Reassembleable Disassembling

Shuai Wang, Pei Wang, Dinghao Wu

*Pennsylvania State University*

24th USENIX Security Symposium, August 2015
Motivation

Analysing and *retrofitting* COTS binaries with...

- software fault isolation
- control-flow integrity
- symbolic taint analysis
- elimination of ROP gadgets
Motivation

Analysing and *retrofitting* COTS binaries with . . .
- software fault isolation
- control-flow integrity
- symbolic taint analysis
- elimination of ROP gadgets

Binary rewriting comes with major drawbacks/limitations
- runtime overhead from patching due to control-flow transfers
- patching requires PIC if code is relocated
- instrumentation significantly increases binary size
- binary reuse only works for small binaries (coverage)
Produce *reassembleable* assembly code from *stripped COTS binaries* in a *fully automated* manner.

- Allows binary-based whole program transformations
- Requires relocatable assembly code $\rightarrow$ symbolization of immediate values
- Complementary to existing work
Symbolization

Given an immediate value in assembly code, is it a constant or a memory address?

- Reassembling transformed program changes binary layout
- Address changes invalidate memory references
- x86
  - No distinction between code and data
  - Variable-length instruction encoding
(Un)Relocatable Assembly Code

binary

\text{mov 0xc0, \%eax}

0xc0: \begin{array}{c} 0xa08 \end{array}

unrelocatable

\text{.text \mov 0xc0, \%eax}

\text{.data \long 0xa08}

\text{assemble}

0xc0: \begin{array}{c} \? \end{array}

relocatable

\text{.text \mov Glob, \%eax}

\text{.data \Glob: \long 0xa08}

\text{assemble}

Glob: \begin{array}{c} 0xa08 \end{array}
Types of Symbol References

Code Section

fun1:
call fun2
fun2:
mov ptr, %eax
lea (%eax, %ebx, 4), ... binaries,
making assumptions on user requirements for our tool
also helps improve its performance. As stated earlier,

handler1:
... 
handler2:
...

Data Section

ptr:
.long table
table:
.long handler1
.long handler2

c2c
c2d
d2c
d2d
Symbolization of c2c and c2d References

- Valid memory references point into code or data section
- Assume all immediates to be references and filter out invalid ones
Symbolization of d2c and d2d References

Assumption 1

“All symbol references stored in data sections are n-byte aligned, where n is 4 for 32-bit binaries and 8 for 64-bit binaries.”

→ Consider only n-byte values which are n-byte aligned
Symbolization of d2c and d2d References

Assumption 1

“All symbol references stored in data sections are n-byte aligned, where n is 4 for 32-bit binaries and 8 for 64-bit binaries.”

→ Consider only n-byte values which are n-byte aligned

Assumption 2

“Users do not need to perform transformation on the original binary data.”

→ Keep start addresses of data sections during reassembly and ignore d2d references
Symbolization of d2c and d2d References

Assumption 1

“All symbol references stored in data sections are n-byte aligned, where n is 4 for 32-bit binaries and 8 for 64-bit binaries.”

→ Consider only n-byte values which are n-byte aligned

Assumption 2

“Users do not need to perform transformation on the original binary data.”

→ Keep start addresses of data sections during reassembly and ignore d2d references

Assumption 3

“d2c symbol references are only used as function pointers or jump table entries.”

→ References need to point to start of a function or form a jump table
Evaluation

- **UROBOROS**: 13,209 SLOC in OCaml and Python; works with x86/x64 ELF binaries
- Intel Core i7-3770 @ 3.4GHz with 8GiB RAM running Ubuntu 12.04
- 122 programs compiled for 32- and 64-bit targets
- gcc 4.6.3 with default configuration and optimization level of each program
- stripped before testing

<table>
<thead>
<tr>
<th>Collection</th>
<th>Size</th>
<th>Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>COREUTILS</td>
<td>103</td>
<td>GNU Core Utilities</td>
</tr>
<tr>
<td>REAL</td>
<td>7</td>
<td>bc, ctags, gzip, mongoose, nweb, oftpd, thttpd</td>
</tr>
<tr>
<td>SPEC</td>
<td>12</td>
<td>C programs in SPEC2006</td>
</tr>
</tbody>
</table>
Architecture of UROBOROS

Binary

Disassembly Module

Linear Disassembler

Disassembly Validator

Data

Meta-Data

Code

Analysis Module

Symbol Lifting

Control-Flow Structure Recovery

Relocatable Assembly

External Analyses & Transformations

Figure 3: The architecture of UROBOROS

Finally, we emphasize that the assembly code generated and transformed by UROBOROS can be directly assembled back as a working binary by normal assemblers. In particular, the binary output is indeed a normal executable file without any abnormal characteristics such as patched or duplicated sections. Therefore, the reassembled binary can be disassembled again by UROBOROS or be processed by other reverse engineering tools.

We have implemented a prototype of UROBOROS in OCaml and Python, with a total of 13,209 lines of code. Our prototype works for both x86 and x64 ELF binaries.

