CSci 5980/8980
Manual and Automated Binary Reverse Engineering
Slides 6: Binary RE for rewriting

Stephen McCamant
University of Minnesota

## Application: binary rewriting

- Key use for automated binary RE: enable modification
- Instrumentation, optimization, security hardening
- Needs limited capabilities, but highly accurate

## Static vs. dynamic rewriting

- Static rewriting: transform complete binary before execution
- Dynamic rewriting: rewrite parts of code as they are executed
- Dynamic rewriting needs less analysis, but more infrastructure
- Reverse engineering is more closely related to static rewriting

## Disassembly: the easy problem

- Given a starting location, what instruction is it?
  - I.e., HW1 question 1
- Conceptually, not too hard
  - Like a big lookup table
- Practically still not easy to perfectly cover a large ISA

## Disassembly: the hard problem

- Finding what locations in a binary are code to execute
- Includes distinguishing code from data
- Also harder on ISAs with variable-length/unaligned instructions
- Solving precisely is undecidable

## Linear sweep disassembly

- Start at beginning of code, assume every instruction is followed directly by another
  - Computing instruction length is "easy"
- Classic implementation: `objdump`
- Works well for well-behaved binaries (esp. GCC on x86)
  - Strict separation between code and data
  - With aligned instructions, easy to get a superset

## Recursive disassembly

- Statically follow control flow from an entry point
  - Explore both sides of branches
  - Explore callee and after a function call
- False positives from no-return functions
- False negatives from indirect jumps

## Superset disassembly

- Ultimate approach to deal with variable-length instructions:
  - Disassemble starting at every byte offset
- \+ No missed instructions
- − Guaranteed to also have lots of junk

## Indirect jump analysis

- Key gap in recursive disassembly: code reachable (only) via indirect jumps
- Precise analysis is still undecidable
- But partial or over/under-approximations are possible

## Jump tables

- One easier case of indirect jumps: `switch` statements
- Jump target comes from a nearby lookup table
- But must determine structure and bounds of the table

## VTables

- Implementation of C++ virtual method dispatch
- Table of method implementations, layout specified by C++ ABI
- Enumerating targets simpler than recovering inheritance structure

## Function pointers

- Hardest case of indirect jumps in C code
- Use can be unstructured
  - E.g., stored in complex heap data structures
- For disassembly, just need all targets, not jump to targets map
- Approximation: find all code-address-like values used to initialize

## Function boundaries

- Nice to have but not truly needed for rewriting
- Many good approximations, hard to get perfectly
- Depending on the compiler, may not be a unique right answer

## Symbols vs. stripped

- Many binary rewriting tools fudge by requiring some symbol information
- Enables some but not all use cases
- Worst case: require special compiler to save information no real compiler saves
- Best case: metadata already needed for ASLR or PIE

## Reassembleable disassembly

- Better but even harder: recover assembly code that uses labels
- Enables rewriting just by changing output and re-assembling
- A 2015 paper pointed out this was desirable but not available
- My opinion: still no fully satisfactory implementation

## Label symbolization

- Identify which occurrences of the bit pattern `0x4011e2` are meant to point to the function that currently lives there
- Also undecidable, but FPs and FNs are both hard to live with
- Fairly simple heuristics get tantalizing close
- Expensive analysis like symbolic execution can also help

## Type analysis

- Definitely also reverse engineering, but IMO not so closely related to rewriting
- Better the subject of its own survey
- We'll also come back to this later