Lezione 10/10/23

Tipi di disassemblatori

Il disassemblatore è un tool che legge codice macchina (espresso in byte) e associa, per ogni sequenza di byte, un'istruzione macchina simbolica.

- **Lineari**: Dati dei valori numerici $v_1, ..., v_n$, cioè un flusso di byte, va ad associare ad ogni byte una istruzione macchina (nb: una istruzione macchina potrebbe richiedere più di un byte). Non è detto che tutte le istruzioni abbiano la stessa lunghezza (questo è vero in architetture RISC come ARM, dove $32\ bit \rightarrow 4\ byte\ \forall\ istruzione$, mentre in altre architetture, come CISC, non è vero.) Il disassemblatore lineare è però abbastanza *dummy*, non comprende il codice. Nel caso dei salti, non si preoccupa di dove saltare. Ad esempio: $v_1, v_2 \mid\ v_3,\ v_4 \mid$ se v_1, v_2 identificano un salto verso v_4 (l'istruzione macchina parte da v_4), ma il disassemblatore associa $2\ byte = 1\ istruzione$, allora "salterà" verso v_3. Un esempio di disassemblatore lineare è `objdump -d`.
- **Interattivi**: Segue la semantica delle istruzioni. Se c'è *jump*, si riallinea a dove arriva il salto e continua con l'analisi del codice macchina. Il set di istruzioni *escluse*,comprese tra *jump* e *dove arriva* viene lasciato all'utente. Questo perchè c'è un'*interazione* tra utente e disassemblatore, quindi se ottengo nuove conoscenze, le riverso nel disassemblatore e ottengo nuove informazioni. Se ad esempio riconosco una struttura dati, e la individuo, posso dire al disassemblatore di risolvere i riferimenti a questa struttura dati, in modo da non vedere più indirizzi generici, ma richiami a questa struttura. Un esempio di disassemblatore interattivo è **Ghidra**. GIà nel fornire un file da analizzare a Ghidra, questi ci fornirà informazioni sul file (formato, architettura, compilatore etc....) che possiamo "dare per buono" o non seguire. Ci chiederà inoltre cosa usare per l'analisi, alcune "tecniche" sono assodate ed affidabili, altre, in rosso, sono sperimentali. Generalmente quelle proposte vanno bene per i nostri scopi.

32 vs 64 bit

Classifichiamo in *x_86* a *32 bit* e *x_86_64 / AMD64* a *64 bit*. La maggior differenza risiede nella dimensione dei registri (*4 byte* contro *8 byte*), ed alcune differenze tra registri o istruzioni.

Intel vs AT&T

Nell'architettura *Intel* ho prima la destinazione (dove scrivo), poi la sorgente (dove leggo). Per *AT&T* è il contrario, abbiamo prima la sorgente (dove leggo), e dopo la destinazione (dove

scrivo). Esempio: `mov reg,8` quanti byte uso per rappresentare 8? dipende dal registro, se esso è a 32 o 64. `mov ptr,8`, quanto è grande l'indirizzo?

- In Intel l'istruzione è `mov DWORD ptr,8`
- In AT&T l'istruzione è `movl 8,ptr` dove con *l* intendiamo *long*, 4 byte. Abbiamo anche 1 byte (*b*), 2 byte (*w*), e 8 byte (*q*).
- Con il suffisso `-M` vedo sempre i suffissi.

Esadecimale, base 16

Si preferisce tale base perchè ha un mapping coi bit. Tra poco vedremo come. $257_{10} = 16^2 + 1 = (101)_{16} = 16^2 + 16^0$, è facile da convertirlo in binario, poichè basta scrivere ciascuno degli elementi sfruttando 4 bit : $0001|0000|0001$. Questa scrittura viene compattata con l'uso di numeri da 0 a 9 e lettere da *a* (1001) ad *f* (1111).
Ad esempio: $1010|1011|0011|0111|1111 = a|b|3|7|f$

Big Endian vs Little Endian

Prendiamo $259_{10} = 256 + 3 = 103_{16} = |0001|0000|0011|$ Se raggruppiamo a blocchi di 8 bit avremmo: $...0001|00000011 = 1|3$ (Quindi, se lavoro con 8 bit/2 byte, il numero più grande è *f f*, e posso rappresentare da 256 byte diversi, 2^8, da 0 a 255.) Con 8 bit dimezziamo lo spazio per memorizzare informazioni! Come rappresento $1|3$ in **memoria**?

