Partial Replay of Long-Running Applications

Alvin Cheung
Armando Solar-Lezama
Sam Madden

MIT CSAIL
Bugs are difficult to reproduce

"It had never evidenced itself until that day," said spokesman Ralph DiNicola. "This fault was so deeply embedded, it took them weeks of poring through millions of lines of code and data to find it."

[the plots] show that fixing 50% of the bugs requires appx. 100 to 300 days ... The median bug-fix time is about 200 days.

<table>
<thead>
<tr>
<th>App</th>
<th># bugs can’t be reproduced in bugzilla</th>
</tr>
</thead>
<tbody>
<tr>
<td>gnome</td>
<td>4528</td>
</tr>
<tr>
<td>mysql</td>
<td>4175</td>
</tr>
<tr>
<td>gentoo</td>
<td>2011</td>
</tr>
<tr>
<td>redhat / fedora</td>
<td>2623</td>
</tr>
<tr>
<td>firefox</td>
<td>1367</td>
</tr>
<tr>
<td>apache</td>
<td>297</td>
</tr>
</tbody>
</table>
Reproducing Bugs

• Ask user for buggy input

• Guide model checker to find execution trace
  – Non-trivial effort and time

• Use software replayer
Software Replay
Software Replay

Logging slows program down

Replaying large logs take time

Log gets large

User site

Developer site
bbr to the rescue!
Replayer Wishlist

- Small runtime overhead
- Small log size
- Fast replay time
bbr: A Branch Deterministic Partial Replayer

*Small runtime overhead / Small log size*

- Record only branches and dynamic array indices
  - Huge log size reduction for data-intensive apps

*Fast replay time*

- Replay fragment of original execution
- Find execution trace that follows the same control flow path as the original
  - We call that a *branch-deterministic* trace
- Use symbolic execution to find execution trace
  - We call this a *partial symbolic* replay
Running Example

- memcached.c, commit f1f4aec

```c
char *do_add_delta (item *it, const int64_t delta)
{
    int64_t value = ITEM_data(it);
    int incr = parse_command(...);
    ...
    if (incr) {
       ...
    }
    else {
       value -= delta;
       if (value < 0) {
          value = 0;
       }
    }
    ...
```

What if value was negative?

value and delta originally >= 0
while (!shutdown) {
  ...
  bbr_checkpoint();
  process_req();
}

Using bbr to replay

User site

Developer site

process_req()
1
2
2123
process_req()
...

log

ex

solver

replayer

symbolic constraints

execution trace
Symbolic Execution Example

Code

```c
char *do_add_delta (item *it,
    const int64_t delta)
{
    int64_t value = ITEM_data(it);
    int incr = parse_command(…)

    if (incr) {
        ...
    }
    else {
        value -= delta;

        if (value < 0) {

            value = 0;
        }
    }
}
```

Symbolic State

- value $\rightarrow$ symVar1
- incr $\rightarrow$ symVar2

Log: branch not taken
Add constraint:

```
symVar2 \neq 0
```

- delta $\rightarrow$ symVar3
- value $\rightarrow$ symVar1 - symVar3

Log: branch taken
Add constraint:

```
symVar1 - symVar2 < 0
```

value $\rightarrow$ 0
bbr internals
Modeling Memory
Continuum of memory models

Entirely symbolic

- Any symbolic variable can be an address
- Rely on solver to find actual values for addresses

✔ No need to explicitly keep track of aliases

✘ Generate huge constraints with long solve times
✘ Not scalable to replay long executions
Modeling Memory

Continuum of memory models

- All addresses must be concrete values
- Needs complete alias knowledge

- Extremely efficient and scalable
- Can’t do this due to partial replay!
Why do we not have complete alias information?

- Allow replaying of execution fragments
- Access memory locations allocated prior to start of replay
  - We don’t know what they point to and their aliasing information
  - Make assumptions about possible aliases
  - Explicitly keep track of may-aliases
  - Ask solver to solve for the actual aliases
Modeling Memory: Our Approach

Continuum of memory models

- More scalable
- Works for partial replay

- Can’t replay bugs that rely on unsafe memory operations such as buffer overruns
  
  There are many other tools that target those bugs
Parallel Solver

• Constraints consist of independent groups
  – i.e., do not share any variables

• Split constraints and solve in parallel
  – Substantial savings in solve time
Experiments
Goals

• Runtime overhead
• Log growth rates
• Ability to replay real-world bugs
• Effectiveness of parallel solving
Overhead Experiment

• 6 different long-running apps
• Compared time overhead of native versus 4 different logging mechanisms
  
  Replay from beginning:
  
  – *non-det*: log all non-deterministic data
  – *loads*: log values of unpredictable memory loads
  
  Partial replay:
  
  – *bbr*: truncates log after N requests
  – *snapshot*: core dumps every N requests + log all non-deterministic data in between
Overhead Experiment

Normalized overhead

sqlite  memcached  tcpdump  betaftp  thttpd  ghttpd

bbr  non-det  snapshot  loads
Discussion

- bbr has the lowest time overhead and log growth rate
  - Partially due to data-intensive nature of apps

- Results on CPU-intensive apps were not as good
  - Apps executed many branches
Bug Replay Experiment

• Replayed a total of 11 different real bugs

<table>
<thead>
<tr>
<th>Bug</th>
<th>LOC</th>
<th># constraints</th>
<th>Solve time</th>
</tr>
</thead>
<tbody>
<tr>
<td>sqlite cast</td>
<td>2.4M</td>
<td>86k</td>
<td>5hr</td>
</tr>
<tr>
<td>memcached CAS</td>
<td>24k</td>
<td>1705</td>
<td>158s</td>
</tr>
<tr>
<td>tcpdump ISIS</td>
<td>61M</td>
<td>2.3M</td>
<td>5s</td>
</tr>
<tr>
<td>thttpd defang</td>
<td>514k</td>
<td>21k</td>
<td>2s</td>
</tr>
<tr>
<td>ghttpd CGI</td>
<td>352k</td>
<td>8k</td>
<td>2s</td>
</tr>
</tbody>
</table>

• Variety of bugs were replayed
## Constraint Splitting Experiment

- Replayed different # of requests for two web servers and compared solve times

<table>
<thead>
<tr>
<th>App (# requests)</th>
<th># constraints</th>
<th># groups</th>
<th>Single solve time</th>
<th>Parallel solve time</th>
</tr>
</thead>
<tbody>
<tr>
<td>ghttpd (10)</td>
<td>20383</td>
<td>823</td>
<td>20s</td>
<td>1s</td>
</tr>
<tr>
<td>ghttpd (50)</td>
<td>60384</td>
<td>2314</td>
<td>32min</td>
<td>6min</td>
</tr>
<tr>
<td>betaftp (10)</td>
<td>1534</td>
<td>106</td>
<td>2s</td>
<td>1s</td>
</tr>
<tr>
<td>Betaftp (50)</td>
<td>13223</td>
<td>530</td>
<td>40min</td>
<td>13min</td>
</tr>
</tbody>
</table>

- Significant difference in solve times
bbr: a partial replayer using symbolic execution

Low overhead
Small log sizes
Reproduce real-world bugs