



Platform-independent static binary code analysis using a meta- assembly language

Thomas Dullien, Sebastian Porst
zynamics GmbH

CanSecWest 2009

Overview

The REIL Language

Abstract Interpretation

MonoREIL

Results



Motivation

- Bugs are getting harder to find
- Defensive side (most notably Microsoft) has invested a lot of money in a „bugocide“
- Concerted effort: Lots of manual code auditing aided by static analysis tools
- Phoenix RDK: Includes „lattice based“ analysis framework to allow pluggable abstract interpretation in the compiler



Motivation

- Offense needs automated tools if they want to avoid being sidelined
- Offensive static analysis: Depth vs. Breadth
- Offense has no source code, no Phoenix RDK, and should not depend on Microsoft
- We want a static analysis framework for offensive purposes

Overview

The REIL Language

Abstract Interpretation

MonoREIL

Results



REIL

- Reverse Engineering Intermediate Language
- Platform-Independent meta-assembly language
- Specifically made for static code analysis of binary files
- Can be recovered from arbitrary native assembly code
 - Supported so far: x86, PowerPC, ARM



Advantages of REIL

- Very small instruction set (17 instructions)
- Instructions are very simple
- Operands are very simple
- Free of side-effects
- Analysis algorithms can be written in a platform-independent way
 - Great for security researchers working on more than one platform



Creation of REIL code

- Input: Disassembled Function
 - x86, ARM, PowerPC, potentially others
- Each native assembly instruction is translated to one or more REIL instructions
- Output: The original function in REIL code

Example

```

597DA51B netapi32.xpsp3.dll::sub_597DA51B
597DA53E cmp word ax, word si
597DA541 jz cs:loc_597DD83D
  
```

```

597DA53E00: and eax, 65535, t1
597DA53E01: and esi, 65535, t3
597DA53E02: and t1, 32768, t4
597DA53E03: and t3, 32768, t5
597DA53E04: sub t1, t3, t6
597DA53E05: and t6, 32768, t7
597DA53E06: bsh t7, -15, SF
597DA53E07: xor t4, t5, t8
597DA53E08: xor t4, t7, t9
597DA53E09: and t8, t9, t10
597DA53E0A: bsh t10, -15, OF
597DA53E0B: and t6, 65536, t11
597DA53E0C: bsh t11, -16, CF
597DA53E0D: and t6, 65535, t12
597DA53E0E: bisz t12, , ZF
597DA54100: jcc ZF, , 1501419581
  
```



Design Criteria

- Simplicity
- Small number of instructions
 - Simplifies abstract interpretation (more later)
- Explicit flag modeling
 - Simplifies reasoning about control-flow
- Explicit load and store instructions
- No side-effects



REIL Instructions

- One Address
 - Source Address * 0x100 + n
 - Easy to map REIL instructions back to input code
- One Mnemonic
- Three Operands
 - Always
- An arbitrary amount of meta-data
 - Nearly unused at this point



REIL Operands

- All operands are typed
 - Can be either registers, literals, or sub-addresses
 - No complex expressions
- All operands have a size
 - 1 byte, 2 bytes, 4 bytes, ...



The REIL Instruction Set

- Arithmetic Instructions
 - ADD, SUB, MUL, DIV, MOD, BSH
- Bitwise Instructions
 - AND, OR, XOR
- Data Transfer Instructions
 - LDM, STM, STR



The REIL Instruction Set

- Conditional Instructions
 - BISZ, JCC
- Other Instructions
 - NOP, UNDEF, UNKN
- Instruction set is easily extensible



REIL Architecture

- Register Machine
 - Unlimited number of registers t_0, t_1, \dots
 - No explicit stack
- Simulated Memory
 - Infinite storage
 - Automatically assumes endianness of the source platform



Limitations of REIL

- Does not support certain instructions (FPU, MMX, Ring-0, ...) yet
- Can not handle exceptions in a platform-independent way
- Can not handle self-modifying code
- Does not correctly deal with memory selectors

Overview

The REIL Language

Abstract Interpretation

MonoREIL

Results



Abstract Interpretation

- Theoretical background for most code analysis
- Developed by Patrick and Rhadia Cousot around 1975-1977
- Formalizes „static abstract reasoning about dynamic properties“
- Huh ?
- A lot of the literature is a bit dense for many security practitioners



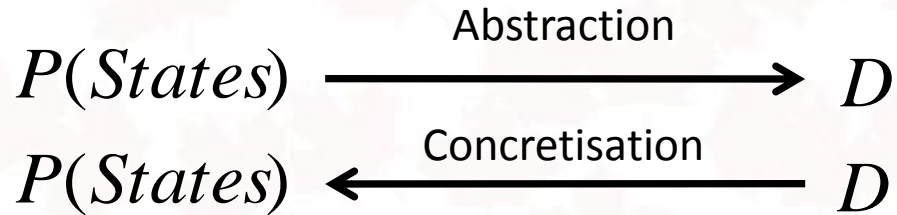
Abstract Interpretation

- We want to make statements about programs
- Example: Possible set of values for variable x at a given program point p
- In essence: For each point p , we want to find $K_p \in P(\text{States})$
- Problem: $P(\text{States})$ is a bit unwieldly
- Problem: Many questions are undecidable (where is the w*inker that yells „halting problem“) ?



