Heartbeat could be used as a password to access electronic health records

Researches at Binghamton State University of New York have devised a new way to protect personal electronic health records using a patient's own heartbeat. Traditional security measures -- like cryptography or encryption -- can be expensive, time-consuming, and computing-intensive. Binghamton researchers encrypted patient data using a person's unique electrocardiograph (ECG) -- a measurement of the electrical activity of the heart measured by a biosensor attached to the skin -- as the key to lock and unlock the files. Since an ECG may change due to age, illness or injury -- or a patient may just want to change how their records are accessed -- researchers are currently working out ways to incorporate those variables.

- [https://www.sciencedaily.com/releases/2017/01/170118125240.htm](https://www.sciencedaily.com/releases/2017/01/170118125240.htm)
Administrivia

- Lab 1 due on Thursday (1/26)
- Homework 2 due next Tuesday (1/31)
- Optional Section this Thursday
  - Floating Point and GDB (The GNU Debugger)
Floating point topics

- Fractional binary numbers
- IEEE floating-point standard
- Floating-point operations and rounding
- **Floating-point in C**

- There are many more details that we won’t cover
  - It’s a 58-page standard...
Floating Point in C

- C offers two (well, 3) levels of precision

  - `float` 1.0f single precision (32-bit)
  - `double` 1.0 double precision (64-bit)
  - `long double` 1.0L (“double double” or quadruple) precision (64-128 bits)

- `#include <math.h>` to get INFINITY and NAN constants

- Equality (==) comparisons between floating point numbers are tricky, and often return unexpected results, so just avoid them!

- Instead use abs(f1 – f2) < 2^-20 or some other threshold
Floating Point Conversions in C

- **Casting between int, float, and double changes the bit representation**
  - **int → float**
    - May be rounded (not enough bits in mantissa: 23)
    - Overflow impossible
  - **int or float → double**
    - Exact conversion (all 32-bit ints representable)
  - **long → double**
    - Depends on word size (32-bit is exact, 64-bit may be rounded)
  - **double or float → int**
    - Truncates fractional part (rounded toward zero)
    - “Not defined” when out of range or NaN: generally sets to $T_{\min}$
      (even if the value is a very big positive)
#include <stdio.h>

int main(int argc, char* argv[]) {
    int a = 33554435;
    printf("a = %d\n(float) a = %f
\n\n", a, (float) a);

    float f1 = 1.0;
    float f2 = 0.0;
    int i;
    for (i = 0; i < 10; i++)
        f2 += 1.0/10.0;

    printf("0x%08x 0x%08x\n", *(int*)&f1, *(int*)&f2);
    printf("f1 = %10.9f\n", f1);
    printf("f2 = %10.9f\n\n", f2);

    f1 = 1E30;
    f2 = 1E-30;
    float f3 = f1 + f2;
    printf("f1 == f3? %s\n", f1 == f3 ? "yes" : "no");
    return 0;
}
Floating Point Summary

- Floats also suffer from the fixed number of bits available to represent them
  - Can get overflow/underflow
  - “Gaps” produced in representable numbers means we can lose precision, unlike ints
    - Some “simple fractions” have no exact representation (e.g. 0.2)
    - “Every operation gets a slightly wrong result”
- Floating point arithmetic not associative or distributive
  - Mathematically equivalent ways of writing an expression may compute different results
- Never test floating point values for equality!
- Careful when converting between ints and floats!
Number Representation Really Matters

- **1991**: Patriot missile targeting error
  - clock skew due to conversion from integer to floating point

- **1996**: Ariane 5 rocket exploded ($1 billion)
  - overflow converting 64-bit floating point to 16-bit integer

- **2000**: Y2K problem
  - limited (decimal) representation: overflow, wrap-around

- **2038**: Unix epoch rollover
  - Unix epoch = seconds since 12am, January 1, 1970
  - signed 32-bit integer representation rolls over to TMin in 2038

- **Other related bugs:**
  - 1982: Vancouver Stock Exchange 10% error in less than 2 years
  - 1994: Intel Pentium FDIV (floating point division) HW bug ($475 million)
  - 1997: USS Yorktown “smart” warship stranded: divide by zero
  - 1998: Mars Climate Orbiter crashed: unit mismatch ($193 million)
## Roadmap

### C:
```
car *c = malloc(sizeof(car));
c->miles = 100;
c->gals = 17;
float mpg = get_mpg(c);
free(c);
```

### Java:
```
Car c = new Car();
c.setMiles(100);
c.setGals(17);
float mpg =
c.getMPG();
```

### Assembly language:
```
get_mpg:
    pushq  %rbp
    movq   %rsp, %rbp
    ...
    popq  %rbp
    ret
```

### Machine code:
```
0111010000011000
100011010000010000000010
1000100111000010
11000001111110101000011111
```

### Computer system:
- x86 assembly
- Procedures & stacks
- Executables
- Arrays & structs
- Memory & caches
- Processes
- Virtual memory
- Operating Systems

### OS:
- Windows 8
- Mac
- Linux
Translation

What makes programs run fast(er)?

