Property Directed Test Case Generation

Dr. Arie Gurfinkel

Dr. Edward Schwartz

Dr. William Klieber

Jeffrey Gennari

Disclaimer

This material is based upon work funded and supported by DoD/SEI LENS under Contract No. FA8721-05-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center sponsored by the United States Department of Defense.

DM-0004040

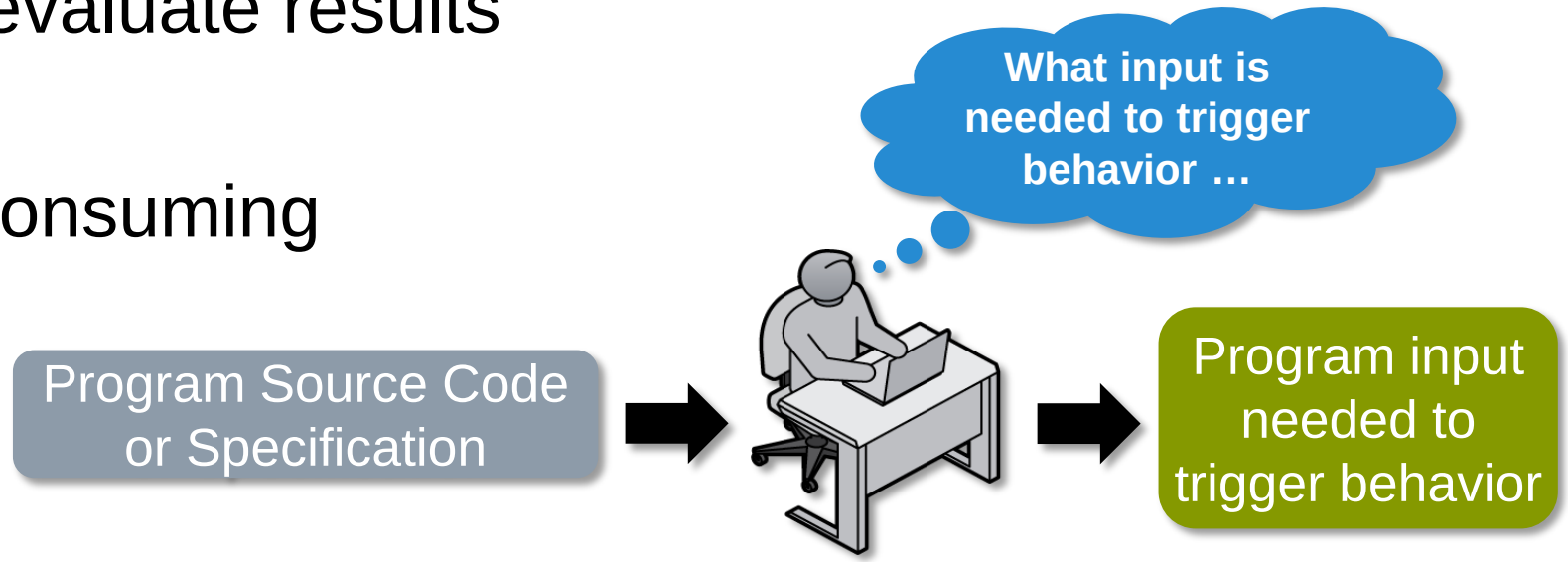
Big Picture: Generating Test Case Input is Complex

Software evaluations often include creating test cases to trigger interesting behaviors

- Finding bugs, ensuring critical functionality executes, determining if unwanted functionality is present, etc.