5.2 Disassembly

In our prototypical implementation, the linear disassembler employed by UROBOROS’s disassembly module is objdump from GNU Binutils. We implement an interactive disassembly process originally proposed in BinCFI [50]. In this process, the disassembler communicates with a validator which corrects disassembly errors due to “data gaps” between adjacent code blocks. The interactive procedure is as follows:

• objdump tries to decode the input binary for the first time.
• The validator examines the output and check if there are explicit errors reported by objdump. In case there are no errors, the raw disassembly process terminates. Otherwise, the validator assumes the errors are caused by data embedded in code and computes the upper and lower bounds of identified “data gaps”.
• With the computed range of identified “gaps”, the validator guides objdump to decode the binary again, with those “gaps” skipped.
• Repeat this decode-validate process until no error occurs or the running time of the whole process reaches a time limit specified by users.

We leverage three rules proposed in BinCFI to validate the disassembly results and locate the data “gaps”, i.e., “invalid opcode”, “direct control transfers outside the current module”, and “direct control transfer to the middle of an instruction”. Since identifying bounds of each data gap can rely on the control-flow information of decoded instructions, the validator occasionally leverages UROBOROS’s analysis module to retrieve the control-flow information.

5.3 Support for Program Transformation

UROBOROS provides basic support for program-wide transformations by partially recovering control-flow structures of the decoded instructions. We collect all the control transfer instructions to divide each function into multiple basic blocks. Control-flow graphs are rebuilt on top of these basic blocks. As a prototype, UROBOROS currently only processes direct control transfers. Regarding the intractable indirect transfers, a potential solution is to use value set analysis (VSA) [5] for destination computation. We leave including indirect control transfers in the CFG as future work.

5.4 Meta-Information Recovery

UROBOROS recovers the program-linkage table (PLT) and the export table in ELF binaries. The PLT table supports dynamic linkage by redirecting intra-module transfers on its stubs to external functions. As the base address of the PLT table can change after reassembling, we translate the memory references to PLT stubs to their corresponding external function names, and let the linker to rebuild the PLT table with correct memory references during link time. In particular, this table is dumped out from the input binary and parsed into multiple entries, each containing the memory address of a PLT stub with...
Figure 3: The architecture of UROBOROS

Architecture of UROBOROS

- Binary
  - Linear Disassembler
    - Disassembly Module
      - Disassembly Validator
      - Data
      - Code
  - Meta-Data
- Analysis Module
  - Symbol Lifting
  - Control-Flow Structure Recovery
- Relocatable Assembly
  - External Analyses & Transformations

Finally, we emphasize that the assembly code generated and transformed by UROBOROS can be directly assembled back as a working binary by normal assemblers. In particular, the binary output is indeed a normal executable file without any abnormal characteristics such as patched or duplicated sections. Therefore, the reassembled binary can be disassembled again by UROBOROS or be processed by other reverse engineering tools.

We have implemented a prototype of UROBOROS in OCaml and Python, with a total of 13,209 lines of code. Our prototype works for both x86 and x64 ELF binaries.

5.2 Disassembly

In our prototypical implementation, the linear disassembler employed by UROBOROS’s disassembly module is `objdump` from GNU Binutils. We implement an interactive disassembly process originally proposed in BinCFI [50]. In this process, the disassembler communicates with a validator which corrects disassembly errors due to “data gaps” between adjacent code blocks. The interactive procedure is as follows:

1. `objdump` tries to decode the input binary for the first time.
2. The validator examines the output and checks if there are explicit errors reported by `objdump`. In case there are no errors, the raw disassembly process terminates. Otherwise, the validator assumes the errors are caused by data embedded in code and computes the upper and lower bounds of identified “data gaps.”
3. With the computed range of identified “gaps,” the validator guides `objdump` to decode the binary again, with those “gaps” skipped.
4. Repeat this decode-validate process until no error occurs or the running time of the whole process reaches a time limit specified by users.

We leverage three rules proposed in BinCFI to validate the disassembly results and locate the data “gaps”, i.e., “invalid opcode”, “direct control transfers outside the current module”, and “direct control transfer to the middle of an instruction”. Since identifying bounds of each data gap can rely on the control-flow information of decoded instructions, the validator occasionally leverages UROBOROS’s analysis module to retrieve the control-flow information.

5.3 Support for Program Transformation

UROBOROS provides basic support for program-wide transformations by partially recovering control-flow structures of the decoded instructions. We collect all the control transfer instructions to divide each function into multiple basic blocks. Control-flow graphs are rebuilt on top of these basic blocks. As a prototype, UROBOROS currently only processes direct control transfers. Regarding the intractable indirect transfers, a potential solution is to use value set analysis (VSA) [5] for destination computation. We leave including indirect control transfers in the CFG as future work.