Code Time

User program in C

Compile Time

C compiler

Assembler

Run Time

Hardware

.c file

.exe file
HW Interface Affects Performance

**Source code**
- Different applications or algorithms

**Compiler**
- Perform optimizations, generate instructions

**Architecture**
- Instruction set

**Hardware**
- Different implementations
  - Intel Pentium 4
  - Intel Core 2
  - Intel Core i7
  - AMD Opteron
  - AMD Athlon
  - ARM Cortex-A53
  - Apple A7

---

**C Language**
- Program A
- Program B
- Your program

**Compiler**
- GCC
- Clang

**Architecture**
- x86-64
- ARMv8 (AArch64/A64)
Instruction Set Architectures

- The ISA defines:
  - The system’s state (e.g. registers, memory, program counter)
  - The instructions the CPU can execute
  - The effect that each of these instructions will have on the system state
**Instruction Set Philosophies**

- **Complex Instruction Set Computing (CISC):** Add more and more elaborate and specialized instructions as needed
  - Lots of tools for programmers to use, but hardware must be able to handle all instructions
  - x86-64 is CISC, but only a small subset of instructions encountered with Linux programs

- **Reduced Instruction Set Computing (RISC):** Keep instruction set small and regular
  - Easier to build fast hardware
  - Let software do the complicated operations by composing simpler ones
General ISA Design Decisions

- **Instructions**
  - What instructions are available? What do they do?
  - How are they encoded?

- **Registers**
  - How many registers are there?
  - How wide are they?

- **Memory**
  - How do you specify a memory location?
General ISA Design Decisions

- Instructions
  - What instructions are available? What do they do?
  - How are they encoded? Instructions are data!

- Registers
  - How many registers are there?
  - How wide are they? Size of a word

- Memory
  - How do you specify a memory location? Different ways to build up an address
# Mainstream ISAs

<table>
<thead>
<tr>
<th>Designer</th>
<th>Intel, AMD</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bits</td>
<td>16-bit, 32-bit and 64-bit</td>
</tr>
<tr>
<td>Introduced</td>
<td>1978 (16-bit), 1985 (32-bit), 2003 (64-bit)</td>
</tr>
<tr>
<td>Design</td>
<td>CISC</td>
</tr>
<tr>
<td>Type</td>
<td>Register-memory</td>
</tr>
<tr>
<td>Encoding</td>
<td>Variable (1 to 15 bytes)</td>
</tr>
<tr>
<td>Endianness</td>
<td>Little</td>
</tr>
</tbody>
</table>

## x86-64 Instruction Set

Macbooks & PCs
(Core i3, i5, i7, M)

## ARM architectures

<table>
<thead>
<tr>
<th>Designer</th>
<th>ARM Holdings</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bits</td>
<td>32-bit, 64-bit</td>
</tr>
<tr>
<td>Introduced</td>
<td>1985; 31 years ago</td>
</tr>
<tr>
<td>Design</td>
<td>RISC</td>
</tr>
<tr>
<td>Type</td>
<td>Register-Register</td>
</tr>
<tr>
<td>Encoding</td>
<td>AArch64/A64 and AArch32/A32 use 32-bit instructions, T32 (Thumb-2) uses mixed 16- and 32-bit instructions. ARMv7 user-space compatibility[1]</td>
</tr>
<tr>
<td>Endianness</td>
<td>Bi (little as default)</td>
</tr>
</tbody>
</table>

## Smartphone-like devices
(iPhone, iPad, Raspberry Pi)

ARM Instruction Set

## Digital home & networking equipment
(Blu-ray, PlayStation 2)

MIPS Instruction Set

<table>
<thead>
<tr>
<th>Designer</th>
<th>MIPS Technologies, Inc.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bits</td>
<td>64-bit (32→64)</td>
</tr>
<tr>
<td>Introduced</td>
<td>1981; 35 years ago</td>
</tr>
<tr>
<td>Design</td>
<td>RISC</td>
</tr>
<tr>
<td>Type</td>
<td>Register-Register</td>
</tr>
<tr>
<td>Encoding</td>
<td>Fixed</td>
</tr>
<tr>
<td>Endianness</td>
<td>Bi</td>
</tr>
</tbody>
</table>
Definitions