Dealing with unwieldy stuff

- Reason about something simpler:



- Example: Values vs. Intervals



Lattices

- In order for this to work, D must be structurally similar to $P(\text{States})$
- $P(\text{States})$ supports intersection and union
- You can check for inclusion (contains, does not contain)
- You have an empty set (bottom) and „everything“ (top)



Lattices

- A lattice is something like a generalized powerset
- Example lattices: Intervals, Signs, $P(\text{Registers})$, mod p



Dealing with halting

- Original program consists of $p_1 \dots p_n$ program points
- Each instruction transforms a set of states into a different set of states
- $p_1 \dots p_n$ are mappings $P(\text{States}) \rightarrow P(\text{States})$
- Specify $p'_1 \dots p'_n : D \rightarrow D$
- This yields us $\tilde{p} : D^n \rightarrow D^n$



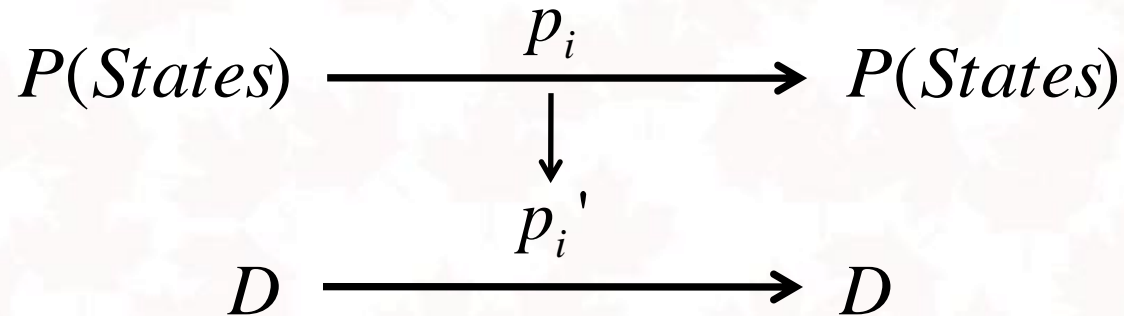
Dealing with halting

- We cheat: Let D be finite $\rightarrow D^n$ is finite
- Make sure that \tilde{p} is monotonous (like this talk)
- Begin with initial state l
- Calculate $\tilde{p}(l)$
- Calculate $\tilde{p}(\tilde{p}(l))$
- Eventually, you reach $\tilde{p}^n(l) = \tilde{p}^{n-1}(l)$
- You are done – read off the results and see if your question is answered



Theory vs. practice

- A lot of the academic focus is on proving correctness of the transforms



- As practitioner we know that p_i is probably not fully correctly specified
- We care much more about choosing and constructing a D so that we get the results we need



Overview

The REIL Language

Abstract Interpretation

MonoREIL

Results



MonoREIL

- You want to do static analysis
- You do not want to write a full abstract interpretation framework
- We provide one: MonoREIL
- A simple-to-use abstract interpretation framework based on REIL



What does it do ?

- You give it
 - The control flow graph of a function (2 LOC)
 - A way to walk through the CFG (1 + n LOC)
 - The lattice D (15 + n LOC)
 - Lattice Elements
 - A way to combine lattice elements
 - The initial state (12 + n LOC)
 - Effects of REIL instructions on D (50 + n LOC)



How does it work?

- Fixed-point iteration until final state is found
- Interpretation of result
 - Map results back to original assembly code
- Implementation of MonoREIL already exists
- Usable from Java, ECMAScript, Python, Ruby

Overview

The REIL Language

Abstract Interpretation

MonoREIL

Results



Register Tracking

- First Example: Simple
- Question: What are the effects of a register on other instructions?
- Useful for following register values



Register Tracking

- Demo



Register Tracking

- Lattice: For each instruction, set of influenced registers, combine with union
- Initial State
 - Empty (nearly) everywhere
 - Start instruction: { tracked register }
- Transformations for MNEM op1, op2, op3
 - If op1 or op2 are tracked → op3 is tracked too
 - Otherwise: op3 is removed from set



Negative indexing

- Second Example: More complicated
- Question: Is this function indexing into an array with a negative value ?
- This gets a bit more involved



Negative indexing

- Simple intervals alone do not help us much
- How would you model a situation where
 - A function gets a structure pointer as argument
 - The function retrieves a pointer to an array from an array of pointers in the structure
 - The function then indexes negatively into this array
- Uh. Ok.



