Static Binary Rewriting

R. Sekar

October 21, 2021

Binary Instrumentation

Why Binaries?

- Unavailability of source code
- Ease of deployment
- Completeness: Low-level libraries and hand-written assembly
- Soundness: Compiler optimizations can eliminate security-critical code

Challenges of Working With Binaries

- Size and complexity of instruction sets such as x86 and ARM.
  - Techniques often limited to a single processor
  - Only a subset of instructions supported
- High performance overheads
  - Dynamic instrumentation (e.g., Pin) is robust, but slow.
  - *Static instrumentation can be fast, but faces challenges on large/complex binaries.*

Overcoming Challenges: Instruction Set Complexity

- Modern instruction sets are complex
  - Intel's manual is 1500+ pages and 1100+ instructions
  - ARM's manual is over 1000 pages (and growing!)
  - *Frequent additions of ISA extensions*
- Solution: Translate ("lift") assembly to a higher level, architecture-independent intermediate representation (IR)
- *But:* manual modeling is tedious, error-prone, and impossible to keep up with
  - Most existing tools support only the top one or two architectures.
  - What about non-main-stream processors, e.g., in IoT environments?

Overcoming Challenges: Instruction Set Complexity

- Modern compilers (e.g., GCC) can generate code for numerous architectures
  1. *Source* $\longrightarrow$ *IR:* Translate source code to architecture-neutral intermediate representation
  2. *IR* $\longrightarrow$ *Asm:* Translate IR to assembly using architecture-specific *machine descriptions*
- IR contains detailed semantics that has been is extensively tested
- Question: Can we reverse the IR to assembly translation process?
  - Lifts assembly to a common IR that is simpler to analyze

LISC: Learning Instr. Semantics from Compilers

- Black-box approach: does not depend on gcc internals
- Learns Asm $\rightarrow$ RTL (gcc's IR) mapping from examples
  - Almost an endless supply of examples available!
  - LISC learns a decision tree with variables
    - Not a standard classification problem: we are learning a function
    - Must ensure sound translation in all cases

LISC Approach

1. Collect training data
   - Compile many packages to collect $\langle rtl, asm \rangle$ pairs

LISC Approach

1. Collect training data
   - Compile many packages to collect $\langle rtl, asm \rangle$ pairs
2. Parameterize: for each pair $\langle rtl, asm \rangle$
   - Parse *rtl* and *asm* into trees
   - identify the parameters (leaves)
   - compute the mapping between them

⟨sub $34, %rbx
(set (reg rbx) (plus (reg rbx) (const -34)))⟩

LISC Approach

1. Collect training data
   - Compile many packages to collect $\langle rtl, asm \rangle$ pairs
2. Parameterize: for each pair $\langle rtl, asm \rangle$
   - Parse *rtl* and *asm* into trees
   - identify the parameters (leaves)
   - compute the mapping between them

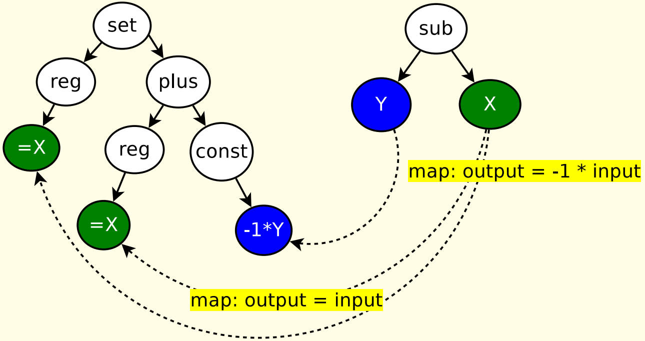
⟨sub $ 34 , % rbx
(set (reg rbx) (plus (reg rbx) (const - 34)))⟩

LISC Approach

1. Collect training data
   - Compile many packages to collect ⟨*rtl*, *asm*⟩ pairs
2. Parameterize: for each pair ⟨*rtl*, *asm*⟩
   - Parse *rtl* and *asm* into trees
   - identify the parameters (leaves)
   - compute the mapping between them

⟨sub $ 34, % rbx

(set (reg rbx) (plus (reg rbx) (const - 34)))⟩

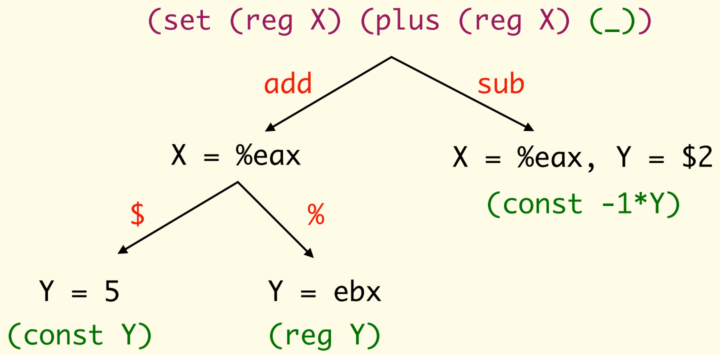


LISC Approach

1. Collect training data
   - Compile many packages to collect $\langle rtl, asm \rangle$ pairs
2. Parameterize: for each pair $\langle rtl, asm \rangle$
   - Parse *rtl* and *asm* into trees
   - identify the parameters (leaves)
   - compute the mapping between them
3. Build transducer from parameterized pairs
   - transducer is an automaton similar to Moore/Mealy machine
   - input is *asm* tree, output is *rtl* tree

