There and back again
Binary Analysis with mcsema

Andrew Ruef
There and back again
Street Fighting
Binary Analysis with mcsema

Andrew Ruef
Hi

• Now:
  • PhD Programming Languages
  • Advised by Mike Hicks
  • Research at Trail of Bits

• Before:
  • Startups
  • Defense contractors
  • Big companies
Introduction
Problem: binary programs
What if humans didn’t read it?

- We ask machines to do everything else
- Let’s have them read native instructions and analyze them just like they analyze other programs
- What new problems show up?
- What existing problems are magnified?
- Does anything get easier?
What if humans didn’t read it?

• We ask machines to do everything else
• Let’s have them read native instructions and analyze them just like they analyze other programs
• What new problems show up?
• What existing problems are magnified?
• Does anything get easier?
  • Trick question, nothing ever gets easier
Native Instructions
What’s in machine code?

“I’ve never seen the inside of a rabbit’s brain before. What’s in there, anyway?”

“Nobody knows yet. Johnson and I are hoping it’s cupcakes.”
What’s in machine code?

- Statements that look like this

```
mov     eax, [ebp-4]
movzx   eax, byte ptr [eax]
mov     [ebp-9], al
cmp     byte ptr [ebp-9], 0
```

- The code that contains those statements itself

- Some entry point
What’s in machine code?
What’s in machine code?

Not stack or heap, just “memory”
What instructions does this miss?

• Does your model include multiple threads?
  • If no, then you miss xbegin / xcommit / xabort

• Does your model include devices and privilege levels?
  • If no, then you miss (some of) the behavior of iret and friends

• What about individual page permissions and virtual memory?
  • Then you miss implicit exceptions due to page permissions
Can we make instructions explicit?

• What if we used some pure, core language to represent transitions on states?
• Spoiler alert – this is what everyone does
• We’ll use LLVM for this language, for reasons I will defend later
Compilers and other instruction sources
Provenance

- What produces instructions? Compilers, right?
  - That’s a big assumption

- What rules do compilers have to play by?
  - Their own The ABI

- What’s the gap between what compilers *must* do and what they *frequently* do?
  - Significant
Binary or compiler output analysis?
... more simply

Compilers / configurations you've seen before

Everything else

INCREASING WEIRDNESS
Don’t be these people

Kaspersky Lab Experts Discover Unknown Programming Language in the Duqu Trojan; Appeal to Programming Community for Support in Analysis

11 Mar 2012
Virus News

The language, which DuQu used to communicate with command-and-control servers, turns out to be a special type of C code compiled with the Microsoft Visual Studio Compiler 2008.
Why should you care?

• Compromise is the essence of diplomacy having a working / scalable system

• You can’t handle all the weirdness that the system has to offer

• Know the gaps

• Also know where systems will fail?
Breaking assumptions

• Undefined flags used in control decisions

• Lots of control flow through memory

• No stack / all data accesses through push and pop
Motivating mcseema
mcsema

- Translate X86 into IR
- LLVM translation
- Function identification
- Stack translation
- KLEE
Goal: take X86, put it into an IR

- Sub goals:
  - Have collaborators
  - Produce executable from IR
  - Do some static analysis

- What IR to use?
  - Use an existing one
  - Make our own
What about VEX?

- Valgrind is a dynamic binary translator
- DBTs have the same problems we do
- Valgrind represents the semantics of native programs as VEX
- VEX is nasty
  - Small number of expressions and statements
  - ~1600 values in the binop op enumerator
Tradeoffs we’ll make

• Fewer fancy abstractions like memory
  • No assumptions about stack or heap

• Some assumptions about code
  • Immutable

• An interconnected mass of pulsating maggots components

• Take native code and print it as LLVM
Why LLVM?

- Lots of thought went into the design of the IR
  - If not LLVM, then we would reproduce this thought and surely get something wrong

- Lots of tools exist to work with this IR
  - Symbolic executors, abstract interpreters, code generators, optimizers

- The type system of the IR is already close to what the machine is
  - No signed / unsigned types, integer bit vector machines

- Existing LLVM expertise is transferrable

- Some of these reasons are political, some are engineering
Anatomy of a decoder

• Machine state is represented as an LLVM record type
  • Registers are field members

• Translated instructions are sequences of LLVM instructions that modify the machine state

• Machine state is spilled to the stack on function entry, synced on function call and function return
Flags

- EFLAGS is broken out as a sequence of 1-bit virtual registers in the machine state
- Instructions set registers, now they also set flag registers

- Lots of flag assignment code is dead by construction
- Conservative DCE removes “lots” of flag assignment code