- **Architecture (ISA):** The parts of a processor design that one needs to understand to write assembly code
  - “What is directly visible to software”

- **Microarchitecture:** Implementation of the architecture
  - CSE/EE 469, 470

- Are the following part of the architecture?
  - Number of registers?
  - How about CPU frequency?
  - Cache size? Memory size?
Definitions

- **Architecture (ISA):** The parts of a processor design that one needs to understand to write assembly code
  - “What is directly visible to software”

- **Microarchitecture:** Implementation of the architecture
  - CSE/EE 469, 470

- Are the following part of the architecture?
  - Number of registers? Yes
  - How about CPU frequency? No
  - Cache size? Memory size? No
Assembly Programmer’s View

- **Programmer-visible state**
  - **PC**: the Program Counter ($\%\text{rip}$ in x86-64)
    - Address of next instruction
  - Named registers
    - Together in “register file”
    - Heavily used program data
  - Condition codes
    - Store status information about most recent arithmetic operation
    - Used for conditional branching

- **Memory**
  - Byte-addressable array
  - Code and user data
  - Includes the Stack (for supporting procedures)
x86-64 Assembly “Data Types”

- Integral data of 1, 2, 4, or 8 bytes
  - Data values
  - Addresses (untyped pointers)

- Floating point data of 4, 8, 10 or 2x8 or 4x4 or 8x2
  - Different registers for those (e.g. \%xmm1, \%ymm2)
  - Come from extensions to x86 (SSE, AVX, ...)

- No aggregate types such as arrays or structures
  - Just contiguously allocated bytes in memory

- Two common syntaxes
  - “AT&T”: used by our course, slides, textbook, gnu tools, ...
  - “Intel”: used by Intel documentation, Intel tools, ...
  - Must know which you’re reading

Not covered
In 410
What is a Register?

- A location in the CPU that stores a small amount of data, which can be accessed very quickly (once every clock cycle)

- Registers have *names*, not *addresses*
  - In assembly, they start with % (e.g. %rsi)

- Registers are at the heart of assembly programming
  - They are a precious commodity in all architectures, but *especially* x86
### x86-64 Integer Registers – 64 bits wide

<table>
<thead>
<tr>
<th>%rax</th>
<th>%eax</th>
<th>%r8</th>
<th>%r8d</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rbx</td>
<td>%ebx</td>
<td>%r9</td>
<td>%r9d</td>
</tr>
<tr>
<td>%rcx</td>
<td>%ecx</td>
<td>%r10</td>
<td>%r10d</td>
</tr>
<tr>
<td>%rdx</td>
<td>%edx</td>
<td>%r11</td>
<td>%r11d</td>
</tr>
<tr>
<td>%rsi</td>
<td>%esi</td>
<td>%r12</td>
<td>%r12d</td>
</tr>
<tr>
<td>%rdi</td>
<td>%edi</td>
<td>%r13</td>
<td>%r13d</td>
</tr>
<tr>
<td>%rsp</td>
<td>%esp</td>
<td>%r14</td>
<td>%r14d</td>
</tr>
<tr>
<td>%rbp</td>
<td>%ebp</td>
<td>%r15</td>
<td>%r15d</td>
</tr>
</tbody>
</table>

- Can reference low-order 4 bytes (also low-order 2 & 1 bytes)
Some History: IA32 Registers – 32 bits wide

<table>
<thead>
<tr>
<th>Register</th>
<th>Origin</th>
<th>Name Origin</th>
<th>General Purpose</th>
<th>16-bit virtual registers (backwards compatibility)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%eax</td>
<td>%ax</td>
<td>%ah %al</td>
<td>accumulate</td>
<td></td>
</tr>
<tr>
<td>%ecx</td>
<td>%cx</td>
<td>%ch %cl</td>
<td>counter</td>
<td></td>
</tr>
<tr>
<td>%edx</td>
<td>%dx</td>
<td>%dh %dl</td>
<td>data</td>
<td></td>
</tr>
<tr>
<td>%ebx</td>
<td>%bx</td>
<td>%bh %bl</td>
<td>base</td>
<td></td>
</tr>
<tr>
<td>%esi</td>
<td>%si</td>
<td></td>
<td>source index</td>
<td></td>
</tr>
<tr>
<td>%edi</td>
<td>%di</td>
<td></td>
<td>destination index</td>
<td></td>
</tr>
<tr>
<td>%esp</td>
<td>%sp</td>
<td></td>
<td>stack pointer</td>
<td></td>
</tr>
<tr>
<td>%ebp</td>
<td>%bp</td>
<td></td>
<td>base pointer</td>
<td></td>
</tr>
</tbody>
</table>

