Detecting Trigger-Based Behavior in Embedded Applications

Dorottya Papp

© 2016 Dorottya Papp

Trigger-Based Behavior (aka Logic Bomb)

- Hidden malicious functionality in the application that executes only under specific conditions

- Specific conditions (=trigger) may be:
  - Special user name/password → backdoor
  - Sequence of ports in arriving packets → portknocking
  - Time and date → time-bomb

- Examples
  - CIA blasted Siberian gas pipeline with a time bomb (1982, confirm: 2012)
  - Logic bomb took out ATM infrastructure in South Korea (2013)

Detection is Challenging

- Penetration testing: subject the system to attacks in a controlled environment
  - Unlikely to supply the correct trigger
- Dynamic testing: focuses on normal conditions
  - Unlikely to be triggered under normal conditions
- Static analysis: can uncover it manually
  - Human resource is costly
  - Humans are unreliable

- **Can the process be (semi-)automated?**
  - Automated: computer categorizes the SW as clean/not
  - Semi-automated: computer categorizes as suspicious/not, human analyst confirms → steers manual analysis to places of interest

Approach

- Goal: categorize the conditions that affect the execution
- How can the conditions be uncovered? **Symbolic execution**
  - Resource-exhaustive → does not scale well
  - Source not available → does not work (e.g. library calls, kernel functions)
  - Unsolvable constraint → tools hang/abort

Approach

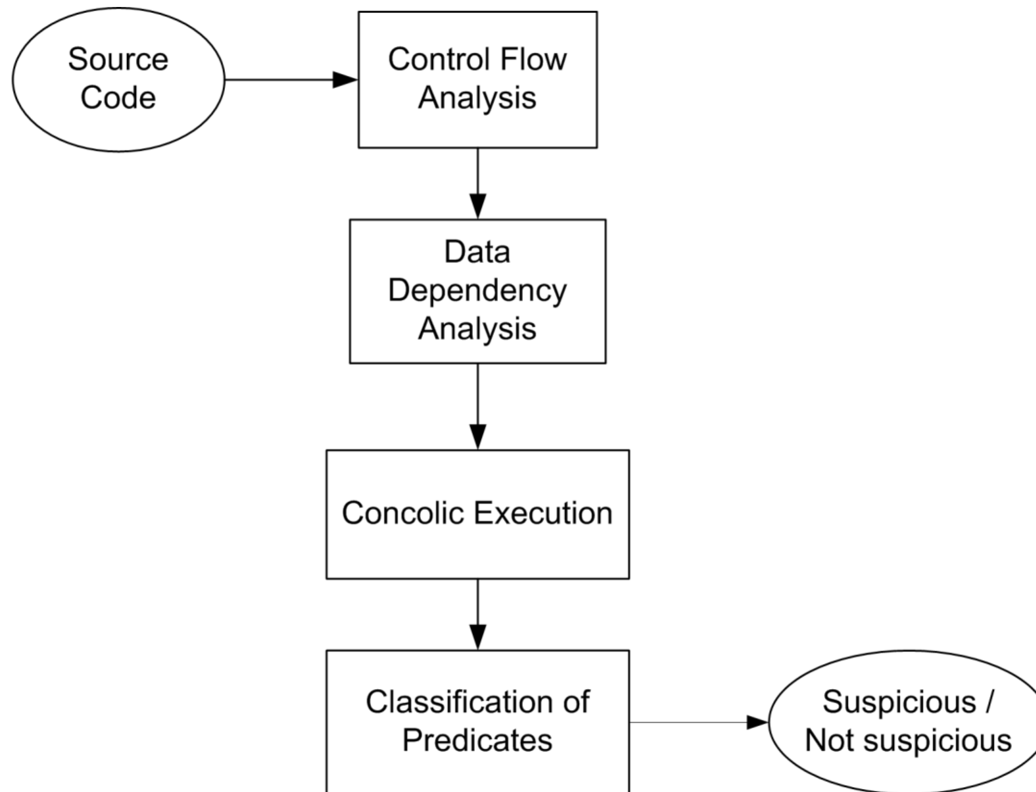
- Goal: categorize the conditions that affect the execution

- How can the conditions be uncovered? **Symbolic execution**
  - Resource-exhaustive → does not scale well
  - Source not available → does not work (e.g. library calls, kernel functions)
  - Unsolvable constraint → tools hang/abort

- Use **concolic execution**: mix concrete and symbolic executions
  - If code cannot be executed symbolically for whatever reasons, execute it as if no symbolic execution took place
  - Tools require the user to specify code for symbolic execution (typically variables) → what should be symbolic??

