BinRec: Dynamic Binary Lifting and Recompilation

Anil Altinay*, Joseph Nash*, Taddeus Kroes*
Prabhu Rajasekaran, Dixin Zhou,
Adrian Dabrowski, David Gens, Yeoul Na, Stijn Volckaert,
Cristiano Giuffrida, Herbert Bos, Michael Franz

*Equal Contribution Joint-First Authors
Legacy Binaries Need Help

- Source code or toolchain has been lost
- Microsoft patched CVE-2017-11882 in Equation Editor
- Binary Rewriting to patch, reoptimize, instrument, or harden binaries
Limitations of Static Rewriting

- 5 challenges for static binary rewriting
  - Code vs Data Separation
  - Indirect Control Flow Resolution
  - Ill-formed Code
  - Obfuscation
  - External Entry Points

- Static approaches use **heuristics** since they can’t solve these challenges in a principled way

- Produce rewritten binaries with **poor performance**, especially with instrumentation

- Require **re-implementing** well known analyses within every framework
BinRec vs McSema[6]

BinRec Binaries' Overhead

-0.5  0.0  0.5  1.0  1.5  2.0  2.5

-0.02 0.29

astar-00 bzip2-00 gobmk-00 h264ref-00 hhmmer-00 libquantum-00 mcf-00 omnetpp-00 perlbench-00 sjeng-00 xalan-00 astar-03 bzip2-03 gobmk-03 h264ref-03 hhmmer-03 libquantum-03 mcf-03 omnetpp-03 perlbench-03 sjeng-03 xalan-03 geomean-00 geomean-03
BinRec Framework

**Highlights**

- Lift binaries to LLVM IR
- Enable off-the-shelf compiler transformations
  - Safe Stack, ASAN, Optimizations, De-obfuscation, CFI
- Lift and run all C/C++ benchmarks in SPEC CINT 2006
- Better performing than existing lifting frameworks
  - Rev.ng[13] : 2.25x (static linked)
  - Multiverse[7] : 1.60x (w/o instrumentation)
  - McSema[6] : >2x (only 4 binaries)
  - BinRec : 1.29x
Leveraging Dynamic Traces to Overcome Static Rewriting Challenges
Code vs Data

- A statically unsolvable problem (Horspool and Marovac [3])

- Solution:
  - Copy of original program in case of inlined code and data as in prior work [10,11]
  - Dynamically observe the use of ambiguous values
  - Never accidentally disassemble data as code.

- libjpeg example [12]
void callback_func(j_common_ptr cinfo) {
    printf(".");
}

int main (int argc, char **argv) {
    struct jpeg_decompress_struct info; // jpeg_info
    struct jpeg_progress_mgr progress;

    ...  
    // After some initialization code
    progress.progress_monitor = callback_func;
    progress.pass_limit = 0x8048860;
    progress.pass_count = 0L;

    info.progress = &progress;
    jpeg_start_decompress( &info );

    char *data = (char *) malloc(dataSize);
    readData(info, data);
    ...
}
Code vs Data in libjpeg

```c
void callback_func(j_common_ptr cinfo) {
    printf(".");
}

int main (int argc, char **argv) {
    struct jpeg_decompress_struct info; // jpeg_info
    struct jpeg_progress_mgr progress;
    ...
    // After some initialization code
    progress.progress_monitor = callback_func;
    progress.pass_limit = 0x8048860;
    progress.pass_counter = 0L;
    info.progress = &progress;
    jpeg_start_decompress( &info );

    char *data = (char *) malloc(dataSize);
    readData(info, data);
    ...
}
```

McSema mis-handles this case!

Callback function is stored in a struct

Constant is same as address of callback function
Code vs Data in libjpeg

```c
void callback_func(j_common_ptr cinfo) {
    printf(".");
}

int main (int argc, char **argv) {
    struct jpeg_decompress_struct info; // JPEG_INFO
    struct jpeg_progress_mgr progress;
    ...
    // After some initialization code
    progress.progress_monitor = callback_func;
    progress.pass_limit = 0x8048860;
    progress.pass_counter = 0L;
    info.progress = &progress;
    jpeg_start_decompress( &info );
    char *data = (char *) malloc(dataSize);
    readData(info, data);
    ...
}
```

- Callback function is stored in a struct
- Constant is same as address of callback function
- McSema mis(handles this case!)
Indirect Control Flow

- Static approaches use heuristics with value set analysis
- BinRec records the exact target addresses of each indirect control flow

```
ret
%pc = load i32, i32* @PC
switch %pc, label %otherwise
  [ i32 &A, label %BasicBlock_A
    i32 &B, label %BasicBlock_B
  ]
Traces observed:
ret to A
ret to B
```
External Entry Points: Callbacks

Binary Code

```c
int compare( const void* a, const void* b ) {
    ....
    ....
}

int main() {
    int arr[] = {5, 3, 1, -1};
    int size = sizeof arr / sizeof *arr;
    qsort( arr, size, sizeof( int ),
          compare);
}
```

