iGen: Dynamic Interaction Inference for Configurable Software

ThanhVu (Vu) Nguyen*,
Ugur Koc†, Javran Cheng†, Jeffrey S. Foster†, Adam A. Porter†

*University of Nebraska, Lincoln, †University of Maryland, College Park

FSE 2016
Modern software systems are highly-configurable

- Increases flexibility and add features
- But too many configs complicate many analysis tasks
  - Understanding, testing, debugging, etc
  - How configs affect line coverage (the focus of this work)
Interactions in Configurable Systems

Modern software systems are highly-configurable
- Increases flexibility and add features
- But too many configs complicate many analysis tasks
  - Understanding, testing, debugging, etc
  - How configs affect line coverage (the focus of this work)

A precise and compact description of configurations is valuable
- Help developers analyze useful info about configs
  - Find important options affecting program coverage
  - Compute minimal set of configs to achieve high coverage
- Discover such a description is possible in practice (not every config leads to different coverage behaviors)
**Example**

Program with 7 config options

- 6 bools and $z \in \{0, 1, 2, 3, 4\}$
- **Config space:** 320 configs

<table>
<thead>
<tr>
<th>$s$</th>
<th>$t$</th>
<th>$u$</th>
<th>$v$</th>
<th>$x$</th>
<th>$y$</th>
<th>$z$</th>
<th>cov</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>$L_2, L_3$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$L_0, L_1, L_3, L_4, L_5$</td>
</tr>
</tbody>
</table>

//opts: s, t, u, v, x, y, z

```c
int maxz = 3;

if(x && y) {
    printf("L0\n");
    if(!(0 < z && z < maxz))
        printf("L1\n");
else{
    printf("L2\n");
}

printf("L3\n");
if(u && v) {
    printf("L4\n");
    if(s || t){
        printf("L5\n");
    }
}  
```
Program with 7 config options

- 6 bools and $z \in \{0, 1, 2, 3, 4\}$
- **Config space:** 320 configs

<table>
<thead>
<tr>
<th>s</th>
<th>t</th>
<th>u</th>
<th>v</th>
<th>x</th>
<th>y</th>
<th>z</th>
<th>cov</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>$L_2, L_3$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>$L_0, L_1, L_3, L_4, L_5$</td>
</tr>
</tbody>
</table>

Use **interactions** to describe config space

- $x \land y$: $L_0$
- $x \land y \land z \in \{0, 3, 4\}$: $L_1$
- $\overline{x} \lor \overline{y}$: $L_2$
- $u \land v$: $L_4$
- $(u \land v) \land (s \lor t)$: $L_5$

//opts: s, t, u, v, x, y, z

```c
int maxz = 3;
if(x && y) {
    printf("L0\n");
    if(!(0 < z && z < maxz))
        printf("L1\n");
} else{
    printf("L2\n");
}
printf("L3\n");
if(u && v) {
    printf("L4\n");
    if(s || t){
        printf("L5\n");
    }
}
```
Example

Program with 7 config options

- 6 bools and $z \in \{0, 1, 2, 3, 4\}$
- **Config space**: 320 configs

<table>
<thead>
<tr>
<th>s</th>
<th>t</th>
<th>u</th>
<th>v</th>
<th>x</th>
<th>y</th>
<th>z</th>
<th>cov</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>$L2, L3$</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>$L0, L1, L3, L4, L5$</td>
</tr>
</tbody>
</table>

Use **interactions** to describe config space

- $x \land y$: $L0$
- $x \land y \land z \in \{0, 3, 4\}$: $L1$
- $\overline{x} \lor \overline{y}$: $L2$
- $u \land v$: $L4$
- $(u \land v) \land (s \lor t)$: $L5$

Interaction **templates**: **conj** $(x \land y)$, **disj** $(\overline{x} \lor \overline{y})$, **mixed** $(u \land v) \land (s \lor t)$

```c
//opts: s, t, u, v, x, y, z
int maxz = 3;

if(x && y) {
    printf("L0\n");
    if(!(0 < z && z < maxz))
        printf("L1\n");
} else{
    printf("L2\n");
}

printf("L3\n");
if(u && v) {
    printf("L4\n");
    if(s || t){
        printf("L5\n");
    }
}
```
iGen: a dynamic approach to finding interactions wrt line coverage