- Con **Big Endian** il byte più significativo va a sinistra, quindi in memoria scrivo $1|3$.
- Con **Little Endian**, il byte più significativo va a destra, quindi in memoria scrivo $3|1$.

In realtà, qui abbiamo supposto un *ordinamento totale*, in realtà potrebbe anche essere *parziale*, dato v_1, v_2, v_3, v_4 possiamo avere:

- Little Endian Intel totale: $v_4, ..., v_1$
- Little Endian parziale: $v_2, v_1 | v_4, v_3$
- Big Endian: $v_1, ..., v_4$

Registri

Rappresentano memoria immediata che usa il processore.

In **X_86** abbiamo:

- *segmenti*: *CS* (dove sta il codice), *DS-ES-FS-GS* (data), *SS* (stack)

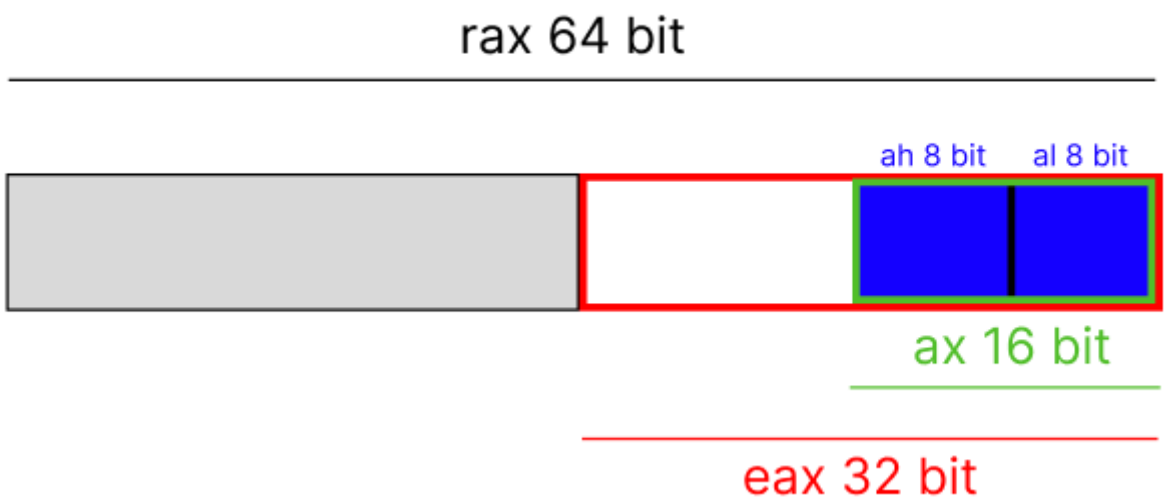
- *general purpose*: *eax* (accumulo) e sottoinsiemi *ax/ah/al ebx* e sottoinsiemi *bx/bh/bl ecx; edx; esi/si; edi/di; ebp/bp*
- *special purpose*: eflags (bit stato nel processore, interruzioni)/flags eip (instruction pointer) cr0, cr1...
- *floating point*: mmx0,...,mmx7 di tipo vettoriale usabili anche in virgola mobile, chiamandoli fpr0,...,frp7 xmm0,xmm15 anch'essi vettoriali.

La limitazione principale è che 32 registri sono pochi, quindi spesso si lavora con lo stack.

In **X_86_64** principalmente estendiamo la maggior parte dei registri:

- *general purpose*: *rax/eax/ax/ah/al*; *rbx; rcx*; *rsi*, ..., *rdi*, ..., *rbp*
- Aggiunta di nuovi registri *r8,...,r15*
- *special purpose*: *rflags*, *rip*.

Si lavora in pagine o segmenti. Non vedo i registri dei segmenti nel codice macchina, spesso sono *impliciti*. Ad esempio: `mov reg, dword ptr p`, in questo caso è implicito che, trattandosi di dati, uso il segmento *DS*. Se uso `call f`, con *f* funzione, allora uso *CS*, poichè si tratta di testo. Con `mov reg FS:[]` sto specificando di usare *FS*, altrimenti è sottointeso *DS*.



NB: L'Endianess si vede in memoria, **NON** nei registri! Quindi qui non mi preoccupo di dove collocare le cose.