ILLINOIS
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

allvm - Binary Decompilation

Sandeep Dasgupta
University of Illinois Urbana Champaign

March 29, 2016

Outline

Goal & Motivation

Possible Approaches

Our Approach

Decompile `Machine Code` $\longrightarrow$ `LLVM IR`

mcsema

Demo

Backup

Research Goal

- Research Goal
  - Obtain “richer” LLVM IR than native machine code.
  - Enable advanced compiler techniques (e.g. pointer analysis, information flow tracking, automatic vectorization)

Why “richer” LLVM IR

- Source code analysis not possible
  - IP-protected software
  - Malicious executables
  - Legacy executables
- Source code analysis not sufficient
  - What-you-see-is-not-what-you-execute
- End-user security enforcement
- Platform aware optimizations

Outline

Goal & Motivation

Possible Approaches

Our Approach

Decompile `Machine Code` $\longrightarrow$ `LLVM IR`

mcsema

Demo

Backup

The 3 Possible Approaches

- Decompile `Machine Code` $\longrightarrow$ `LLVM IR`
  - Easy to adopt
  - No compiler support needed
- "Annotated" `Machine Code` $\longrightarrow$ `LLVM IR`
  - Effective reconstruction of higher level IR
  - Minimal compiler support needed
- Ship `LLVM IR`
  - Benefit: *No loss* of information via conversion to and from binary code.

Decompile `Machine Code` $\longrightarrow$ `LLVM IR`

- Challenge: Quality
  - Reconstructing code and control flow - much researched.
  - Variable recovery
  - Function & ABI rules recovery

“Annotated” `Machine Code` → `LLVM IR`

- Challenge:
  - Annotations must be “minimal” & sufficient
  - Annotations must be compiler and IR-independent
  - Adoption

Ship `LLVM IR`

- Challenge:
  - Adoption in Non LLVM based compilers
  - Stable distribution format for shipping
  - Risks to intellectual property
  - Code size bloat

Outline

Our Approach

- Long term goal
  - Minimal compiler-independent annotations to reconstruct high-quality IR
- Short term goals
  1. Experiment with `Machine Code` $\rightarrow$ `LLVM IR`, to **understand** the challenges better
     - To select an existing decompilation framework.
     - Experiment with different variable and type recovery strategies
  2. Design suitable annotations for what cannot be inferred without them

Outline

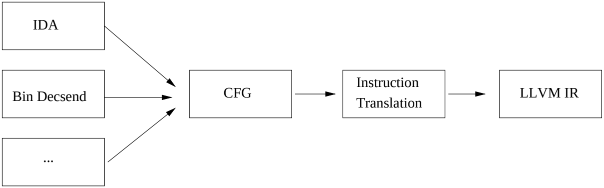
Variable & Function Parameter Recovery

- Benefit
  - Enables many fundamental analysis (Dependence, Pointer analysis)
  - Functional IR
- State of the art
  - Divine
    - State of the art variable recovery
  - Second Write
    - Heuristics for function parameter detection
    - Scalable variable and type recovery
  - TIE
    - Type recovery

Outline

Choosing mcsema

- Functional `LLVM IR`
- Separation of modules: CFG recovery and CFG $\longrightarrow$ `LLVM IR`
- Actively supported and open sourced



Support & Limitations

- What Works
  - Integer Instructions
  - FPU and SSE registers
  - Callbacks, External Call, Jump tables
- In Progress
  - FPU and SSE Instructions: Not fully supported
  - Exceptions
  - Better Optimizations

Outline

mcsema: Demo

Outline

What-you-see-is-not-what-you-execute

The following compiler (Microsoft C++ .NET) induced vulnerability was discovered during the Windows security push in 2002