- Focus on options having finite domains, e.g., boolean, \{0, 64, 128\}
- Scale to large, highly-configurable systems, e.g., httpd: $\geq 2^{50}$ configs
- Language independent, e.g., tested on programs written in C, Perl, Python, OCaml, Haskell
- Work in presence of framework, libraries, and native code
iGen: Overview

- Run program on a set of initial configs, obtain cov info
- For each covered location, infer interactions
- Inferred results may be imprecise (insufficient data), thus create new (counterexamples) configs to refine interactions
- Repeat until can no longer find new interactions or refine existing ones
Demonstration

Interactions

- \( x \land y \): \( L_0 \)
- \( x \land y \land z \in \{0, 3, 4\} \): \( L_1 \)
- \( \overline{x} \lor \overline{y} \): \( L_2 \)
- \( u \land v \): \( L_4 \)
- \( (u \land v) \land (s \lor t) \): \( L_5 \)

//opts: s, t, u, v, x, y, z
int maxz = 3;

if(\( x \land y \)) {
    printf("L0\n");
    if(!(0 < z && z < maxz))
        printf("L1\n");
} else{
    printf("L2\n");
}

printf("L3\n");
if(\( u \land v \)) {
    printf("L4\n");
    if(s || t){
        printf("L5\n");
    }
}
Initial Configurations

Create initial configs having each option value used at least once and obtain cov info

- Contains all possible settings of each individual option
- E.g., all 5 values \( \{0, 1, 2, 3, 4\} \) of \( z \) are used

<table>
<thead>
<tr>
<th>config</th>
<th>s</th>
<th>t</th>
<th>u</th>
<th>v</th>
<th>x</th>
<th>y</th>
<th>z</th>
<th>coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>( c_1 )</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>( L2, L3, L4 )</td>
</tr>
<tr>
<td>( c_2 )</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>( L0, L1, L3 )</td>
</tr>
<tr>
<td>( c_3 )</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>( L2, L3, L4 )</td>
</tr>
<tr>
<td>( c_4 )</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>( L0, L1, L3, L4 )</td>
</tr>
<tr>
<td>( c_5 )</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>4</td>
<td>( L2, L3, L4, L5 )</td>
</tr>
</tbody>
</table>

For each loc, infer interactions using different templates, e.g., \( \text{conj} \ (x \land y) \), \( \text{disj} \ (\overline{x} \lor \overline{y}) \), \( \text{mixed} \ (u \land v) \land (s \lor t) \)
Conjunctive Interactions

The **conj** template

- **conjunctions** of membership constraints
  e.g., \( x \in \{1\} \land y \in \{1\} \land z \in \{0, 3, 4\} \)
  for \( L1 \)

```c
int maxz = 3;
if(x && y) {
    ...
    if(!(0 < z && z < maxz))
        printf("L1\n");
    ...
}
```
Conjunctive Interactions

The **conj** template

- **conjunctions** of membership constraints
  
e.g., \( x \in \{1\} \land y \in \{1\} \land z \in \{0, 3, 4\} \)
  
for \( L1 \)

- Use **pointwise union** \( \cup \) to compute **conj** over configs
  
1. Union the values for each option, e.g., \( s = \{0, 1\} = 0 \lor 1 = \top \)

<table>
<thead>
<tr>
<th>config</th>
<th>s</th>
<th>t</th>
<th>u</th>
<th>v</th>
<th>x</th>
<th>y</th>
<th>z</th>
<th>coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>( c2 )</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>( L0, L1, L3 )</td>
</tr>
<tr>
<td>( c4 )</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>( L0, L1, L3, L4 )</td>
</tr>
</tbody>
</table>

union | \( \top \) | \( \top \) | \( \top \) | \( \top \) | 1 | 1 | 0, 3 |

2. Conjoin the unions to get \( x \land y \land z \in \{0, 3\} \)
Conjunctive Interactions

The conj template

- **conjunctions** of membership constraints
e.g., \( x \in \{1\} \land y \in \{1\} \land z \in \{0, 3, 4\} \)
for \( L1 \)

