ELECTRICAL *and* COMPUTER ENGINEERING DEPARTMENT

Robust Low-overhead Binary rewriter (RL-Bin)

Amir Majlesi-Kupaei, Kapil Anand, Khaled Elwazeer, Rajeev Barua

November 3 , 2017

Goal

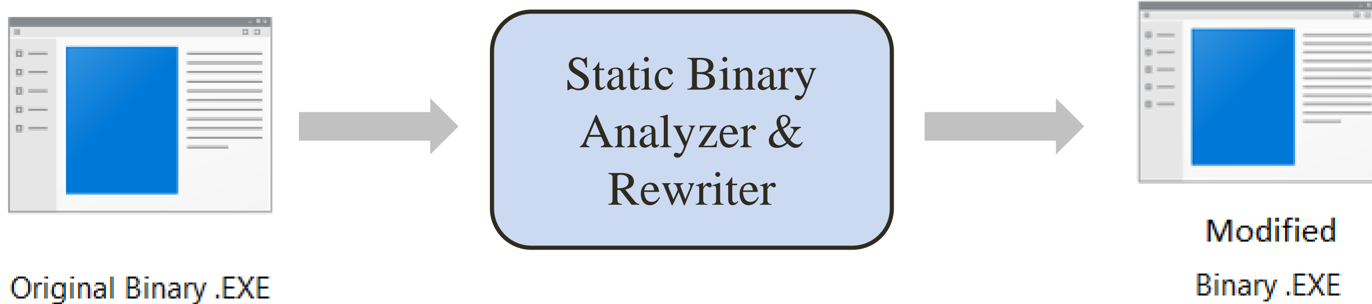
Develop a reliable, low-overhead binary rewriter that works for all benign programs.

- Must always work.
- Must be low overhead for use in deployment.
  - Conversations with industry: <5% overhead.

Goal

- Surprisingly, this does not exist today!
  - We can get reliable or low-overhead, but not both!

Static Binary Rewriters



- Existing Static Rewriters
  - Second Write (UMD team's previous work)
  - DynInst'06
  - ATOM
  - Diablo
  - Pebil

Static Binary Rewriters Limitations

- Does not work for obfuscated code
  - Disassembled code may actually be data
    - Fall through or destination of a conditional branch could be data
- Does not support:
  - Dynamically generated code
  - Self-modifying code
- The hash of the binary file changes ⇒ integrity check on file may fail.

Existing Dynamic Binary Rewriters

- Code Cache based Designs
  - DynamoRio: 20%
  - Intel's Pin: 54%
  - Valgrind: 330%
  - Diota: 20% ~ 400%
  - DynInst'11: 6% ~ 200 %
- In-place Designs
  - BIRD (hybrid static-dynamic): 5%

Dynamic Binary Rewriters Limitations

- Code cache based designs have unacceptable overhead due to
  - Code cache creation and management
  - Heavy address translation
- Existing in-place designs
  - Do not support
    - Obfuscation
    - Dynamically generated
    - Self modifying code

RL-Bin

- Our Robust Low-overhead Binary rewriter (RL-Bin)
  - Full code coverage
    - Supports dynamically generated code
    - Supports self modifying code
  - Support obfuscated code
    - Unconditional to conditional branch obfuscation
    - Exception based obfuscation
  - Still low overhead! (Currently 9%; work in progress)

Our method

Overall approach:

- Do not rely on static analysis
- Instead discover code dynamically as it executes
  - Conceptually discover every CTI's target as code when that CTI executes
- Two cases need runtime code validation
  - Conditional branches: because we cannot assume that both targets are code
  - Indirect CTIs: targets are discovered dynamically

Handling Conditional Branches

The problem:

- For obfuscated code, both conditional branch targets may not be code

Method:

- Insert hardware breakpoints at both CTI targets
- When invoked dynamically, code is set as discovered, and it can be rewritten
  - To reduce overhead, remove breakpoint after the first time it triggers

Handling Indirect CTIs

The problem:

- Indirect CTI targets are statically unknown ⇒ must be discovered dynamically.

The method:

- Insert instrumentation before each CTI using trampoline to register runtime target as code
  - Most common indirect CTIs are returns:
    - Can be optimized if we can verify that function is “safe” (We use JIT “static” analysis at runtime to try to prove that function will return to caller)

Software Infrastructure

- Developing a low overhead binary rewriting tool.
- We are using Capstone disassembler as part of the binary rewriter.
- Initial target platform
  - 32bit Windows applications

Thank you!

Prof. Rajeev Barua
Department of Electrical & Computer Engineering
University of Maryland, College Park.
Email: barua@umd.edu