Library Code

```c
void qsort(void *base,
           size_t nel,
           size_t width,
           int (*compar)(const void *,
                          const void *))
{
    ....
    ....
    ....
    compare(arg1, arg2);
}
```

- **Callback function**: Passed to `qsort` function
- **Passed to qsort function**: Callback function
- **qsort invokes callback function**: Callback function is called by `qsort`
Support for External Entry Points

Problem: The callback function pointer still points to the original callback function

Library Code

```c
void qsort(void *base, size_t nel, size_t width, int (*compar)(const void *, const void *))
{
    ....
    ....
    ....
    compare(arg1, arg2);
}
```

Recovered Code

```c
int compare_recovered(....) {
    ....
}
int main_recovered() {
    ....
    qsort(...., compare);
}
```
Support for External Entry Points

Problem: The callback function pointer still points to the original callback function

Recovered Code

```c
int compare_recovered( .... ) {
    ....
}

int main_recovered() {
    ....
    qsort( ...., compare);
}
```

Library Code

```c
void qsort(void *base,
          size_t nel,
          size_t width,
          int (*compar)(const void *,
                        const void *))
{
    ....
    ....
    ....
    compare(arg1, arg2);
}
```

1. qsort invokes original callback function
Support for External Entry Points

- Option 1: statically link library code into the analysis region
  - Problem: High memory usage

- Option 2: update code pointers
  - Problem: Heuristics fail

- Option 3: create a lookup table
  - Problem: Performance degradation
Support for External Entry Points

Our Dynamic Approach

Original Code Region

compare: jmp compare_recovered

Library Code

void qsort(void *base, size_t nel, size_t width, int (*compar)(const void *, const void *))
{
    ....
    ....
    compare(arg1, arg2);
}

Recovered Code

int compare_recovered(...)
{
    ....
}
int main_recovered()
{
    ....
    qsort(...., compare);
}

No need for arguments patching!

1. Use original address space as trampolines
2. No need for arguments patching!
3. original address space as trampolines
4. original address space as trampolines
Coverage for Dynamic Analysis

Dynamic lifting engine efficiently covers paths of interest

Installed handlers provides recovery and iterative improvement
BinRec Architected for Coverage

- Coverage for Dynamic Analysis
- Dynamic lifting engine efficiently covers paths of interest
- Installed handlers provides recovery and iterative improvement
BinRec Architected for Coverage

- Coverage for Dynamic Analysis
- Dynamic lifting engine efficiently covers paths of interest
- Installed handlers provides recovery and iterative improvement
Multi-Trace Merging

- **Drive execution** - Trusted inputs, fuzzing, concolic execution
- **Build CFG** – Merge basic block boundaries, control flow edges
Path Miss := instructions needed for the current workload were not observed in the initial lifting

Path Miss Handlers are installed in every control flow transfer
  - Optimized Out
  - Report and Log
  - Fallback
  - Incremental Lifting
Path Miss Handler: Incremental Lifting

- Use logged ‘path misses’ as points to restart lifting
Incremental Lifting of Bzip2
Correct and Performant Rewriting of SPEC CINT 2006

BinRec Binaries' Overhead
## BinRec vs Static Rewriters

<table>
<thead>
<tr>
<th>SPEC Int Geomean</th>
<th>O3</th>
</tr>
</thead>
<tbody>
<tr>
<td>BinRec</td>
<td>1.29x</td>
</tr>
<tr>
<td>Multiverse [7]</td>
<td>1.60x</td>
</tr>
<tr>
<td>Rev.ng[13]</td>
<td>2.25x</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>O0</th>
<th>mcf</th>
<th>bzip2</th>
<th>sjeng</th>
<th>libquantum</th>
</tr>
</thead>
<tbody>
<tr>
<td>BinRec</td>
<td>0.83x</td>
<td>0.76x</td>
<td>0.77x</td>
<td>0.95x</td>
</tr>
<tr>
<td>McSema</td>
<td>2.31x</td>
<td>2.84x</td>
<td>3.43x</td>
<td>2.07x</td>
</tr>
</tbody>
</table>

- Static approaches are less precise
  - More possible behaviors -> less optimization is possible

- Dynamic lifting has a one-time cost (~450x on SPEC)

<table>
<thead>
<tr>
<th>SPEC Int Geomean</th>
<th>O0</th>
<th>O3</th>
</tr>
</thead>
<tbody>
<tr>
<td>BinRec</td>
<td>178480s</td>
<td>138379s</td>
</tr>
<tr>
<td>McSema</td>
<td>371s</td>
<td>320s</td>
</tr>
</tbody>
</table>
Now we can have nice things!