- Use **pointwise union** \( \sqcup \) to compute **conj** over configs
  1. Union the values for each option, e.g., \( s = \{0, 1\} = 0 \lor 1 = T \)

<table>
<thead>
<tr>
<th>config</th>
<th>s</th>
<th>t</th>
<th>u</th>
<th>v</th>
<th>x</th>
<th>y</th>
<th>z</th>
<th>coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>( c_2 )</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>( L0, L1, L3 )</td>
</tr>
<tr>
<td>( c_4 )</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>( L0, L1, L3, L4 )</td>
</tr>
<tr>
<td>union</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>1</td>
<td>1</td>
<td>0,3</td>
<td></td>
</tr>
</tbody>
</table>

- Conjoin the unions to get \( x \land y \land z \in \{0, 3\} \)

To infer conj for a loc: apply \( \sqcup \) to configs covering that loc
- For \( L1 \), \( c_2 \sqcup c_4 = x \land y \land z \in \{0, 3\} \)

```c
int maxz = 3;
if(x && y) {
    ...
    if(!(0 < z && z < maxz))
        printf("L1\n");
    ...
}
```
Conjunctive Interactions

The \text{conj} template

- \textit{conjunctions} of membership constraints
e.g., \(x \in \{1\} \land y \in \{1\} \land z \in \{0,3,4\}\)

\text{for } L1

- Use \textbf{pointwise union} \(\cup\) to compute \textit{conj} over configs

1. Union the values for each option, e.g., \(s = \{0,1\} = 0 \lor 1 = \top\)

\begin{tabular}{||c|c|c|c|c|c|c|c||}
\hline
config & s & t & u & v & x & y & z & coverage \\
\hline
\hline
c2 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & L0, L1, L3 \\
c4 & 0 & 0 & 1 & 1 & 1 & 3 & 0 & L0, L1, L3, L4 \\
\hline
\hline
union & T & T & T & T & 1 & 1 & 0 & L0, L1, L3, L4 \\
\hline
\end{tabular}

2. Conjoin the unions to get \(x \land y \land z \in \{0,3\}\)

To infer \textit{conj} for a loc: apply \(\cup\) to configs covering that loc

- For \(L1\), \(c2 \cup c4 = x \land y \land z \in \{0,3\}\)
- Almost, but not quite right (\(x \land y \land z \in \{0,3,4\}\))

\begin{verbatim}
int maxz = 3;
define(x && y) {
    ...
    if(!(0 < z && z < maxz))
        printf("L1\n");
    ...
}
\end{verbatim}
For $L_1$, $\text{conj} = x \land y \land z \in \{0, 3\}$

- Need more configs
- E.g., a config having $z = 4$ covering $L_1$

```c
int maxz = 3;
if(x && y) {
    ...
    if(!(0 < z && z < maxz))
        printf("L1\n");
}
...
Interaction Refinement

For $L_1$, $\textit{conj} = x \land y \land z \in \{0, 3\}$

- Need more configs
- E.g., a config having $z = 4$ covering $L_1$

