AI Decisively Defeats Human Poker Players

Libratus lived up to its “balanced but forceful” Latin name by becoming the first AI to beat professional poker players at heads-up, no-limit Texas Hold'em. Developed by Carnegie Mellon University, the AI won the “Brains Vs. Artificial Intelligence” tournament against four poker pros by $1,766,250 in chips over 120,000 hands (games). Researchers can now say that the victory margin was large enough to count as a statistically significant win (99.7 percent certainty).

Libratus focuses on improving its own play, [described] as safer and more reliable compared to the riskier approach of trying to exploit opponent mistakes.

Adminstrivia

- Homework 3 released today, due next Thu (2/9)
- Lab 2 deadline pushed to Monday (2/13)
  - Definitely want to start before the Midterm

- **Midterm** (2/10) in lecture
  - Reference sheet + 1 *handwritten* cheat sheet
  - Find a study group! Look at past exams!
  - Aiming for average of 75%

- **Midterm review session** (2/7) in BAG 261 from 5-7:30pm
Procedures

- Stack Structure

- **Calling Conventions**
  - Passing control
  - **Passing data**
  - Managing local data

- Register Saving Conventions

- Illustration of Recursion
Procedure Data Flow

Registers (NOT in Memory)
- First 6 arguments:
  - %rdi
  - %rsi
  - %rdx
  - %rcx
  - %r8
  - %r9

- Diane’s Silk Dress Costs $89

- Return value: %rax

Stack (Memory)
- Only allocate stack space when needed
x86-64 Return Values

- By convention, values returned by procedures are placed in %rax
  - Choice of %rax is arbitrary

1) **Caller** must make sure to save the contents of %rax before calling a **callee** that returns a value
   - Part of register-saving convention

2) **Callee** places return value into %rax
   - Any type that can fit in 8 bytes – integer, float, pointer, etc.
   - For return values greater than 8 bytes, best to return a *pointer* to them

3) Upon return, **caller** finds the return value in %rax
Data Flow Examples

```c
void multstore
   (long x, long y, long *dest)
{
    long t = mult2(x, y);
    *dest = t;
}
```

```c
long mult2
   (long a, long b)
{
    long s = a * b;
    return s;
}
```

```assembly
00000000000400540 <multstore>:
   # x in %rdi, y in %rsi, dest in %rdx
   ... 
400541: movq %rdx,%rbx    # “Save” dest
400544: call 400550 <mult2> # mult2(x,y)
   # t in %rax
400549: movq %rax,(%rbx)  # Save at dest
   ... 

00000000000400550 <mult2>:
   # a in %rdi, b in %rsi
400550: movq %rdi,%rax    # a
400553: imulq %rsi,%rax   # a * b
   # s in %rax
400557: ret               # Return
```
Procedures

- Stack Structure
- **Calling Conventions**
  - Passing control
  - Passing data
  - Managing local data
- Register Saving Conventions
- Illustration of Recursion
Stack-Based Languages

- Languages that support recursion
  - *e.g.* C, Java, most modern languages
  - Code must be *re-entrant*
    - Multiple simultaneous instantiations of single procedure
  - Need some place to store *state* of each instantiation
    - Arguments, local variables, return pointer

- Stack allocated in *frames*
  - State for a single procedure instantiation

- Stack discipline
  - State for a given procedure needed for a limited time
    - Starting from when it is called to when it returns
  - Callee always returns before caller does
Call Chain Example

```
yoo (...) {
  •
  • who();
  •
}

who (...) {
  •
  amI();
  • amI();
  •
}

amI(...) {
  •
  if(...) {
    amI();
  }
  •
}
```

Procedure `amI` is recursive (calls itself)
1) Call to yoo

```c
yoo (...) {
  •
  •
  who();
  •
  •
}
```

Stack diagram:
- `main`?
- `yoo`
- `who`
- `aml`
- `%rbp`
- `%rsp`
2) Call to who

```
yoo(...) {
  who(...) {
    •
    amI();
    •
    amI();
    •
  }
}
```

Stack

```
yoo

who

%rbp

%rsp

“create” frame by manipulating %rsp
```
3) Call to amI (1)

```plaintext
yoo(...) {
  who(...) {
    amI(...) {
      .
      if() {
        amI()
      }
    }
    .
  }
}
```