LLVM IR + dynamic linking support ==

No need to rewrite transformations
Address Sanitizer in BinRec

- ASAN: A memory access violation finding tool available in LLVM
- Works with off the shelf ASAN no modifications on binaries
- All memory accesses are instrumented
- Heap allocations are instrumented
- No stack variable symbolization -> stack allocations are not instrumented by ASAN
- ASAN runtime library links and reports violations
Obfuscation and Ill-formed Code

Unaligned / Overlapping Instructions

Virtualization

Packing

Code Encryption

Fig. 1: Simplified example cases of dereferencing a dangling pointer for Use-After-Free under pointer-spray attack (assuming 0xdeadbeefcafebabe is the crafted pointer). The box in the middle represents dangling pointed object and each row indicates pointer-type member variable. Assume there are five possible dangling pointers due to randomization. For better visualization, the memory dump is shown in big-endian format.

and Data Execution Prevention (DEP) as the previous study does. Finally, we expect that our attacker feeds untrusted input (e.g., PDF document, JavaScript, Network Stream) to the corresponding application parser that has heap vulnerabilities.

B. Successful Triggering of Heap Vulnerabilities

Any triggering step of heap vulnerabilities that occurs due to out-of-bounds access are affected by byte-granularity heap randomization. For example, the first use of dangling-pointer in use-after-free guarantees to crash any application with 87.5% (75% in 32-bit) probability as there are eight (four in 32-bit) possible outcomes of the misinterpreted pointer alignment.

Consider the exploitation steps of use-after-free: (i) an object is freed and a dangling pointer is created, (ii) the attacker places a crafted object around the dangling-pointed memory region, and (iii) the program uses the dangling pointer as if the original object member variables (pointer member variables) are still intact thus using attacker's crafted pointer. These steps imply that there are two independent heap chunk allocations around the dangling-pointed heap area. Although the address of each heap chunks is random, if the allocation granularity is bigger than the pointer-width, an attacker can spray the heap and overlap the fake object and dangling-pointer thus successfully trigger the use-after-free without pinpointing the exact memory addresses.

This effectiveness can be described by depicting a simplified example. Figure 1 depicts an example case of dereferencing a dangling pointer (to access a pointer member variable) after attacker launches a pointer-spray attack. For simplicity, let's assume attacker wants to hijack a pointer into 0xdeadbeefcafebabe and there are five unpredictable cases of dangling pointers which will be randomly decided at runtime.

In Figure 1a, an attacker can hijack the target pointer member variable with a very high chance because the heap randomization follows word-granularity. The attacker can spray the eight-byte sequence "DE AD BE EF CA FE BA BE" sufficiently long to defragment the heap region and bypass the randomization. However in Figure 1b, the randomization is byte-granularity thus the attack fails with 87.5% probability regardless of the spray; unless the pointer is composed with same bytes (we discuss this issue at the end of this section).

The effectiveness of byte-granularity heap randomization is not specific to particular heap vulnerabilities. We emphasize that any exploitation step which involves the use of crafted pointer upon out-of-bounds heap access is affected. For example, exploitation of heap overflow, uninitialized heap access vulnerability also involves out-of-bounds heap access [8], [11] thus affected by byte-granularity heap randomization.

So far, the security effectiveness of byte-granularity heap randomization seems small, as one out of eight (or four) triggering attempts will succeed. However, this probability of single dereferencing is not the probability of a successful attack. Modern heap exploitation usually involves multiple combination and repetition of such bug triggering. According to Google Project-Zero, successful exploitation of CVE-2015-3077 required up to 31 times of pointer confusion. As heap exploitation involves multiple uses of crafted pointers, the defense probability will increase exponentially. However exact
Control-Flow Integrity in BinRec

- Only observed control flows are allowed
  - C -> G disallowed

- Contexts are merged
  - Performance Vs Precision

- Indirect CFT -> Direct CFT
  - Ret = switch %pc, label %error
    - [ i32 &D, label %BB_D ]

- BinCFI uses an address taken heuristic over-approximation
  - BinRec is on average at least 25x more restrictive than BinCFI
BinRec: Dynamic Binary Lifting and Recompilation

- First of its kind dynamic trace lifting and recompilation of stripped binaries
- Heuristic free and supports obfuscated code
- Enables off-the-shelf transformations, which only existed for source code
- Low overhead (29%)
Thanks and Acknowledgements

- We thank our shepherd and the anonymous reviewers for their feedback.
- Thanks to Alyssa Milburn for editing assistance, and Chinmay Deshpande for testing and ongoing efforts.
- This material is based upon work partially supported by the Defense Advanced Research Projects Agency (DARPA) under contracts FA8750-15-C-0124 and FA8750-15-C-0085, by the United States Office of Naval Research (ONR) under contract N00014-17-1-2782, by the National Science Foundation under awards CNS-1619211 and CNS-1513837. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the Defense Advanced Research Projects Agency (DARPA) or its Contracting Agents, the Office of Naval Research or its Contracting Agents, the National Science Foundation, or any other agency of the U.S. Government.
1. "Computer History Museum - 108" by phrenologist is licensed under CC BY-NC 2.0


5. B. Dolan-Gavitt, T. Leek, J. Hodosh, W. Lee. Tappan Zee (North) Bridge: Mining Memory Accesses for Introspection. 20th ACM Conference on Computer and Communications Security (CCS), Berlin, Germany, November 2013