```c
int maxz = 3;
if(x && y) {
    ...
    if(!(0 < z && z < maxz))
        printf("L1\n");
}
...
```

Idea: create new configs to refine existing results

- Select an existing int for some loc
- Systematically change int to create \textit{potentially counterexample} configs (cex's)
Interaction Refinement

For \( L_1 \), \( \text{conj} = x \land y \land z \in \{0, 3\} \)

- Need more configs
- E.g., a config having \( z = 4 \) covering \( L_1 \)

\[
\text{int maxz} = 3;
\text{if}(x \&\& y)\
...\
\text{if}(!(0 < z \&\& z < \text{maxz}))\
\text{printf}("L1\n");
\}
...

Idea: create new configs to refine existing results

- Select an existing int for some loc
- Systematically change int to create potentially counterexample configs (cex’s)

Intuition: if cex’s, which are different than int, can still cover loc, then can use them to refine \textit{int}.
Interaction Refinement: Example

- Pick an existing int to refine, e.g., \( conj = x \land y \land z \in \{0, 3\} \) for \( L1 \)
Interaction Refinement: Example

- Pick an existing int to refine, e.g., \( \text{conj} = x \land y \land z \in \{0, 3\} \) for \( L1 \)
- Systematically change \( \text{conj} \) to create cex’s

<table>
<thead>
<tr>
<th>config</th>
<th>s</th>
<th>t</th>
<th>u</th>
<th>v</th>
<th>x</th>
<th>y</th>
<th>z</th>
<th>coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>( c_6 )</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>( L2, L3 )</td>
</tr>
<tr>
<td>( c_7 )</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>3</td>
<td></td>
<td>( L2, L3 )</td>
</tr>
<tr>
<td>( c_8 )</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>( L0, L3 )</td>
</tr>
<tr>
<td>( c_9 )</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>( L0, L3 )</td>
</tr>
<tr>
<td>( c_{10} )</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>4</td>
<td>( L0, L1, L3 )</td>
</tr>
</tbody>
</table>

- Each cex disagrees with \( \text{conj} = x \land y \land z \in \{0, 3\} \) in exactly one setting, e.g., \( c_6 \) has \( x = 0 \), \( c_7 \) has \( y = 0 \), and \( c_8 \) has \( z = 1 \),..
- Create random settings for other options (e.g., \( s, t, u, v \))
Interaction Refinement: Example

- Pick an existing int to refine, e.g., \( \text{conj} = x \land y \land z \in \{0, 3\} \) for \( L1 \)

- Systematically change \( \text{conj} \) to create cex’s

<table>
<thead>
<tr>
<th>config</th>
<th>( s )</th>
<th>( t )</th>
<th>( u )</th>
<th>( v )</th>
<th>( x )</th>
<th>( y )</th>
<th>( z )</th>
<th>coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>( c_6 )</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>( L2, L3 )</td>
</tr>
<tr>
<td>( c_7 )</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>3</td>
<td>( L2, L3 )</td>
</tr>
<tr>
<td>( c_8 )</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>( L0, L3 )</td>
</tr>
<tr>
<td>( c_9 )</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>( L0, L3 )</td>
</tr>
<tr>
<td>( c_{10} )</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>4</td>
<td>( L0, L1, L3 )</td>
</tr>
</tbody>
</table>

- Each cex disagrees with \( \text{conj} = x \land y \land z \in \{0, 3\} \) in exactly one setting, e.g., \( c_6 \) has \( x = 0 \), \( c_7 \) has \( y = 0 \), and \( c_8 \) has \( z = 1 \), ..

- Create random settings for other options (e.g., \( s, t, u, v \))

- Next iteration, applying \( \cup \) to \( c_2, c_4, c_{10} \) yields \( x \land y \land z \in \{0, 3, 4\} \) (the correct interaction for \( L1 \))
Disjunctive Interactions

The \texttt{disj} template

- E.g., $\overline{x} \lor \overline{y}$ for $L2$
- Cannot apply $\lor$ directly (get \textit{conj}, not \textit{disj})

```c
...  
if(x && y) {
    printf("L0\n");
    ...
} else{
    printf("L2\n");
}
```
Disjunctive Interactions

The disj template

• E.g., \( \overline{x} \lor \overline{y} \) for \( L2 \)
• Cannot apply \( \cup \) directly (get conj, not disj)

Intuition:

• Every loc is either covered or not-covered by a config
• Complement of non-covering configs \( \equiv \) covering configs

\[
\text{\ldots if}(x \&\& y) \{ \\
\quad \text{printf}("L0\n"); \\
\quad \text{\ldots} \\
\} \text{ else}\{ \\
\quad \text{printf}("L2\n"); \\
\} \\
\]
Disjunctive Interactions

The \textit{disj} template

- E.g., $\overline{x} \lor \overline{y}$ for $L_2$
- Cannot apply $\cup$ directly (get \textit{conj}, not \textit{disj})

Intuition:

- Every loc is either covered or \textit{not-covered} by a config
- \textit{Complement of non-covering} configs $\equiv$ covering configs

Idea: apply $\cup$ on \textit{non-covering} configs and negate