Transducer Construction Example

```
add %ebx, %eax → (set (reg eax) (plus (reg eax) (reg ebx)))
add $5, %eax   → (set (reg eax) (plus (reg eax) (const 5)))
sub $2, %eax   → (set (reg eax) (plus (reg eax) (const -2)))
```



LISC: Evaluation

- Completeness:
  - *99.5%* of x86 and *99.8%* of ARM instructions achieved
    - after training with about 10 chosen binaries
  - Remaining are mostly NOPs and other obsolete instructions (e.g., BCD arithmetic)
- Soundness:
  - Proved under reasonable assumptions
    - context-independent translation of RTLs into assembly
  - Also experimentally verified on core instructions
- Now LISC v2 supports x86_64
  - Work done originally on x86_32
  - Download from `http://seclab.cs.sunysb.edu/seclab/liscV2/`

Static Binary Instrumentation: Challenges

- Robust static disassembly
  - Including low-level libraries and hand-written assembly
- Static instrumentation without breaking complex code
  - Fixing up indirect control transfers
  - Fixing up direct transfers
  - Tolerating disassembly errors
- Secure instrumentation
  - Ensure instrumentation of *all* code
  - Ensure that added security checks cannot be bypassed

Static Disassembly: BinCFI approach

- Take advantage of the fact that the presence of data in code is rare
- Use linear disassembly, followed by error detection and correction
- Error detection is based on control flow consistency
- Tolerate disassembly errors:
  - Ensure that if data is disassembled as code, that does not cause misbehavior of instrumented code

Pointer Fixup

- Direct control transfers: Instrument assembly code
  - "Reassemblable disassembly:" Disassembled code can be reassembled into binary with full preservation of behavior
  - Use labels so that the assembler can figure out actual instruction offsets etc.
- Indirect control transfers:
  - Static analysis to discover all possible code pointers
    - Conservative approach: may include non-code pointers, but cannot leave out legitimate ones
  - Address translation to translate at runtime
    - Provides most transparency benefits of dynamic translation techniques

Safe and Secure Instrumentation

- Make a second copy of code and instrument it
  - It is OK if you disassemble and instrument data, as the original data is left in place
- Control-flow integrity ensures that only disassembled code is instrumented
  - If some code is somehow missed, it leads to failure rather than security violation
- CFI also protects all the added instrumentation
  - CFI disallows “jumping past instrumentation”

BinCFI Results

- Supports large and low-level COTS (“stripped”) binaries
  - glibc, Firefox, Adobe Reader, gimp, etc.
    - Over 300MB of (intel 32-bit) binaries in total.
- Eliminates 99% of control-flow targets and 93% of possible gadgets
  - Remaining gadgets provide very limited capability
- Good performance while providing full transparency
  - About 10% overhead on CPU-intensive C-benchmarks, somewhat higher for C++ programs

Static Instrumentation: Further Performance Improvements...

- Most of BinCFI's overhead comes from runtime code pointer translation
- *Question:* Can we avoid this runtime translation?
  - Requires code pointers to be translated at instrumentation time

Static Instrumentation: Further Performance Improvements...

- Most of BinCFI's overhead comes from runtime code pointer translation
- *Question:* Can we avoid this runtime translation?
  - Requires code pointers to be translated at instrumentation time
- *Yes:* For 64-bit position-independent binaries
  - Almost all code on modern Linux distributions falls in this category
  - Pointers are all explicitly identified in these binaries
    - but there is no information on which of these point to code

Static Instrumentation: Further Performance Improvements...

- Most of BinCFI's overhead comes from runtime code pointer translation
- *Question:* Can we avoid this runtime translation?
  - Requires code pointers to be translated at instrumentation time
- *Yes:* For 64-bit position-independent binaries
  - Almost all code on modern Linux distributions falls in this category
  - Pointers are all explicitly identified in these binaries
    - but there is no information on which of these point to code
- Approach: Develop static analysis to distinguish code and data pointers
  - Relies on detailed instruction semantics

Influential Work in Binary Instrumentation by Past Students of CSE 509

- Fine-grained binary code randomization [ACSAC '20, ACSAC '17]
- Accurate detection of function boundaries in binaries [DSN '17]
- Extracting instruction semantics from compiles [FSE '17, ASPLOS '16]
- Binary instrumentation for ROP Defense [ACSAC '15]
- Code and Control-flow integrity [ACSAC '15]
- Platform for Static Binary Instrumentation [VEE '14]
- Control-flow Integrity for COTS Binaries [USENIC Sec '13, Best paper award]