Overview

- Focus is on the source code
- Observation: hidden functionality is protected by branches

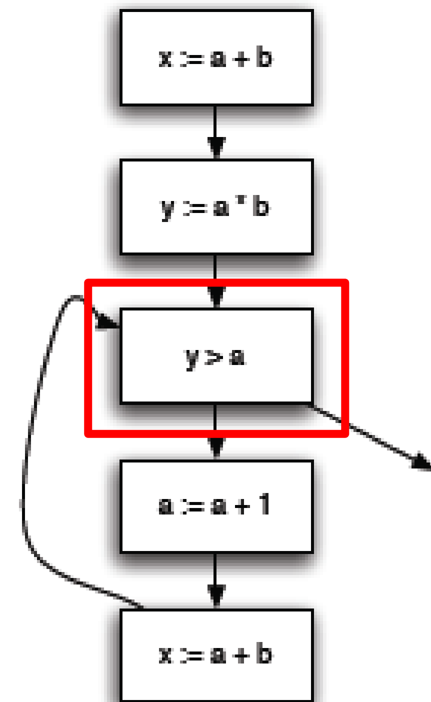


Overview – Control Flow Analysis

- Focus is on the source code
- Observation: hidden functionality is protected by branches

- Symbolic vars: `y`, `a`

```
x := a + b;
y := a * b;
while (y > a) {
    a := a + 1;
    x := a + b
}
```

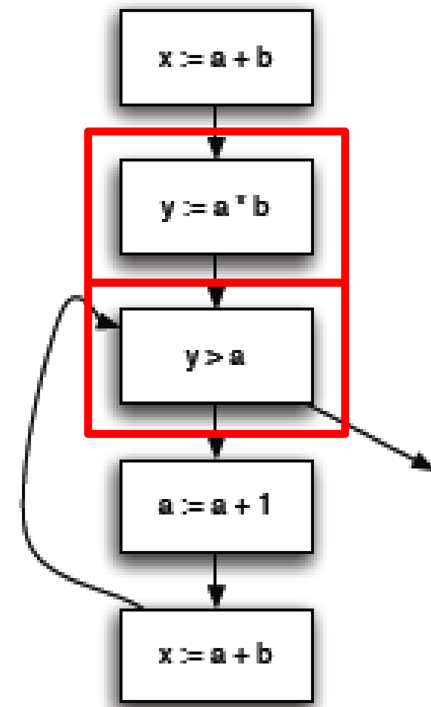


Overview – Data Dependency Analysis

- Focus is on the source code
- Observation: hidden functionality is protected by branches

- Symbolic vars: a

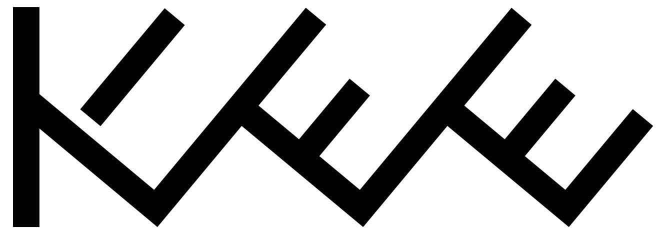
```
x := a + b;
y := a * b;
while (y > a) {
    a := a + 1;
    x := a + b
}
```



Overview – Concolic Execution

- Focus is on the source code

- Tool: KLEE LLVM Execution Engine
  - Symbolic virtual machine
  - Implements concolic execution
  - Built on top of the LLVM compiler infrastructure
  - Open source
  - Support for multiple solvers



Path Constraint Example KLEE

```
array arg0[5] : w32 -> w8 = symbolic
(query [
        (Eq 110 (Read w8 0 arg0))
        (Eq 111 (Read w8 1 arg0))
        (Eq 105 (Read w8 2 arg0))
        (Eq 0 (Read w8 3 arg0))
] false)
```

- arg0 = ”noi\0…”

Evaluation

- Example: stealthy portknocking written in C

- Make all variables symbolic → KLEE crashed after 3 hours
- Implement analysis steps
- Make interesting vars symbolic →

Evaluation

- Example: stealthy portknocking written in C

- Make all variables symbolic → KLEE crashed after 3 hours
- Implement analysis steps
- Make interesting vars symbolic → KLEE crashed after 11 hours

- Helped? Yes, increased runtime by ~300% ☺
- Worked? No ☹
- What went wrong?

Back to the Drawing Board

- Conditions are dependent on **external** events
  - Date and time are managed by the OS/firmware
  - Network packets originate potentially from the attacker
  - User name/password come from the user

- → Interesting variables depend on external data sources!
  - Portknocking: the packet should be executed symbolically

Back to the Drawing Board

- Conditions are dependent on **external** events
  - Date and time are managed by the OS/firmware
  - Network packets originate potentially from the attacker
  - User name/password come from the user

- → Interesting variables depend on external data sources!
  - Portknocking: the packet should be executed symbolically

- Result: KLEE reached the hidden functionality in ~2 mins

Future(/Current) Work

- Automatically make external data dependent variables symbolic
- Have a list of C functions that return data from external data sources
  - Should be easy to modify (e.g. add a function call)
- Apply instrumentation to many open source C programs
- Run KLEE on instrumented code → path constraints

Thank You!

ANY QUESTIONS?