```c
... if(x && y) {
    printf("L0\n");
    ...}
} else{
    printf("L2\n");
}
```
Disjunctive Interactions: Example

To infer $disj$ for $L_2$

- Obtain configs $c_2$ and $c_4$ that do not cover $L_2$
  
<table>
<thead>
<tr>
<th>config</th>
<th>s</th>
<th>t</th>
<th>u</th>
<th>v</th>
<th>x</th>
<th>y</th>
<th>z</th>
<th>coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>$c_2$</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>$L_0, L_1, L_3$</td>
</tr>
<tr>
<td>$c_4$</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>$L_0, L_1, L_3, L_4$</td>
</tr>
<tr>
<td>union</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>1</td>
<td>1</td>
<td>0,3</td>
<td></td>
</tr>
</tbody>
</table>

- Compute $c_2 \cup c_4$ to get $conj' = x \land y \land z \in \{0, 3, 4\}$
- Negate $conj'$ to get $disj = \overline{x} \lor \overline{y} \lor z \in \{1, 2\}$ for $L_2$
- (At the end) Check that $disj$ actually satisfies all configs covering $L_2$
Disjunctive Interactions: Example

To infer $\text{disj}$ for $L2$

- Obtain configs $c_2$ and $c_4$ that do not cover $L2$

<table>
<thead>
<tr>
<th>config</th>
<th>$s$</th>
<th>$t$</th>
<th>$u$</th>
<th>$v$</th>
<th>$x$</th>
<th>$y$</th>
<th>$z$</th>
<th>coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>$c_2$</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>$L0, L1, L3$</td>
</tr>
<tr>
<td>$c_4$</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>$L0, L1, L3, L4$</td>
</tr>
</tbody>
</table>

| union | T | T | T | T | 1 | 1 | 0,3 |          |

- Compute $c_2 \cup c_4$ to get $\text{conj}' = x \land y \land z \in \{0, 3, 4\}$
- Negate $\text{conj}'$ to get $\text{disj} = \overline{x} \lor \overline{y} \lor z \in \{1, 2\}$ for $L2$
- (At the end) Check that $\text{disj}$ actually satisfies all configs covering $L2$

- $\cup +$ negation: straightforward extension of $\text{conj}$ inference
- Subsequent iterations create cex’s to refine $\text{conj}'$ and thus $\text{disj}$
- Also used to compute mixed interactions, e.g., $u \land v \land (s \lor t)$ for $L5$
Experiments

- iGen is implemented in Python and uses Z3 to simplify formulae

- 29 subject programs:
  - 9 GNU and 8 Powertool coreutils (e.g., cat, ln, ls),
    10 various progs (e.g., gzip, pandoc, httpd),
    2 progs from prev work (vsftp, ngircd)
  - 5 Languages: C, Perl, Python, OCaml, Haskell
  - Locs: 25 - 250K
  - Options: 2 - 50 binary or finite-domain valued
  - Config spaces: 4 to $1.1 \times 10^{15}$ possible configs
  - Test suites: default tests (if available) + manually created tests

- Evaluation
  - Medians over 21 runs for each program (randomness due to creating initial and new configs)
  - Tested on 2.4 Ghz Intel Xeon, 16 GB RAM, Linux
Correctness: Does iGen produce correct interactions?

Comparing iGen’s iterative algorithm to exhaustive run

- Obtain “ground truths”
  - Create all possible configs for 14 programs with smallest config space
  - Use existing symbolic execution info for vsftpd and ngircd

- Results: iGen produces similar coverage (missed 3 lines) and interaction (92% similarity) results comparing ground truths
Correctness: Does iGen produce correct interactions?

Comparing iGen’s iterative algorithm to exhaustive run

- Obtain “ground truths”
  - Create *all* possible configs for 14 programs with smallest config space
  - Use existing symbolic execution info for vsftpd and ngircd

- **Results**: iGen produces similar coverage (missed 3 lines) and interaction (92% similarity) results comparing ground truths