Abstract locations

- For each instruction, what are the contents of the registers ? Let's slowly build complexity:
- If `eax` contains `arg_4`, how could this be modelled ?
 - `eax = *(esp.in + 8)`
- If `eax` contains `arg_4 + 4` ?
 - `eax = *(esp.in + 8) + 4`
- If `eax` can contain `arg_4+4`, `arg_4+8`, `arg_4+16`, `arg_4 + 20` ?
 - `eax = *(esp.in + 8) + [4, 20]`



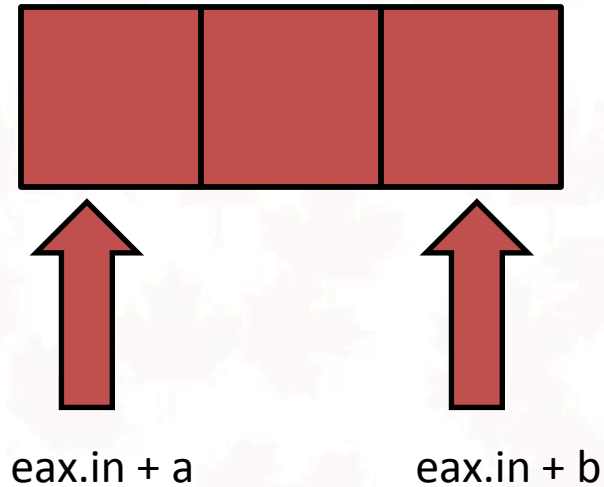
Abstract locations

- If `eax` can contain `arg_4+4`, `arg_8+16` ?
 - $eax = *(esp.in + [8,12]) + [4,16]$
- If `eax` can contain any element from
 - `arg_4` → `mem[0]` to `arg_4` → `mem[10]`, incremented once, how do we model this ?
 - $eax = (*(esp.in + [8,8]) + [4, 44]) + [1,1]$
- OK. An abstract location is a base value and a list of intervals, each denoting memory dereferences (except the last)



