-formed seed inputs
8658 inputs

Concolic Testing
50 well-formed seed inputs
6883 inputs
Sometimes Concolic Testing Is Not Much Better Than Random Fuzzing

- **Random Fuzz Testing**
  - 50 well-formed seed inputs
  - 8658 inputs
  - 14.2% coverage

- **Concolic Testing**
  - 50 well-formed seed inputs
  - 6883 inputs
  - 14.7% coverage
Sometimes Concolic Testing Is Not Much Better Than Random Fuzzing

Random Fuzz Testing
- 50 well-formed seed inputs
- 8658 inputs
- 17.6% inputs reach
- 14.2% coverage
- 2 hours

Concolic Testing
- 50 well-formed seed inputs
- 6883 inputs
- 16.5% inputs reach
- 14.7% coverage
Most Generated Inputs Get Rejected Quickly

Parser

seed input
Most Generated Inputs Get Rejected Quickly

Parser

Parser rejects invalid inputs

seed input
Most Generated Inputs Get Rejected Quickly

Parser

seed input
Most Generated Inputs Get Rejected Quickly

Valid inputs reach deeply
Most Generated Inputs Get Rejected Quickly

Key idea: generate only valid inputs
Input–Format Grammar Guides
Creation Of Effective Inputs

Parser

solve(\text{PC}') \Rightarrow \text{new input}

seed input
Input–Format Grammar Guides
Creation Of Effective Inputs

Parser

solve(PC’ ∩ Grammar) ➔ new valid input

Hampi string solver

seed input
String–Constraint Solver Helps Create Valid Inputs

Seed input (for JavaScript interpreter):

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

Constraints on tokens
(created during execution)

```javascript
token_9 = 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):
```javascript
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; }
```
String–Constraint Solver Helps Create Valid Inputs

Seed input (for JavaScript interpreter):

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

Constraints on tokens
(created during execution)

```
token_9 = 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):

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

Constraints on tokens (created during execution)

- `token_9 = function`
- `token_1 = id`
- `token_2 = (`
- `token_3 = )`
- `token_4 ≠ {`

**Normal solver ➔ nonparsable input**

```javascript
function f() var var v = 3; }
```
String–Constraint Solver Helps Avoid Dead–End Inputs

Seed input (for JavaScript interpreter):

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

Constraints on tokens (created during execution)

- `token_0 = function`
- `token_1 = id`
- `token_2 = (`
- `token_3 = )`
- `token_4 ≠ {`

Normal solver ➔ nonparsable input
```javascript
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

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