Manual Inspection on iGen’s results

- Identify several interactions involving *all* options
- Discover mismatched behaviors, e.g., GNU `uname` and Perl `uname`
Efficiency:
How does iGen perform?

Scale well to large programs

- Use a small fraction of total config space, e.g., httpd: $838/10^{15}$
- Much faster than prev symbolic exec work, e.g., vsftpd, ngircd: an hr vs 2 weeks

<table>
<thead>
<tr>
<th>prog</th>
<th>cspace</th>
<th>configs</th>
<th>time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>1,024</td>
<td>157</td>
<td>34</td>
</tr>
<tr>
<td>uname</td>
<td>2,048</td>
<td>95</td>
<td>15</td>
</tr>
<tr>
<td>cat</td>
<td>4,096</td>
<td>131</td>
<td>42</td>
</tr>
<tr>
<td>mv</td>
<td>5,120</td>
<td>106</td>
<td>38</td>
</tr>
<tr>
<td>ln</td>
<td>10,240</td>
<td>213</td>
<td>96</td>
</tr>
<tr>
<td>date</td>
<td>17,280</td>
<td>680</td>
<td>350</td>
</tr>
<tr>
<td>join</td>
<td>18,432</td>
<td>323</td>
<td>158</td>
</tr>
<tr>
<td>sort</td>
<td>6,291,456</td>
<td>1346</td>
<td>3113</td>
</tr>
<tr>
<td>ls</td>
<td>3.5e+14</td>
<td>2175</td>
<td>9837</td>
</tr>
<tr>
<td>p-id</td>
<td>256</td>
<td>82</td>
<td>283</td>
</tr>
<tr>
<td>p-uname</td>
<td>64</td>
<td>28</td>
<td>62</td>
</tr>
<tr>
<td>p-cat</td>
<td>128</td>
<td>26</td>
<td>246</td>
</tr>
<tr>
<td>p-ln</td>
<td>4</td>
<td>4</td>
<td>42</td>
</tr>
<tr>
<td>p-date</td>
<td>3,360</td>
<td>160</td>
<td>2061</td>
</tr>
<tr>
<td>p-join</td>
<td>4,608</td>
<td>111</td>
<td>1573</td>
</tr>
<tr>
<td>p-sort</td>
<td>2,048</td>
<td>116</td>
<td>3947</td>
</tr>
<tr>
<td>p-ls</td>
<td>6.7e7</td>
<td>272</td>
<td>13803</td>
</tr>
<tr>
<td>cloc</td>
<td>524,288</td>
<td>210</td>
<td>5017</td>
</tr>
<tr>
<td>ack</td>
<td>4.3e+9</td>
<td>1347</td>
<td>23127</td>
</tr>
<tr>
<td>grin</td>
<td>2,097,152</td>
<td>242</td>
<td>411</td>
</tr>
<tr>
<td>pylint</td>
<td>5.8e+10</td>
<td>1916</td>
<td>27175</td>
</tr>
<tr>
<td>hlint</td>
<td>8,192</td>
<td>328</td>
<td>9525</td>
</tr>
<tr>
<td>pandoc</td>
<td>4.0e+9</td>
<td>653</td>
<td>23515</td>
</tr>
<tr>
<td>unison</td>
<td>393,216</td>
<td>381</td>
<td>4641</td>
</tr>
<tr>
<td>bibtex2html</td>
<td>1.2e+9</td>
<td>670</td>
<td>667</td>
</tr>
<tr>
<td>gzip</td>
<td>131,072</td>
<td>495</td>
<td>12029</td>
</tr>
<tr>
<td>httpd</td>
<td>1.1e+15</td>
<td>838</td>
<td>197390</td>
</tr>
<tr>
<td>vsftpd</td>
<td>2.1e+9</td>
<td>620</td>
<td>652</td>
</tr>
<tr>
<td>ngircd</td>
<td>29,764</td>
<td>650</td>
<td>1469</td>
</tr>
</tbody>
</table>
Analysis: What can we learn from iGen’s results?

- Interactions are rare (far less than number of possible *ints*)
  - E.g., iGen discovers 4 *ints* for p-cat, which has 4373 possible *ints*
  - Overall $\leq 0.1\%$ than number of possible interactions
Analysis: What can we learn from iGen’s results?

- Interactions are rare (far less than number of possible *ints*)
  - E.g., iGen discovers 4 *ints* for p-cat, which has 4373 possible *ints*
  - Overall ≤ 0.1% than number of possible interactions

- Longer *conj* tend to be built on shorter *conj*
  - E.g., if $a \land b \land c \land d$ is an interaction, then $a \land b$ is also likely an *int* (potentially due to nested guards)
  - For most programs, *conj* of length $\geq 3$ include a shorter *int*
Analysis: What can we learn from iGen’s results?

- Interactions are rare (far less than number of possible *ints*)
  - E.g., iGen discovers 4 *ints* for p-cat, which has 4373 possible *ints*
