allvm - Binary Decompilation

Sandeep Dasgupta

University of Illinois Urbana Champaign

March 31, 2016

Goal & Motivation

Possible Approaches

Our Approach

Present Status

- 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 and dynamic optimizations

Goal & Motivation
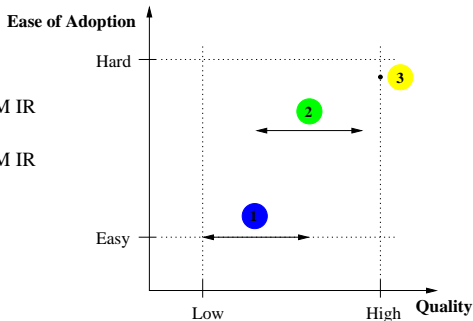
Possible Approaches

Our Approach

Present Status

3 Possible Approaches

Research Goal: Obtain "richer" LLVM IR than native machine code.

Possible Approaches

1 Decompile Machine Code ⟶ LLVM IR

2 "Annotated" Machine Code ⟶ LLVM IR

3 Ship LLVM IR

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

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

Benefits	Challenges

• Effective reconstruction
• Minimal compiler support needed
	• Annotations to 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, tvOS apps, ThinLTO

Outline

Goal & Motivation

Possible Approaches

Our Approach

Present Status

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

Goal & Motivation

Possible Approaches

Our Approach

Present Status

Present Status

Action Items	Status
Selected "mcsema" among the existing `Machine Code → LLVM IR` solutions. • Comparison of mcsema with existing tools • Evaluation of mcsema	Done
Literature survey on variable, type, function param recovery	Done
Implementing a variable and type recovery model using mcsema	Ongoing

- Actively supported and open sourced
- Well documented
- Functional `LLVM IR`
- Separation of modules: CFG recovery and Instruction translation (`CFG` $\rightarrow$ `LLVM IR`)

```
x86 (32 or 64) Binary Code ──────▶  ┌──────────────┐   CFG   ┌──────────────────────┐  LLVM IR
                                     │ CFG Recovery │ ──────▶ │ Instruction Traslation│ ──────▶
                                     │     Tool     │         └──────────────────────┘
                                     └──────────────┘
                                     ┌──────────────┐
                                     │   IDA Pro    │
                                     │     or       │
                                     │  Bin Descend │
                                     │     or       │
                                     │     ...      │
                                     └──────────────┘
```
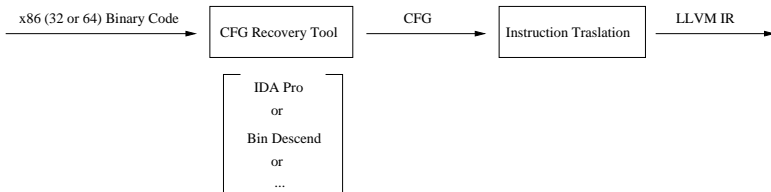
Instruction Translation

- Processor state: Modeled as struct of ints
- Processor memory: Modeled as flat array of bytes

start:

    mov eax, [esp − 4h]

    add eax, 1

    ret

Binary Code

Translation →

RECOVERED_FUNC (struct RegisterContext regctx) :

    VAR_EAX = regctx.EAX

    VAR_ESP = regctx.ESP

    VAR_EAX = [VAR_ESP − 4]

    VAR_EAX += 1

    regctx.EAX = VAR_EAX

    regctx.ESP = VAR_ESP

END

High level view of Recovered Code

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

Variable, Type, Function Param Recovery

- Enables
  - Fundamental analysis (Dependence, Pointer analysis)
  - Optimizations (register promotion)
- State of the art
  - Divine
    - State of the art variable recovery
  - TIE
    - Type recovery
  - Second Write
    - Heuristics for function parameter detection
    - Scalable variable and type recovery

Today: Functioning translation from `Machine Code` →
executable `LLVM IR` (IR quality is poor)

Questions ?

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);