Stack:
- yoo
- who
- amI
- amI
- amI
- amI
- %rbp
- %rsp
4) Recursive call to `amI` (2)
5) (another) Recursive call to `amI` (3)
6) Return from (another) recursive call to amI

yoo(...) {
  who(...) {
    amI(...) {
      if() {
        amI()
      }
    }
  }
}

Stack

yoo
who
amI \_1
amI \_2
amI \_3

"deallocate" stack frame by moving %rsp back up

data still exists, but you shouldn't use it
7) Return from recursive call to `amI`
8) Return from call to `amI`

```c
yoo(...)
{
  who(...)
  {
    amI();
    amI();
  }
}
```

Stack:
```
%rbp
yoo
%rsp
who
  amI
  amI
  amI
  amI
```

*New stack frame overwrites old data!*
9) (second) Call to `amI (4)`
10) Return from (second) call to `amI`
11) Return from call to who

```c
yoo (...) {
    //...
    who();
    //...
}
```

In total (counting main):
7 stack frames created
max depth of 6 stack frames

Stack

%rbp → yoo
%rsp →

Who

ami

ami

ami

ami

ami

ami

ami

ami

ami

ami
x86-64/Linux Stack Frame

- **Caller’s Stack Frame**
  - Extra arguments (if > 6 args) for this call

- **Current/Callee Stack Frame**
  - Return address
    - Pushed by `call` instruction
  - Old frame pointer (optional)
  - Saved register context
    (when reusing registers)
  - Local variables
    (If can’t be kept in registers)
  - “Argument build” area
    (If callee needs to call another function - parameters for function about to call, if needed)
Peer Instruction Question

Answer the following questions about when `main()` is run (assume `x` and `y` stored on the Stack):

```
int main() {
    int i, x = 0;
    for (i=0; i<3; i++)
        x = randSum(x);
    printf("x = %d\n",x);
    return 0;
}
```

```
int randSum(int n) {
    int y = rand()%20;
    return n+y;
}
```

- Higher/larger address: `x` or `y`?
- How many total stack frames are created? 8
- What is the maximum depth (# of frames) of the Stack?

A. 1  B. 2  C. 3  D. 4
Example: increment

```c
long increment(long *p, long val) {
    long x = *p;
    long y = x + val;
    *p = y;
    return x;
}
```

Increment:

```
movq (%rdi), %rax  # x = *p
addq %rax, %rsi    # y = x + val
movq %rsi, (%rdi)  # *p = y
ret
```

Register Use(s)

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>1st arg (p)</td>
</tr>
<tr>
<td>%rsi</td>
<td>2nd arg (val), y</td>
</tr>
<tr>
<td>%rax</td>
<td>x, return value</td>
</tr>
</tbody>
</table>
Procedure Call Example (initial state)

Return address on stack is the address of instruction immediately following the call to “call_incr”
- Shown here as `main`, but could be anything)
- Pushed onto stack by `call call_incr`
**Procedure Call Example (step 1)**

```c
long call_incr() {
    long v1 = 410;
    long v2 = increment(&v1, 100);
    return v1+v2;
}
```

**Call Incr:**
- `subq $16, %rsp`
- `movq $410, 8(%rsp)`
- `movl $100, %esi`
- `leaq 8(%rsp), %rdi`
- `call increment`
- `addq 8(%rsp), %rax`
- `addq $16, %rsp`
- `ret`

**Stack Structure**

![Stack Structure Diagram]

- `Allocate space for local vars`
  - Manually “push”
  - Setup space for local variables
    - Only `v1` needs space on the stack
  - Compiler allocated extra space
    - Often does this for a variety of reasons, including alignment
Procedure Call Example (step 2)

```c
long call_incr() {
    long v1 = 410;
    long v2 = increment(&v1, 100);
    return v1+v2;
}
```

**Aside:** `movl` is used because 100 is a small positive value that fits in 32 bits. High order bits of `rsi` get set to zero automatically. It takes one less byte to encode a `movl` than a `movq`.
Procedure Call Example (step 3)

```c
long call_incr() {  
    long v1 = 410;  
    long v2 = increment(&v1, 100);  
    return v1+v2;  
}
```

```c
long call_incr() {  
    long v1 = 410;  
    long v2 = increment(&v1, 100);  
    return v1+v2;  
}
```

- **State while inside increment**
  - **Return address** on top of stack is address of the addq instruction immediately following call to increment