- Undefined flags set to LLVM undefined value
Translation example

and ebx, 0x44444
Translation example

%79 = load i64* %RBX_val
%80 = trunc i64 %79 to i32

%81 = and i32 %80, 279620

%82 = lshr i32 %81, 31
%83 = trunc i32 %82 to i1
store i1 %83, i1* %SF_val
%84 = icmp eq i32 %81, 0
store i1 %84, i1* %ZF_val
%85 = trunc i32 %81 to i8
%86 = call i8 @llvm.ctpop.i8(i8 %85)
%87 = trunc i8 %86 to i1
%88 = xor i1 %87, true
store i1 %88, i1* %PF_val
store i1 false, i1* %OF_val
store i1 false, i1* %CF_val
store i1 undef, i1* %AF_val

%89 = zext i32 %81 to i64
store i64 %89, i64* %RBX_val
Function specification

- We only really need one function
- The specification of the CFG also specifies the functions
- This is cheating

- The further away you get from compiler output the less meaning “function” has
Virtual Stacks

Before
After

Stack Frame (esp)

Virtual Stack Frame (v_esp)

- push ebx
  - m[v_esp] = v_esp
  - v_esp = v_esp - 4

- pop eax
  - v_eax = m[v_esp]
  - v_esp = v_esp + 4

42

Original Instruction

mcsema instructions
Advantages / disadvantages

- Sound model of the stack
- No abstraction of variables
  - Kills optimizations, symbolic execution
- Large running time cost

- Fix: every variable identified and moved off of the virtual stack is space saved and maybe code optimized
Tangent: Infer Functions?

• Observation: compilers produce one activation record per function, and functions are generally related to data values stored in this activation record

• Hypothesis: compilers emit instructions such that instructions with code locality cluster with values on the stack with data locality

• This seems true for C and the C compilers we know about
  • Is it true for all HLLs?
  • Must it be true for all C compilers?
Tangent: Infer Functions?
Platform specific special cases

- What about threads?
  - New threads are basically the creation of a new machine state

- What about exceptions? Like SEH?
  - Ugh
Enough to run KLEE on binaries
Abstractions

- Control Flow Analysis
- Memory and the heap
- Type recovery
Control Flow Analysis

- Any errors during CFA corrupt all subsequent analyses

- Overall: convert instruction stream into a control flow graph

- In general, quite hard
Control Flow Analysis

- Some possibilities
  - Use symbolic execution
    - INSIGHT
  - Use abstract interpretation and value set / value range analysis
    - Jakstab, bindead, BAP
  - Use lots of distinct traces and merge them

- All with their advantages and disadvantages
CFA in mcsema

• Control flow specified externally

• Default: specify control flow of application using IDA, export to mcsema
  • Advantages: empirically good results for compiler output analysis
  • Disadvantages: theoretically unfulfilling

• In the future: some form of value range analysis on indirect branches
Memory and the heap

- A sound abstraction: all of memory is a key / value store
  - aka a big flat array

- Some big downsides: optimizer doesn’t know that stack variables are variables

- Would like to be able to allow mscema to try and register allocate stack variables
Memory and the heap

- Heap objects are manipulated via integer pointers and offsets to those pointers

- Downside: analyses can’t do a semantics or type driven analysis of record uses
  - Because there are no records to speak of!

- This is edging us closer and closer towards...
Type Recovery

• Assign some type information to values in the (partially) recovered program

• Assists human analysts understand the program

• Assists automated analyses to be more precise and perform better
  • Optimizations can know what variables are now
  • Symbolic executors can know what regions of memory are disjoint and have different widths
An advantage of LLVM

- The same type infrastructure used to represent the original program (*) is available to represent the recovered programs types!

- Saves you from having to define your own type system

* MANY LARGE CAVEATS
Primitive types

- Partition the type of values into
  - Pointer vs not?
  - Integer widths?
Typing a stack frame

- Some problems addressed by very recent work (Noonan et al PLDI16)
  - What if a stack slot is re-used between a signed and unsigned type?
  - What about polymorphic functions?

- Some remain:
  - How do you type a stack frame that contains an alloca?
  - How do you type malloc in general?
Present status, future, conclusion
What translates now

- Modestly sized (1-40 KLOC) C/C++ programs for Linux and Windows
- Web servers
- CGC challenge binaries
Currently cooking

- A better variable specification scheme as input to mcsema
- A dependent type system for machine code
- Using C as a DSL to specify instruction semantics
Wish list

• Implementation of a better control flow analysis scheme
  • Iterated refinements of recursive descent using value range analysis would be a start

• A better symbolic execution system for LLVM
Thanks!