Computing Summaries of String Loops in C for Better Testing and Refactoring

Timotej Kapus, Oren Ish-Shalom, Shachar Itzhaky, Noam Rinetzky, Cristian Cadar
char *p;
for (p = line; p && *p && whitespace (*p); p++)
;
```c
char *p;
for (p = line; p && *p && whitespace (*p); p++)
    ;
```

```c
char *p = line + strspn(line, "\t")
```
Why?

- Give clarity to the meaning of loops
- Refactoring
- Program analysis
  - Symbolic execution
- Compiler optimisations
Motivation: Refactoring

```c
char *p = path + strlen(path);
for (; *p != '/' && p != path; p--)
    ;
```
Motivation: Refactoring

```c
char *p = path + strlen(path);
for (; *p != '/' && p != path; p--);

char *p = strrchr(path, '/');
p = p == NULL ? path : p;
```
Motivation: Refactoring

- Real examples from `wget` and `bash`
- C code contains lots of loops replicating libc functions
  - Different handling of edge cases
Motivation: Program analysis

- Easier to reason about a known function than an arbitrary loop

*Example* symbolic execution of `strchr`

Two approaches:

1. Unroll loop and gather constraints character by character
2. Model it as `indexOf` in theory of strings
Scope: Memoryless Loops

- Loops conforming to an interface:
  - Argument: single pointer to a buffer
  - Returns: pointer to an offset in the buffer
- Only reads the character under current pointer

```c
char* loopSummary(char*);
```

- Need a vocabulary to express these loops
Remember?

```c
char *p = line + strspn(line, "\n\t")
```
In our vocabulary

```c
char *p = line + strspn(line, "\t"");
```

Loop summary!
We just used characters!

```c
char *p = line + strspn(line, "\0\t")
```

Loop summary!
Vocabulary for expressing simple loops

- Vocabulary has meaning in an interpreter()
- `strspn(P)` and `return(F)`
- Adding a new vocabulary as simple as adding a new `case:`

```c
char* interpreter(char* input) {
    char* result = input;
    while (nextInstruction())
        switch (instruction)
            case 'P':
                result += strspn(result, data(instruction));
            case 'F':
                return result;
    }
```
## Vocabulary for expressing simple loops

<table>
<thead>
<tr>
<th>string.h functions</th>
<th>pointer manipulation</th>
<th>conditionals</th>
</tr>
</thead>
<tbody>
<tr>
<td>• <code>strspn</code></td>
<td>• increment</td>
<td>• is null</td>
</tr>
<tr>
<td>• <code>strcspn</code></td>
<td>• set to start</td>
<td>• is start</td>
</tr>
<tr>
<td>• <code>memchr</code></td>
<td>• set to end</td>
<td></td>
</tr>
<tr>
<td>• <code>strchr</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td>• <code>strrchr</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td>• <code>strpbrk</code></td>
<td>special</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• backward traverse</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• return</td>
<td></td>
</tr>
</tbody>
</table>
Loop Summarisation

*Find sequences of characters that when executed by our interpreter have the same behaviour as the original loop*
Counter-example guided synthesis

- Loop to summarize

- Generate a sequence of characters fitting all counterexamples

- Synthesizer

- Verifier

- Fail - generate counterexample

- Success

- Done
Synthesizer

● Symbolic execution
● Use a symbolic string (program)
● Constrain it to be equivalent on current counterexamples
● Ask an SMT solver for a solution

Verifier

● Symbolic execution
   ○ Bounded equivalence checking strings of length $\leq 3$
● Loops in our scope
   ○ checking lengths $\leq 3$ sufficient to show equivalence for any length (proof in the paper)
**Synthesizer**
- Symbolic execution
- Use a symbolic string (program)
- Constrain it to be equivalent on current counterexamples
- Ask an SMT solver for a solution

**Verifier**
- Symbolic execution
  - Bounded equivalence checking strings of length $\leq 3$
- Loops in **our scope**
  - checking lengths $\leq 3$ sufficient to show equivalence for any length (proof in the paper)

---

Single run of symbolic execution
char *p;
for (p = line;
    p && *p && whitespace (*p);
    p++)
;

CEX: []
char *p;
for (p = line;
    p && *p && whitespace (*p);
    p++)
;

return result;

Program: F

Synthesizer

Verifier

CEX: []
Counterexample: " [] "

```c
for (p = line; *p && p ++);
```

return result;
char *p;
for (p = line;
p && *p && whitespace (*p);
p++)
;

result += strspn(result, "\n");
return result;

Program: P \0F

CEX: ["\n"]
char *p;
for (p = line; p && *p && whitespace(*p); p++)
{

result += strspn(result, "\t");
return result;
}

CEX: ["\t"]

Counterexample: "\t"
char *p;
for (p = line;
    p && *p && whitespace (*p);
    p++)
;

result += strspn(result, "_\t");
return result;