Traditional test case generation requires testers to provide specific inputs, run tests, and evaluate results

- Trial and error
- Complex and time consuming



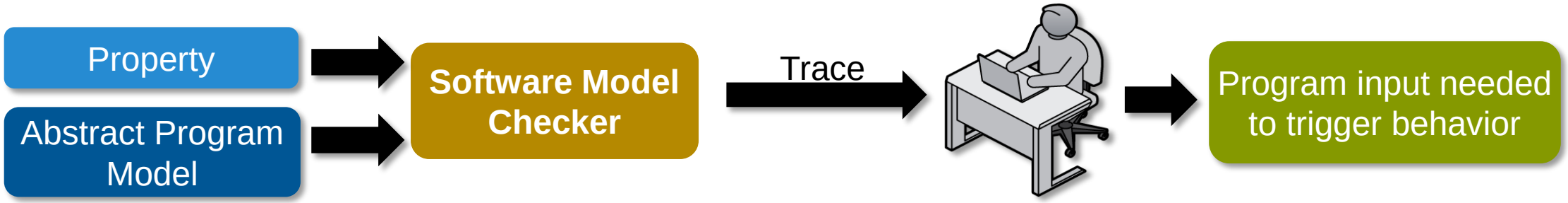
Test Case Generation and Model Checking

Model checkers use program abstractions to exhaustively check a program for specified properties

- For example, can the system get into a bad state; if so, how?

Model checkers produce traces to show how a property is, or is not satisfied

- Required program models very abstract
- Specified properties often expressed in complex, mathematical terms
- Returned traces are abstract and hard to operationalize



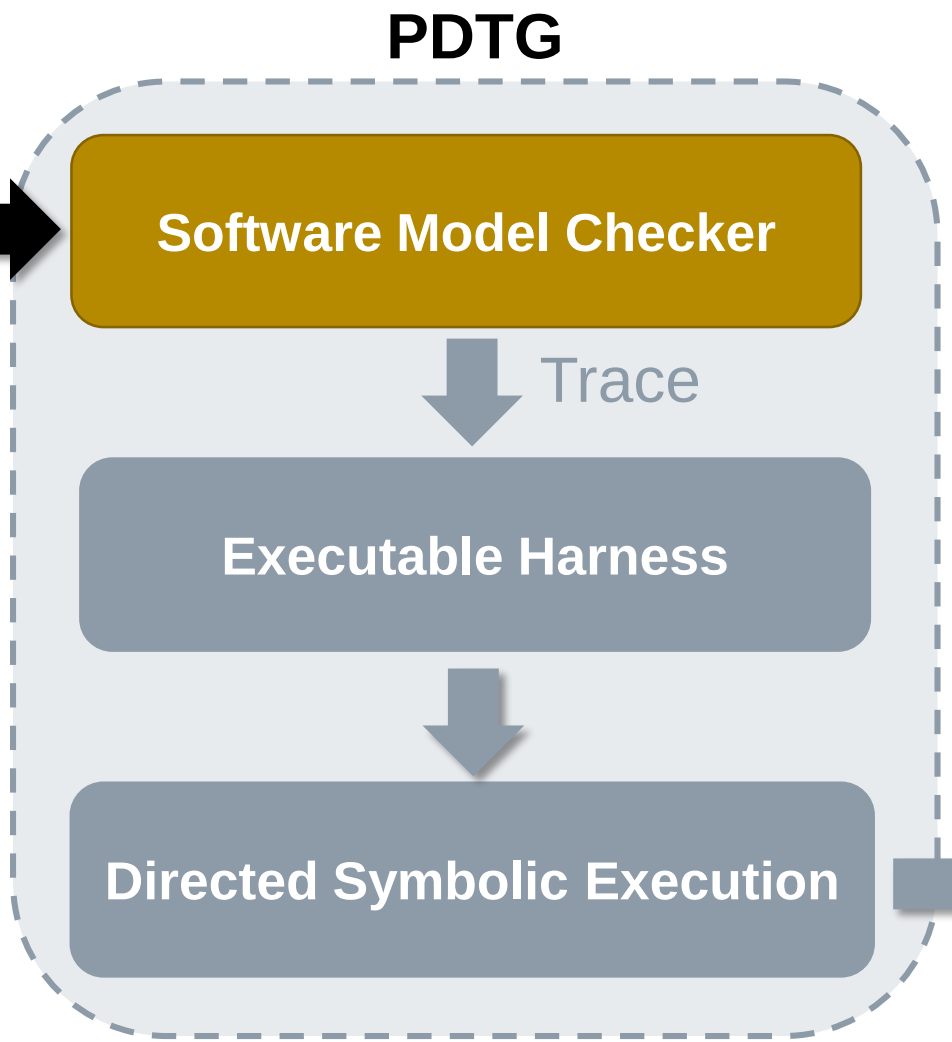
Property Directed Test Case Generation (PDTG)

