A Framework for Robust Low-Overhead Binary Instrumentation

Amir Majlesi-Kupaei, Danny Kim, Kapil Anand, Aparna Kotha, Khaled ElWazeer, Rajeev Barua
The Problem: Instrumenting programs

Program instrumentation is invaluable for following capabilities:
Instrumenting interpreted vs. binary code

Programming Languages

- **Interpreted**
  (Python, Java, ...)
  - Relatively easy to instrument!

- **Compiled**
  (C, C++, Fortran, ...)
  - Instrumentation is very complicated!

Why binary code persists?

1. IP protection
2. High performance

Need a binary rewriter to instrument binary code!
Requirements for Deployment use of a binary rewriter

- Solution must be robust
  It should work for all binaries

- It must also be low-overhead
  High overhead is not tolerated in practice
What is a static rewriter?

Most commercial binaries are stripped, so they lack:

- Debug Info
- Relocation Info
- Symbol Table

Limitations:

- Error prone for all programs
- Do not support obfuscation
- No support for self-modifying code
- Dynamically-generated code not supported
- Binary file will change $\Rightarrow$ Checksum mismatch
Dynamic Solutions

In-place designs
(i.e. Dyninst’06 and BIRD)

- No support for obfuscation, self-modifying and dynamically generated code

Code-cache based designs
(i.e. Pin and DynamoRIO)

- They are robust but have high overhead!
Why Code-Cache has High Overhead?

- Indirect CTIs are everywhere
- They need to be translated
- Address translation cannot be removed!
Summary of existing solutions

Existing Dynamic Binary Rewriters
- Robust but high overhead!

Existing Static Binary Rewriters
- Low overhead but not robust!

Where we need to be.

Better Performance

Robustness

RL-Bin
Our Solution (RL-Bin)

RL-Bin

(Robust Low-overhead Binary Rewriter)

RL-Bin has very low overhead, (less than 5%)

- RL-Bin is robust, it supports
  - Obfuscation
  - Self-modifying code
  - Dynamically-generated code
RL-Bin’s Overall Approach

- Does not rely on static analysis
- Instead, it discovers code dynamically as it executes
- Conceptually, discover every CTI’s target as code when that CTI executes
When we arrive at a new code location, how far can we continue disassembling code?

There are four possibilities:

- **Straight-line code (non-CTI instructions)**
  - The address of the next instruction is known

- **Unconditional jumps**
  - The address of the target is known and fixed

- **Conditional branches**
  - Need run-time verification because we cannot assume that both targets are code

- **Indirect CTIs**
  - Must be verified during run-time because targets are discovered dynamically
Discovering new code: Conditional CTIs

The problem with conditional branches

For obfuscated code, both targets may not be code!

- Unoptimized solution for conditional CTIs
  - Instrumentation
  - RL-Bin
Discovering new code: Conditional CTIs

A lower overhead solution:

- Use Hardware breakpoints instead of instrumentation
- Much lower overhead!

- Optimized handling of conditional CTIs

Breakpoints can be removed if one or both of the targets are executed. Very low overhead!
Discovering new code: Indirect CTIs

- Unoptimized handling of Indirect CTIs

![Diagram]

- The instrumentation could be optimized and either reduced or removed!

- Indefinite number of run-time computer targets.

- Branch target prediction for common case target specialization: 113%

- Function cloning to eliminate returns: 49%

- Safe functions (i.e., those that cannot modify return address): 14%
Whitelisting Library Modules

- Optionally not monitor Win32 or standard library DLLs
- Only possible when call back functions and call back addresses are known

Optimizing Library Calls

- Library Calls done through Import Address Table.
- An indirect call with always one destination!
- Optimize away the check by write-protecting IAT
Handling Self-Modifying Code

To detect self-modifying code,

- Code segment is write-protected
- Any change to the code segment will trigger an exception
- Modified code will be disassembled and analyzed again

- The method can be optimized by adding instrumentation before and after instructions that cause self-modification
  (this is a best effort optimization to avoid triggering a lot of exceptions)
Handling Multi-Threaded Code

Handle race conditions from access to shared data structures, such as disassembly table, etc.

➢ Thread 1 is instrumenting a piece of code
➢ Thread 2 is executing the same code
➢ Thread 2 might execute from an address that is the middle of instrumentation instruction added by thread 1. The application will crash!

Handled by using mutually exclusive access to shared data structures
**Limitations of RL-Bin**

RL-Bin can handle any application that can be debugged by a debugger.

Benign applications are meant to be debugged, so they are supported.

### Troublesome feature

<table>
<thead>
<tr>
<th>Troublesome feature</th>
<th>Why debuggers would fail too</th>
</tr>
</thead>
<tbody>
<tr>
<td>Self-referencing code</td>
<td>• Detects changes in the code (which debuggers change)</td>
</tr>
<tr>
<td>Self-checksumming the memory image</td>
<td>• Debuggers use breakpoints and change memory checksum</td>
</tr>
<tr>
<td>Memory layout checking</td>
<td>• Requires no change in memory layout (which debuggers do)</td>
</tr>
</tbody>
</table>
Results (Spec 2017 Integer)

- Normalized run-time overhead of rewriters without added instrumentation

* Pin has higher overhead than DynamoRIO according to ref. C.-K. Luk et al. ACM, 2005
Results (Spec 2017 Floating Point)

- Normalized run-time overhead of rewriters without added instrumentation
Results with example heavyweight instrumentation

- Run-time overhead of rewriters with added instrumentations to count external calls
- The overhead is significant because all indirect calls (which are common) must be intercepted
Proving Robustness

- **Accuracy and Code Coverage**
  - Number of dynamically executed instructions were measured and compared against DynamoRIO.
  - Matched in all cases for SPECrate 2017 benchmarks

- **Commercial Applications were tested**
  - Obfuscation
  - Self-modifying code
  - Dynamically-generated code
Related Works

- Valgrind
- Pin
- DynamoRIO
- RL-Bin
- Diota
- Dyninst'11
- BIRD
- Atom
- Dyninst'O6
- SecondWrite
- Diablo
- Pebil

Support dynamically generated code:
- Valgrind
- Pin
- DynamoRIO
- RL-Bin
- Diota
- Dyninst'11
- BIRD
- Atom
- Dyninst'O6
- SecondWrite
- Diablo
- Pebil

Support exception based obfuscation:
- Valgrind
- Pin
- DynamoRIO
- RL-Bin
- Diota
- Dyninst'11
- BIRD
- Atom
- Dyninst'O6
- SecondWrite
- Diablo
- Pebil

Support self-modifying code:
- Valgrind
- Pin
- DynamoRIO
- RL-Bin
- Diota
- Dyninst'11
- BIRD
- Atom
- Dyninst'O6
- SecondWrite
- Diablo
- Pebil

Support conditional obfuscation:
- Valgrind
- Pin
- DynamoRIO
- RL-Bin
- Diota
- Dyninst'11
- BIRD
- Atom
- Dyninst'O6
- SecondWrite
- Diablo
- Pebil

No support for obfuscation and dynamic features:
- Valgrind
- Pin
- DynamoRIO
- RL-Bin
- Diota
- Dyninst'11
- BIRD
- Atom
- Dyninst'O6
- SecondWrite
- Diablo
- Pebil
Future Work

- Improve robustness by overcoming limitations
- Developing custom instrumentation API (User-friendly API for instrumentation)
- Plan to release RL-Bin publicly in late 2020
Thanks for your time!