5.4 Meta-Information Recovery

UROBOROS recovers the program-linkage table (PLT) and the export table in ELF binaries. The PLT table supports dynamic linkage by redirecting intra-module transfers on its stubs to external functions. As the base address of the PLT table can change after reassembling, we translate the memory references to PLT stubs to their corresponding external function names, and let the linker to rebuild the PLT table with correct memory references during link time. In particular, this table is dumped out from the input binary and parsed into multiple entries, each containing the memory address of a PLT stub with...
Test input shipped with programs or custom test of major functionality (some of REAL)

<table>
<thead>
<tr>
<th>Assumption Set</th>
<th>Binaries Failing Functionality Tests 32-bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>{}</td>
<td>h264ref, gcc, gobmk, hmer</td>
</tr>
<tr>
<td>{A1}</td>
<td>h264ref, gcc, gobmk</td>
</tr>
<tr>
<td>{A1, A2}</td>
<td>h264ref, gcc, gobmk</td>
</tr>
<tr>
<td>{A1, A3}</td>
<td>gobmk</td>
</tr>
<tr>
<td>{A1, A2, A3}</td>
<td>gobmk</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Binaries Failing Functionality Tests 64-bit</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>perlbench, gcc, gobmk, hmmer, sjeng, h264ref, lmb, sphinx3</td>
</tr>
<tr>
<td></td>
<td>perlbench, gcc, gobmk</td>
</tr>
<tr>
<td></td>
<td>perlbench, gcc, gobmk</td>
</tr>
<tr>
<td></td>
<td>gcc, gobmk</td>
</tr>
</tbody>
</table>
Table 3: Dynamic test results on reassembled binaries

<table>
<thead>
<tr>
<th>Assumption</th>
<th>SPEC</th>
<th>REAL</th>
<th>COREUTILS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time for</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 8 presents processing time for COREUTILS binaries selected using a same algorithm.

Table 4: Symbolization false positives of 32-bit SPEC, REAL and COREUTILS (Others have zero false positive)

<table>
<thead>
<tr>
<th>Benchmark</th>
<th># of Ref.</th>
<th>}</th>
<th>{A1}</th>
<th>{A1, A2}</th>
<th>{A1, A3}</th>
<th>{A1, A2, A3}</th>
</tr>
</thead>
<tbody>
<tr>
<td>perlbench</td>
<td>76538</td>
<td>2</td>
<td>0.026%</td>
<td>0</td>
<td>0.000%</td>
<td>0</td>
</tr>
<tr>
<td>hmmer</td>
<td>13127</td>
<td>12</td>
<td>0.914%</td>
<td>0</td>
<td>0.000%</td>
<td>0</td>
</tr>
<tr>
<td>h264ref</td>
<td>20600</td>
<td>27</td>
<td>1.311%</td>
<td>1</td>
<td>0.049%</td>
<td>0</td>
</tr>
<tr>
<td>gcc</td>
<td>262698</td>
<td>49</td>
<td>0.187%</td>
<td>32</td>
<td>0.122%</td>
<td>0</td>
</tr>
<tr>
<td>gobmk</td>
<td>65244</td>
<td>1348</td>
<td>20.661%</td>
<td>985</td>
<td>15.097%</td>
<td>78</td>
</tr>
</tbody>
</table>

Table 5: Symbolization false negatives of 32-bit SPEC, REAL and COREUTILS (Others have zero false negative)

<table>
<thead>
<tr>
<th>Benchmark</th>
<th># of Ref.</th>
<th>}</th>
<th>{A1}</th>
<th>{A1, A2}</th>
<th>{A1, A3}</th>
<th>{A1, A2, A3}</th>
</tr>
</thead>
<tbody>
<tr>
<td>perlbench</td>
<td>76538</td>
<td>2</td>
<td>0.026%</td>
<td>0</td>
<td>0.000%</td>
<td>0</td>
</tr>
<tr>
<td>hmmer</td>
<td>13127</td>
<td>12</td>
<td>0.914%</td>
<td>0</td>
<td>0.000%</td>
<td>0</td>
</tr>
<tr>
<td>h264ref</td>
<td>20600</td>
<td>27</td>
<td>1.311%</td>
<td>0</td>
<td>0.000%</td>
<td>0</td>
</tr>
<tr>
<td>gcc</td>
<td>262698</td>
<td>11</td>
<td>0.042%</td>
<td>0</td>
<td>0.000%</td>
<td>0</td>
</tr>
<tr>
<td>gobmk</td>
<td>65244</td>
<td>86</td>
<td>1.318%</td>
<td>0</td>
<td>0.000%</td>
<td>0</td>
</tr>
</tbody>
</table>

Some programs in COREUTILS are not suitable for performance benchmarking, including "su", "ssh", etc. After excluding these programs, we have 90 left to inspect in COREUTILS. The experiments revealed that ROBOROS has almost zero impact on binary size when delivering reassemblability. As aforementioned in §5.6, subsequent expansion for COREUTILS is 0.83%, 0.00% for SPEC and 0.57 seconds on binaries from REAL. Data shows that ROBOROS makes it a tool totally practical as far as we know. To roughly investigate ROBOROS's compatibility with other compilers, we try to disassemble and reassemble some binaries compiled by Clang, gcc, gobmk, etc.
No increase in binary size after first disassemble-assemble cycle
Heuristic-based symbolization of memory references

**UROBOROS**\(^1\) provides reassembleable disassembly

Assumes availability of raw disassembly and function starting addresses

Tested with gcc and Clang compiled binaries

Limited support for C++ (need to parse DWARF)

\(^1\)Available at https://github.com/s3team/uroboros