



allvm - Binary Decompilation

Sandeep Dasgupta
University of Illinois Urbana Champaign

March 30, 2016



Goal & Motivation

Possible Approaches

Our Approach

Present Status



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 and dynamic optimizations



Outline

Goal & Motivation

Possible Approaches

Our Approach

Present Status

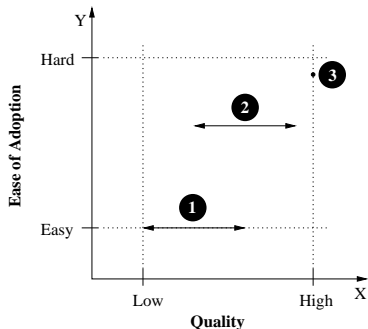


3 Possible Approaches

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





Decompile Machine Code \rightarrow LLVM IR

Benefits	Challenges
<ul style="list-style-type: none">• Easy to adopt• No compiler support needed	<ul style="list-style-type: none">• Reconstructing code and control flow• Variable recovery• Type recovery• Function & ABI rules recovery

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



“Annotated” Machine Code \rightarrow LLVM IR

Benefits	Challenges
<ul style="list-style-type: none">• Effective reconstruction• Minimal compiler support needed	<ul style="list-style-type: none">• Annotations to be<ul style="list-style-type: none">• Minimal• Compiler & IR independent• Adoption

- Tools Available: None



Ship LLVM IR

Benefits	Challenges
<ul style="list-style-type: none">• <i>No loss of information</i>	<ul style="list-style-type: none">• 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

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



Outline

Goal & Motivation

Possible Approaches

Our Approach

Present Status



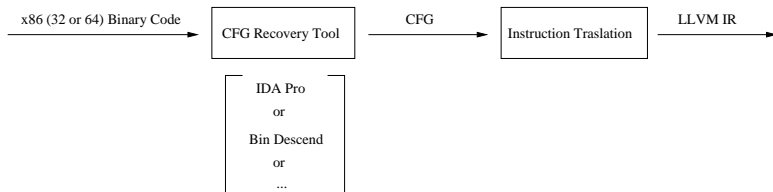
Present Status

- Selected “mcsema” among the existing Machine Code → LLVM IR solution.
 - Analysing the quality of IR recovery.
 - Current support and limitations.
- Literature survey on variable & type recovery research
- (Under Progress) Implementing a variable and type recovery model using mcsema



Selecting mcsema

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





Instruction Translation

- Processor state: Modelled as struct of ints
- Processor memory: Modelled 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



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



Variable, Type & Function Parameter Recovery

- Enables
 - Fundamental analysis (Dependence, Pointer analysis)
 - Optimizations (register promotion)
- 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

Goal & Motivation

Possible Approaches

Our Approach

Present Status



Questions ?



Outline

Goal & Motivation

Possible Approaches

Our Approach

Present Status



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);
```

—————>

```
free(password);
```