  - Overall $\leq 0.1\%$ than number of possible interactions

- Longer conj tend to be built on shorter conj
  - E.g., if $a \land b \land c \land d$ is an interaction, then $a \land b$ is also likely an *int* (potentially due to nested guards)
  - For most programs, conj of length $\geq 3$ include a shorter int

- Most cov achieved by shorter *ints*, but longer *ints* needed for max cov
  - 87% of coverage is obtained by *ints* of length less than 3
  - 5 programs (*id, uname, cat, p-join, httpd*) have interactions involving all options
Analysis: What can we learn from iGen’s results?

- Many *enabling* options (options set in certain ways for high cov), e.g.,
  - vsftpd, disabling ssl and local and enabling anon are important to cov
  - httpd requires both -enable-http and -enable-so (shared modules)
Analysis: What can we learn from iGen’s results?

- Many _enabling_ options (options set in certain ways for high cov), e.g.,
  - vsftpd, disabling ssl and local and enabling anon are important to cov
  - httpd requires both _-enable-http_ and _-enable-so_ (shared modules)

- Disjunctive and mixed interactions are required
  - Appear in in 26/29 benchmark programs
  - Approx 20% of inferred interactions are _disj_ and _mixed_ interactions
Analysis: Minimal Covering Configs

Minimal covering configs

- Use inferred interactions to compute small sets of configs achieving full cov found by iGen
- E.g., only 2/320 configs needed to cover all lines \( L_0 \) – \( L_5 \) in example
- Develop a *greedy algorithm* that combines compatible interactions to create high-cov configurations

**Result:** achieve very small minimal config sets
Analysis: Minimal Covering Configs

Minimal covering configs

- Use inferred interactions to compute small sets of configs achieving full cov found by iGen
- E.g., only 2/320 configs needed to cover all lines $L_0 - L_5$ in example
- Develop a greedy algorithm that combines compatible interactions to create high-cov configurations

Result: achieve very small minimal config sets
Summary

**iGen**: a light weight approach to analyze interactions

- Use dynamic analysis to infer interactions
- Generate new (cex) configurations to refine results
- Efficiently compute different kind of interactions (conj, disj, mixed)

**Evaluation**

- Work on highly-configurable software systems in a variety of languages
- Infer precise interactions using a very small number of configs
- Confirm hypotheses about configurable software
- Config space can be effectively described a small number of ints
- Longer conj's often built on shorter ones
- Most cov achieved by shorter ints, but longer ints needed for max cov
- Enabling options and expressive interactions are necessary
Summary

**iGen**: a light weight approach to analyze interactions
- Use dynamic analysis to infer interactions
- Generate new (cex) configurations to refine results
- Efficiently compute different kind of interactions (*conj*, *disj*, *mixed*)

**Evaluation**
- Work on highly-configurable software systems in a variety of languages
- Infer **precise** interactions using a very small number of configs
- Confirm hypotheses about configurable software
  - Config space can be effectively described a small number of *ints*
  - Longer *conj’s* often built on shorter ones
  - Most cov achieved by shorter *ints*, but longer *ints* needed for max cov
  - Enabling options and expressive interactions are necessary