The IA32 registers are 32 bits wide, with certain 16-bit virtual registers for backwards compatibility.
Memory vs. Registers

- **Addresses**
  - 0x7FFFD024C3DC

- **Big**
  - ~8 GiB

- **Slow**
  - ~50-100 ns

- **Dynamic**
  - Can “grow” as needed while program runs

- **Addresses** vs. **Names**
  - %rdi

- **Big** vs. **Small**
  - (16 x 8 B) = 128 B

- **Slow** vs. **Fast**
  - sub-nanosecond timescale

- **Dynamic** vs. **Static**
  - fixed number in hardware
Three Basic Kinds of Instructions

1) Transfer data between memory and register
   - **Load** data from memory into register
     - \( \%\text{reg} = \text{Mem}[\text{address}] \)
   - **Store** register data into memory
     - \( \text{Mem}[\text{address}] = \%\text{reg} \)

2) Perform arithmetic operation on register or memory data
   - \( c = a + b; \quad z = x \ll y; \quad i = h \& g; \)

3) Control flow: what instruction to execute next
   - Unconditional jumps to/from procedures
   - Conditional branches

**Remember:** Memory is indexed just like an array of bytes!
Operand types

- **Immediate**: Constant integer data
  - Examples: $0x400, -533
  - Like C literal, but prefixed with ‘$’
  - Encoded with 1, 2, 4, or 8 bytes depending on the instruction

- **Register**: 1 of 16 integer registers
  - Examples: %rax, %r13
  - But %rsp reserved for special use
  - Others have special uses for particular instructions

- **Memory**: Consecutive bytes of memory at a computed address
  - Simplest example: (%rax)
  - Various other “address modes”
Moving Data

- **General form:** `mov_ source, destination`
  - Missing letter (_) specifies size of operands
  - Note that due to backwards-compatible support for 8086 programs (16-bit machines!), “word” means 16 bits = 2 bytes in x86 instruction names
  - Lots of these in typical code

- `movb src, dst`
  - Move 1-byte “byte”

- `movw src, dst`
  - Move 2-byte “word”

- `movl src, dst`
  - Move 4-byte “long word”

- `movq src, dst`
  - Move 8-byte “quad word”
# movq Operand Combinations

<table>
<thead>
<tr>
<th>Source</th>
<th>Dest</th>
<th>Src, Dest</th>
<th>C Analog</th>
</tr>
</thead>
<tbody>
<tr>
<td>Imm</td>
<td>Reg</td>
<td>movq $0x4, %rax</td>
<td>var_a = 0x4;</td>
</tr>
<tr>
<td>Mem</td>
<td>Reg</td>
<td>movq $-147, (%rax)</td>
<td>*p_a = -147;</td>
</tr>
<tr>
<td>Mem</td>
<td>Reg</td>
<td>movq %rax, %rdx</td>
<td>var_d = var_a;</td>
</tr>
<tr>
<td>Mem</td>
<td>Reg</td>
<td>movq (%rax), %rdx</td>
<td>var_d = *p_a;</td>
</tr>
</tbody>
</table>

- **Cannot do memory-memory transfer with a single instruction**
  - How would you do it?
Peer Instruction Question

Which of the following statements is TRUE?


A. For float f, (f+2 == f+1+1) always returns TRUE

B. The width of a “word” is part of a system’s architecture (as opposed to microarchitecture)

C. Having more registers increases the performance of the hardware, but decreases the performance of the software

D. Mem to Mem (src to dst) is the only disallowed operand combination in x86-64
Summary

- Converting between integral and floating point data types *does* change the bits
  - Floating point rounding is a HUGE issue!
    - Limited mantissa bits cause inaccurate representations
    - Floating point arithmetic is NOT associative or distributive

- x86-64 is a complex instruction set computing (CISC) architecture

- **Registers** are named locations in the CPU for holding and manipulating data
  - x86-64 uses 16 64-bit wide registers

- Assembly operands include immediates, registers, and data at specified memory locations