allvm - Binary Decompilation

Sandeep Dasgupta
University of Illinois Urbana Champaign

March 30, 2016

Outline

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 executable
  - Legacy executable
- Source code analysis not sufficient
  - What-you-see-is-not-what-you-execute
- Platform aware optimizations

Outline

The 3 Possible Approaches

- Decompile `Machine Code` $\longrightarrow$ `LLVM IR`
- "Annotated" `Machine Code` $\longrightarrow$ `LLVM IR`
- Ship `LLVM IR`

Decompile `Machine Code` → `LLVM IR`

Benefits	Challenges
• Easy to adopt • No compiler support needed	• Reconstructing code and control flow • Variable recovery • Function & ABI rules recovery

- Tools Available: QEMU, BAP, Dagger, Mcsema, Fracture

“Annotated” `Machine Code` $\longrightarrow$ `LLVM IR`

Benefits	Challenges

• Effective reconstruction
• Minimal compiler support needed
	• Annotations must be
	  • Minimal
	  • Compiler & IR independent
	• Adoption

- Tools Available: None

Ship `LLVM IR`

Benefits	Challenges
• *No loss* of information	• Adoption in Non LLVM based compilers • Code size bloat

- Tools Available: Portable Native Client, Renderscript, iOS, watchOS and tvOS apps.

Outline

Our Approach

Long term goal

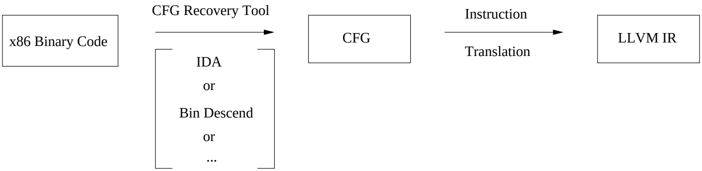
Minimal compiler-independent annotations to reconstruct high-quality IR

Short term goals

1. Experiment with `Machine Code` → `LLVM IR`, to **understand** the challenges better
    - To select from existing decompilation frameworks.
    - Experiment with different variable and type recovery strategies
2. Design suitable annotations for what cannot be inferred without them

Selecting mcsema

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



Instruction Translation: Memory Model

```
start:
  mov eax, [esp − 4h]
  add eax, 1
  ret
```

Binary Code

Translation →

```
RECOVERED_FUNC( RegisterContex) :
  VAR_EAX = RegisterContext.EAX
  VAR_ESP = RegisterContext.ESP

  VAR_EAX = [VAR_ESP − 4 ]
  VAR_EAX += 1

  RegisterContext.EAX = VAR_EAX
  RegisterContect.ESP = VAR_ESP

END
```

High level view of Recovered Code

mcsema: Demo

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

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

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

```
memset(password, '\0', len);
free(password);
```

$\longrightarrow$

```
free(password);
```