**Westworld**: Fuzzing-Assisted Remote Dynamic Symbolic Execution of Smart Apps on IoT Cloud Platforms

Lannan Luo, Qiang Zeng, Bokai Yang, Fei Zuo, and Junzhe Wang

University of South Carolina
Motivation

• On platforms such as SmartThings, official smart apps are manually reviewed.

• Many community members enjoy writing custom smart apps and share them in the SmartThings community forum so that others can use them, which however does not enforce code review.

• Smart apps tend to have bugs.

• Automated testing of smart apps for bug discovery is critical needed.
Current Method of Testing Smart Apps

- Step 1: fill app configurations (user inputs)
Current Method of Testing Smart Apps

- Step 2: select environment inputs

Developers read logs to find bugs
Symbolic Execution

• Symbolic execution is a promising automatic testing technique for finding bugs.

• While many symbolic executors have been proposed for analyzing Windows programs, Linux programs and Java programs, none support the analysis of IoT apps.

• Due to unique characteristics of IoT platforms, multiple challenges exist for symbolically executing IoT apps.
Challenges, Solutions, and Goals

**C1**: Remote Cloud-based environment

**S1**: Remote dynamic symbolic execution

**C3**: Communication cost due to remote execution

**S3**: Boosted generational search

**G**: Efficiency

**G**: Precision

**G**: Completeness

**S2**: Selective code-segment fuzzing

**C2**: Closed-source Platform APIs

Missing execution paths
System Architecture

Code instrumentation
- Instrumentation for path condition collection
- Instrumentation for selective code-segment fuzzing

Path analysis
- Identifying code segment for fuzzing
- Generating test cases

Web interaction

Local ⇐ ⇒ Remote

IoT cloud

(1) PC-collection app
(4) seg-fuzzing app
(2) Results from PC-collection app
(5) Results from seg-fuzzing app
(3)
(6)
An Example

- Return value of a platform API is assigned as a temporary symbolic variable (TSV)
- **Selective code-segment fuzzing:** find out the relation between a TSV and symbolic inputs that it relies on, called *influential symbolic inputs* (ISI)
- **Our insight:** most symbolic inputs usually have a small to moderate number of possible values. E.g., “humidity” has 101 integer values between 0 and 100.
- A for-loop is inserted to iterate over values of ISIs and learn the relation between the TSV and ISIs.
- The relation is combined with symbolic path condition to generate test cases
Comparison with Driller

• **Westworld**: symbolic execution-centric
• **Driller**: fuzzing-centric

**Reason of our design choice**: The communication cost between the remote cloud and local analyzer cannot be omitted. Each testing request is expensive.

• E.g., given a path like (temp<75 && temp>68), Driller cannot avoid generating a lot of testing requests that repetitively take the same path, while symbolic execution is good at this.
Evaluation

- We evaluate Westworld in five aspects: feasibility, completeness, precision, efficiency, and effectiveness in bug finding.

- Three Datasets.
  - *Dataset-I* includes 136 official (84) and third-party (52) apps randomly collected from the SmartThings GitHub repo.
  - *Dataset-II* includes 64 hand-crafted apps with more paths and more complex conditional statements.
  - *Dataset-III* has 8 apps with different types of bugs inserted by us.
Completeness

Table 2: Completeness result (%) (full path coverage is attained by WESTWORLD after a minor implementation change).

<table>
<thead>
<tr>
<th># of paths in apps</th>
<th>WESTWORLD Dataset-I</th>
<th>Dataset-II</th>
<th>Fuzzer Dataset-I</th>
<th>Dataset-II</th>
<th>Concolic executor Dataset-I</th>
<th>Dataset-II</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Max</td>
<td>Min</td>
<td>Avg</td>
<td>Max</td>
<td>Min</td>
<td>Avg</td>
</tr>
<tr>
<td>≥ 20</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>[15, 20)</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>[10, 15)</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>[5, 10)</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>[2, 5)</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
</tr>
</tbody>
</table>

- **Grey-box fuzzer** adopts the coverage-guided input generation technique used in American Fuzzy Lop (AFL).

- **Concolic executor** considers user inputs and environment variables as symbolic inputs (the same as Westworld), but does not apply selective code-segment fuzzing to improve path coverage.
Efficiency

- **W-vanila** executes each test case through one testing request.
- **W-boost** executes all test cases of one generation via one testing request.
Bug Finding

• We apply Westworld to four types of bugs: (1) division by zero, (2) array out of bound, (3) null-pointer dereference, and (4) dead code.

• In Dataset-I, we found 4 apps with null-pointer dereference bugs

• Dataset-III contains 8 apps with different bugs. Westworld can successfully find all the bugs.
  - (1) two apps contain division by zero bugs, (2) four are inserted with dead code, (3) one contains an array out of bound bug, and (4) one contains a null-pointer dereference bug.
Summary

• We have presented the first system that enables dynamic symbolic execution (DSE) of smart apps.

• Exploiting the uniqueness of environment inputs, selective code-segment fuzzing was proposed to assist DSE.

• We implemented Westworld, which performs fuzzing-assisted DSE-centric analysis of smart apps.

• The evaluation shows that Westworld is effective and efficient in path exploration and bug finding.
THANK YOU VERY MUCH