Goal: Instead of relying on human-driven trial and error or interpreting abstract results from model checkers we will automatically generate executables to trigger desired behavior

- Provide evaluators with *concrete evidence* demonstrating a property is present
- Give evaluators an *executable artifact* showing how to trigger the property

Property Directed Test Case Generation (PDTG)

Program's source code augmented to flag areas of interest



Model checker produces counter example (trace) showing how to reach property

- Seahorn
  - http://seahorn.github.io/

Property Directed Test Case Generation (PDTG)

Program's source code augmented to flag areas of interest

PDTG

Software Model Checker

Trace

Executable Harness

Directed Symbolic Execution

Executable

Executable harness implements external methods needed to execute path in trace

KLEE fuses together trace with values from executable harness to produce valid executable

- Resulting executable deterministically triggers property

Example Harness

```
if (get_input() == 0x1234 &&
    get_input() == 0x8765) {
  __VERIFIER_error();
} else {
    return 0;
}
```

```
void get_input () {
  static int x = 0;
  switch (x++) {
    case 0: return 0x1234;
    case 1: return 0x8765;
    default: assert(false); }
}
```

- `get_input()` is an external function
- Program considered buggy if and only if `__VERIFIER_error()` is reachable
- Implementation of external functions linked to original source code
- Causes program to execute `__VERIFIER_error()`

Generating Harnesses for Linux Device Drivers

```
void *ldv_undef_ptr(void)
{
  void *tmp;
  tmp = __c();
  return tmp;
}
////////////////////////
void *is_got =
ldv_undef_ptr();
if (is_got <= (long)2012) {
... }
```

- Samples from Linux Driver Verification (LDV) project
- Harness functions returning pointers are tricky
  - May not be reasonable addresses
  - Might return “new” memory
- Original program instrumented with memory read/store hooks that control access to external memory
  - Still working on general solution

Malware Case Study

PDTG useful for malware analysis

- Force execution of suspected malicious code
  - For example, generate an executable to trigger malware's information stealing features
- Automatically construct program entry point to call function(s) needed to trigger behavior of interest
  - Set all conditions required to reach behavior
- Tested on Gh0st RAT malware

Generate main function to trigger behavior

```
define i32 @main()
entry
%0 = call %class.CSystemManager @__sea_get_arg()
call void @_ZN14CSystemManager9OnReceiveEPhj(%class.CSystemManager* %0)
ret i32 0
```

Conclusions

We implemented the PDTG prototype to discover interesting behaviors, generate harnesses, and produce working executables

- Completed malware case study demonstrating usefulness of approach
- Evaluated PDTG using Linux Driver Verification (LDV) Benchmarks
  - http://forge.ispras.ru/projects/ldv

Results based on 27 files in ldv-validator-0.8	
File Count	**Results**
6	Successfully generated executable to demonstrate property of interest (i.e. trigger known bug)
7	Successfully proved buggy code was unreachable
6	Timeout
5	Memory exhausted
3	Bogged down in complex memory operations

Acknowledgements

We would like to thank our collaborators on this project

- Carnegie Mellon University
  - Dr. Temegshen Kahsai
  - Dr. Limin Jia
  - Jiaqi Liu
- NASA
  - Dr. Jorge Navas