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>&amp;v1</td>
</tr>
<tr>
<td>%rsi</td>
<td>100</td>
</tr>
<tr>
<td>%rax</td>
<td></td>
</tr>
</tbody>
</table>
Procedure Call Example (step 4)

```c
long call_incr() {
    long v1 = 410;
    long v2 = increment(&v1, 100);
    return v1 + v2;
}
```

Stack Structure

- Return addr <main+8>
- Return addr <call_incr+?>
- 510
- Unused

Increment:

1. movq (%rdi), %rax # x = *p
2. addq %rax, %rsi # y = x + 100
3. movq %rsi, (%rdi) # *p = y

State while inside `increment`

- After code in body has been executed

Register Use(s)

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>&amp;v1</td>
</tr>
<tr>
<td>%rsi</td>
<td>510</td>
</tr>
<tr>
<td>%rax</td>
<td>410</td>
</tr>
</tbody>
</table>
Procedure Call Example (step 5)

```c
long call_incr() {
    long v1 = 410;
    long v2 = increment(&v1, 100);
    return v1+v2;
}
```

- After returning from call to `increment`
  - Registers and memory have been modified and return address has been popped off stack

**Stack Structure**

- Return addr <main+8>
- `%rsp+8` 
- `%rsp`

**Stack Structure Diagram**

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>%rdi</code></td>
<td>&amp;v1</td>
</tr>
<tr>
<td><code>%rsi</code></td>
<td>510</td>
</tr>
<tr>
<td><code>%rax</code></td>
<td>410</td>
</tr>
</tbody>
</table>
Procedure Call Example (step 6)

```c
long call_incr() {
    long v1 = 410;
    long v2 = increment(&v1, 100);
    return v1+v2;
}
```

**Stack Structure**

- Return addr <main+8>
- %rsp+8
- 510
- Unused
- %rsp

- Update %rax to contain v1+v2

### Register Use(s)

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>&amp;v1</td>
</tr>
<tr>
<td>%rsi</td>
<td>510</td>
</tr>
<tr>
<td>%rax</td>
<td>510+410</td>
</tr>
</tbody>
</table>
Procedure Call Example (step 7)

```c
long call_incr() {
    long v1 = 410;
    long v2 = increment(&v1, 100);
    return v1+v2;
}
```

**Stack Structure**

- Return addr `<main+8>`
- 410
- Unused

### Register Use(s)

- `%rdi` &v1
- `%rsi` 510
- `%rax` 920

De-allocate space for local `vars` (make sure `%rsp` points to return addr before `ret`).
Procedure Call Example (step 8)

```c
long call_incr() {
    long v1 = 410;
    long v2 = increment(&v1, 100);
    return v1+v2;
}
```

- State *just before* returning from call to `call_incr`

- Stack Structure:
  - Return addr `<main+8>`
  - `%rsp`:
    - Popped off stack into `%rip` by `ret`

- Register Use(s):
  - `%rdi`: `&v1`
  - `%rsi`: `510`
  - `%rax`: `920`
Procedure Call Example (step 9)

```c
long call_incr() {
    long v1 = 410;
    long v2 = increment(&v1, 100);
    return v1 + v2;
}
```

- State immediately after returning from call to `call_incr`
  - Return addr has been popped off stack
  - Control has returned to the instruction immediately following the call to `call_incr` (not shown here)

### Final Stack Structure

- `%rsp` left

### Register Use(s)

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>%rdi</code></td>
<td>&amp;v1</td>
</tr>
<tr>
<td><code>%rsi</code></td>
<td>510</td>
</tr>
<tr>
<td><code>%rax</code></td>
<td>920</td>
</tr>
</tbody>
</table>
Lab 2 Demo

Let’s look at that binary bomb!

```bash
objdump -d bomb > bomb_disas  // store disassembly of bomb in file
called bomb_disas
```

In GDB:

- `stepi <#>`  // execute the next `<#>` asm instr (stepping into function calls)
- `nexti <#>`  // execute the next `<#>` asm instr (stepping over function calls)
- `print /<format> <expr>`  // print value of `<expr>` in `<format>`
- `x /<format> <addr>`  // dereference `<addr>` and print in `<format>`

Notes:

- Annoyingly, register names in `<expr>` and `<addr>` in GDB are preceded by `$`
  - so `$rsp` instead of `g0rsp`
- Common format characters are `‘b’` for binary
- `‘x’` for hex
- `‘s’` for string