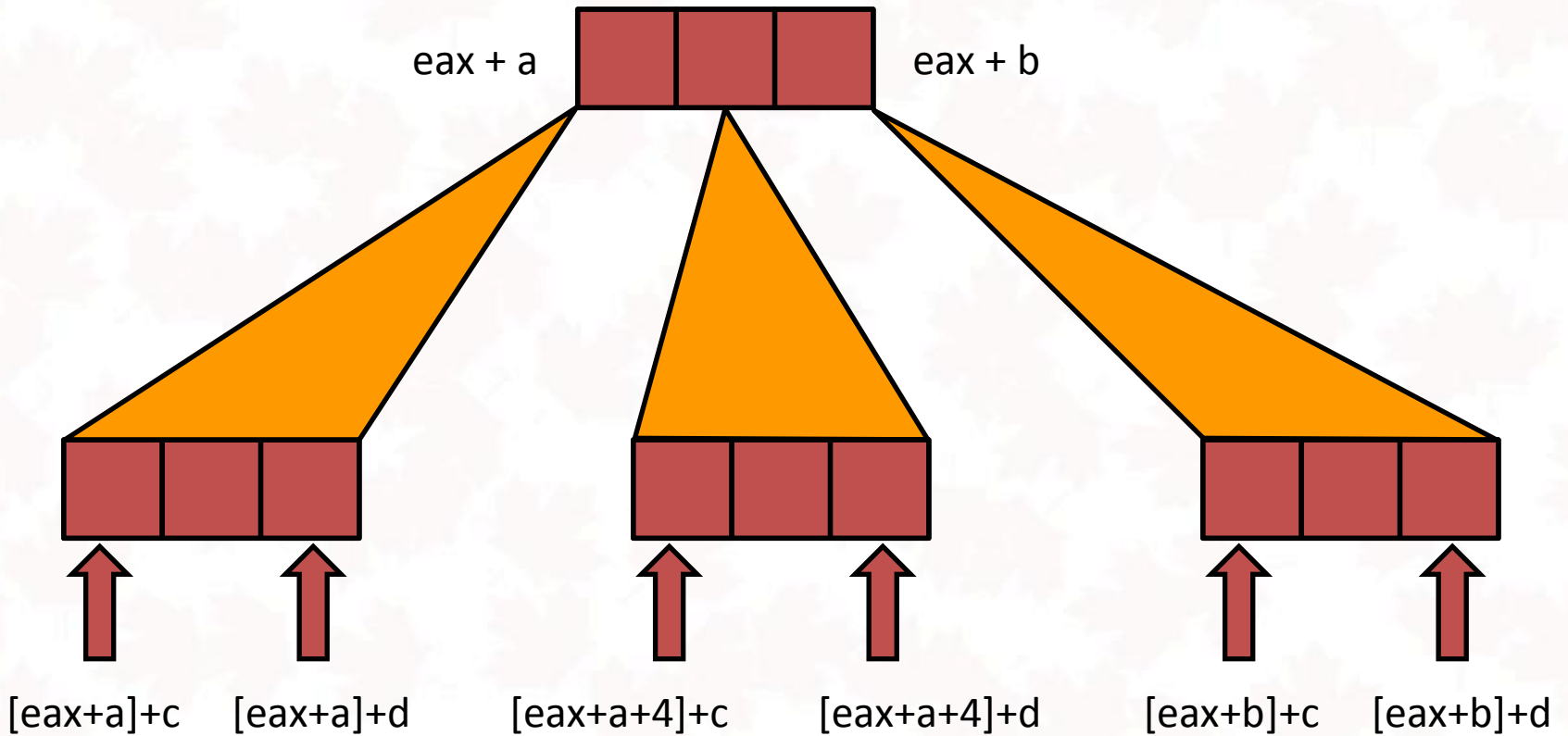
Range Tracking

$\text{eax.in} + [a, b] + [0, 0]$



Range Tracking

$eax + [a, b] + [c, d] + [0, 0]$





Range Tracking

- Lattice: For each instruction, a map:

$$\text{Register} \cup \text{Aloc} \rightarrow \text{Aloc}$$

- Initial State
 - Empty (nearly) everywhere
 - Start instruction: { reg -> reg.in + [0,0] }
- Transformations
 - Complicated. Next slide.



Range Tracking

- Transformations
 - ADD/SUB are simple: Operate on last intervals
 - STM $op_1, , op_3$
 - If op_1 or op_3 not in our input map M skip
 - Otherwise, $M[M[op_3]] = op_1$
 - LDM $op_1, , op_3$
 - If op_1 or op_3 is not in our input map M skip
 - $M[op_3] = M[op_1]$
 - Others: Case-specific hacks



Range Tracking

- Where is the meat ?
- Real world example: Find negative array indexing



MS08-67

- Function takes in argument to a buffer
- Function performs complex pointer arithmetic
- Attacker can make this pointer arithmetic go bad
- The pointer to the target buffer of a wcsncpy will be decremented beyond the beginning of the buffer



MS08-67

- Michael Howard's Blog:
 - “In my opinion, hand reviewing this code and successfully finding this bug would require a great deal of skill and luck. So what about tools? It's very difficult to design an algorithm which can analyze C or C++ code for these sorts of errors. The possible variable states grows very, very quickly. It's even more difficult to take such algorithms and scale them to non-trivial code bases. This is made more complex as the function accepts a highly variable argument, it's not like the argument is the value 1, 2 or 3! Our present toolset does not catch this bug.”



MS08-67

- Michael is correct
 - He has to defend all of Windows
 - His „regular“ developers have to live with the results of the automated tools
 - His computational costs for an analysis are gigantic
 - His developers have low tolerance for false positives



MS08-67

- Attackers might have it easier
 - They usually have a much smaller target
 - They are highly motivated: I will tolerate 100 false positives for each „real“ bug
 - I can work through 20-50 a day
 - A week for a bug is still worth it
 - False positive reduction is nice, but if I have to read 100 functions instead of 20000, I have already gained something



MS08-67

- Demo



Limitations and assumptions

- Limitations and assumptions
 - The presented analysis does not deal with aliasing
 - We make no claims about soundness
 - We do not use conditional control-flow information
 - We are still wrestling with calling convention issues
 - The important bit is not our analysis itself – the important part is MonoREIL
 - Analysis algorithms will improve over time – laying the foundations was the boring part



Status

- Abstract interpretation framework available in BinNavi
- Currently x86
- In April (two weeks !): PPC and ARM
 - Was only a matter of adding REIL translators
- Some example analyses:
 - Register tracking (lame, but useful !)
 - Negative array indexing (less lame, also useful !)



Outlook

- Deobfuscation through optimizing REIL
- More precise and better static analysis
- Register tracking etc. release in April (two weeks !)
- Negative array indexing etc. release in October
- Attempting to encourage others to build their own lattices



Related work ?

- Julien Vanegue / ERESI team (EKOPARTY)
- Tyler Durden's Phrack 64 article
- Principles of Program Analysis (Nielson/Nielson/Hankin)
- University of Wisconsin WISA project
- Possibly related: GrammaTech CodeSurfer x86

Questions ?



(Good Bye, Canada)