Program: P _ \t \0F

Synthesizer

Verifier

CEX: ["_", "\t"]
char *p;
for (p = line;
    p && *p && whitespace (*p);
    p++)
    ;

result += strspn(result, "\t\n");
return result;

CEX: [" ", "\t"]

P \t\n\n25
Synthesis Evaluation

- 13 open source programs
- Semi-automatic process
- Extracted 115 loops fitting our scope
- In total 88/115 synthesised
2h/loop synthesis timeout: 77/115 loops
Impact of timeout and program size
Vocabulary optimisation

- Find a subset of vocabulary that synthesises more loops
- Gaussian process optimization
- 5 minute timeout
- 81/115 loops with 5min timeout
- 7 additional loops found with full vocabulary and 2h timeout

88/115 total

Best performing vocabulary:
- strspn
- strcspn
- memchr
- increment
- backward traverse
- return
Improving symbolic execution

- Use loop summaries to gather more efficient constraints
- Intercept calls to string functions and encode them in theory of strings
- Compare with character by character constraints
  - Theory of strings should have an advantage for longer strings
- Implemented in KLEE
- Compared (only) on the loops we extracted
Improving symbolic execution

![Graph showing the mean time (s) for vanilla.KLEE and str.KLEE against symbolic string length. The graph shows a significant improvement in mean time for str.KLEE as the symbolic string length increases.]
Improving symbolic execution

Loop rank

Speedup

1000x
100x
10x
1x
Compiler optimisation potential?

- Compare the loop summaries (libc library functions) with compiled loops
Refactoring

- Use summaries to create patches and send them to developers
- Developers of libosip, patch and wget accepted the patches

```c
- for(; *tmp == ' ' || *tmp == '\t'; tmp++)
- }
- for(; *tmp == '\n' || *tmp == '\r'; tmp++)
- }                         /* skip LWS */
+ tmp += strspn(tmp, " \t");
+ tmp += strspn(tmp, "\n\r");
```
Conclusion

- Counterexample guided synthesis based technique for summarisation of simple loops in C
- 88/115 loops synthesized
- Applications:
  - Program analysis (symbolic execution)
  - Compiler optimisations
  - Refactoring
2h/loop synthesis timeout: 77/115 loops
<table>
<thead>
<tr>
<th>utility</th>
<th>Total loops</th>
<th>Inner loops</th>
<th>Loops without pointer call</th>
<th>Read only loops</th>
<th>Loops with a read from single pointer</th>
</tr>
</thead>
<tbody>
<tr>
<td>bash</td>
<td>1085</td>
<td>944</td>
<td>438</td>
<td>264</td>
<td>45</td>
</tr>
<tr>
<td>diff</td>
<td>186</td>
<td>140</td>
<td>60</td>
<td>40</td>
<td>14</td>
</tr>
<tr>
<td>gawk</td>
<td>608</td>
<td>502</td>
<td>210</td>
<td>105</td>
<td>17</td>
</tr>
<tr>
<td>git</td>
<td>2904</td>
<td>2598</td>
<td>725</td>
<td>495</td>
<td>108</td>
</tr>
<tr>
<td>grep</td>
<td>222</td>
<td>172</td>
<td>72</td>
<td>42</td>
<td>9</td>
</tr>
<tr>
<td>m4</td>
<td>328</td>
<td>286</td>
<td>126</td>
<td>78</td>
<td>12</td>
</tr>
<tr>
<td>make</td>
<td>334</td>
<td>262</td>
<td>129</td>
<td>102</td>
<td>13</td>
</tr>
<tr>
<td>patch</td>
<td>207</td>
<td>172</td>
<td>88</td>
<td>67</td>
<td>20</td>
</tr>
<tr>
<td>sed</td>
<td>125</td>
<td>104</td>
<td>35</td>
<td>19</td>
<td>1</td>
</tr>
<tr>
<td>ssh</td>
<td>604</td>
<td>544</td>
<td>227</td>
<td>84</td>
<td>12</td>
</tr>
<tr>
<td>tar</td>
<td>492</td>
<td>432</td>
<td>155</td>
<td>106</td>
<td>33</td>
</tr>
<tr>
<td>torture_test</td>
<td>100</td>
<td>95</td>
<td>39</td>
<td>30</td>
<td>25</td>
</tr>
<tr>
<td>wget</td>
<td>228</td>
<td>197</td>
<td>115</td>
<td>83</td>
<td>14</td>
</tr>
<tr>
<td>SUM</td>
<td>7423</td>
<td>6448</td>
<td>2419</td>
<td>1515</td>
<td>323</td>
</tr>
<tr>
<td>Issue</td>
<td>Count</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-------------------------------</td>
<td>-------</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Has Goto</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IOsideeffects</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Non Pointer Return</td>
<td>74</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Return In Loop</td>
<td>70</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Too Many Arguments</td>
<td>28</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Too Many Return Values</td>
<td>31</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SUM</td>
<td>208</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Impact of timeout and program size - 30s timeout
Impact of timeout and program size

Number of synthesized programs

Program size

30s  3min
Impact of timeout and program size