Status

- Project 3 back sometime in the middle of this week.
- Project 4 due this Friday, 11:55pm
Approach

• Engineer’s method:
  – DO UNTIL (exhausted)
  – tweak something
  – IF (better) THEN accept_change

• Scientific method:
  – DO UNTIL (enlightened)
  – make hypothesis
  – experiment
  – revise hypothesis
Power3’s power ... and limits

- Eight pipelined functional units
  - 2 floating point
  - 2 load/store
  - 2 single-cycle integer
  - 1 multi-cycle integer
  - 1 branch

- Powerful operations
  - Fused multiply-add (FMA)
  - Load (or Store) update

- Launch 4 ops per cycle
- Can’t launch 2 stores/cycle
- FMA pipe 3-4 cycles long
- Memory hierarchy (Tues)
Can its power be harnessed?

```c
for (j=0; j<n; j+=4){
p00 += a[j+0]*a[j+2];
m00 -= a[j+0]*a[j+2];
p01 += a[j+1]*a[j+3];
m01 -= a[j+1]*a[j+3];
p10 += a[j+0]*a[j+3];
m10 -= a[j+0]*a[j+3];
p11 += a[j+1]*a[j+2];
m11 -= a[j+1]*a[j+2];
}
8 FMA’s
4 Loads
```

Runs at 4.6 cycles/iteration (= 772 MFLOP/S)

---

**CL.6:**

- **FMA**  \(fp31=fp31,fp2,fp0,fcr\)
- **LFL**  \(fp1=(*)\)double(g3,16)
- **FNMS**  \(fp30=fp30,fp2,fp0,fcr\)
- **LFDU**  \(fp3,gr3=(*)\)double(g3,32)
- **FMA**  \(fp24=fp24,fp0,fp1,fcr\)
- **FNMS**  \(fp25=fp25,fp0,fp1,fcr\)
- **LFL**  \(fp0=(*)\)double(g3,24)
- **FMA**  \(fp27=fp27,fp2,fp3,fcr\)
- **FNMS**  \(fp26=fp26,fp2,fp3,fcr\)
- **LFL**  \(fp2=(*)\)double(g3,8)
- **FMA**  \(fp29=fp29,fp1,fp3,fcr\)
- **FNMS**  \(fp28=fp28,fp1,fp3,fcr\)
- **BCT**  \(ctr=CL.6,\)
Can its power be harnessed (part II)

- 8 FMA, 4 Load - 1.15 cycle/load (previous slide)
- 8 FMA, 6 Load - 1.3 cycle/load
- 8 FMA, 8 Load - 1.2 cycle/load
- 4 Add, 4 Load - 1.1 cycle/load
- Shift, Add, Load, Store - 1.15 cycle/MemOp
- Load, Store - 1.1 cycle/MemOp

---

- I haven’t broken the 1 cycle/MemOp barrier!
- but I’ve only spent 2 days trying …maybe the AGEN unit is disabled...
FLOP to MemOp ratio

- Most scientific programs have at most one FMA per MemOp
- Matrix-vector product: \((K+1)\) loads, \(K\) fma’s
- FFT butterfly: 8 MemOps, 10 floats (but 5 or 6 FMA)
- DAXPY: 2 Loads, 1 Store, 1 FMA
- DDOT: 2 Loads, 1 FMA
- A few have more (use ESSL!)
- Matrix multiply (well-tuned): 2 FMA per load
- Radix-8 FFT

- Performance is limited by Memory Operations!
The effect of pipeline latency

for (i=0; i<size; i++) {
    sum = a[i] + sum;
}

Next add can’t start until previous is finished (3 to 4 cycles later)

for (i=0; i<size; i+=4) {
    sum0 += a[i];
    sum1 += a[i+1];
    sum2 += a[i+2];
    sum3 += a[i+3];
}

sum = sum0+sum1+sum2+sum3;

3.86 cycles/addition

1.1 cycles/addition
The effect of pipeline latency

for (i=0; i<size; i++) {
    sum = a[i] + sum;
}

Next add can’t start until previous is finished (3 to 4 cycles later)

for (i=0; i<size; i+=4) {
    sum0 += a[i];
    sum1 += a[i+1];
    sum2 += a[i+2];
    sum3 += a[i+3];
}

sum = sum0+sum1+sum2+sum3;

May change answer due to different rounding.
What’s so great about Fortran??

DO I = 1, N
    A(I) = B(I)
ENDDO

for (i=0; i<N; i++) {
    b[i] = a[i];
}

Wednesday, June 3, 2009
Fortran vs C - what’s going on??

- C prevents compiler from unrolling code
- A feature, not a bug!
- User may want $b[0]$ and $a[1]$ to be same location
- tricky way to set $a[n] = ... = a[1] = a[0]$
- Most C compilers don’t try to prove non-aliasing
- $a$ and $b$ were `malloc`-ed in this example
- Fortran doesn’t allow arrays to be aliased
- Unless explicit, e.g. via `EQUIVALENCE`
Fortran vs. C - does it matter??

- Yes - Fortran code *should* perform better
  - My tests show both are about 1 cycle/MemOp
  - Fortran *should* be .5 cycle/MemOp
- No - you could get the “Fortran” object code from

```c
for (i=0; i<N; i+=4) {
    b0 = a[i];
    b1 = a[i+1];
    b2 = a[i+2];
    b3 = a[i+3];
    b[i] = b0;
    b[i+1] = b1;
    b[i+2] = b2;
    b[i+3] = b3;
}
```
Decreasing MemOp to FLOP Ratio

for (i=1; i<N; i++)
    for (j=1; j<N; j++)
        b[i,j] = 0.25 * (a[i-1][j] + a[i+1][j] + a[i,j-1] + a[i][j-1]);

for (i=1; i<N-2; i+=3) {
    for(j=1; j<N; j++) {
        b[i+0][j] = ... ;
        b[i+1][j] = ... ;
        b[i+2][j] = ... ;
    }
}

3 loads
4 floats
1 store

5 loads
12 floats
3 store
Compilers: Topics Not Covered

- Lambda Calculus
- Foundation for Programming Languages
- Instruction Scheduling
- Compiling Exceptions
A Few Thoughts on Compilers and Security
Vulnerable Program?

```c
int main(int argc, char *argv[])
{
    char buffer[500];
    strcpy(buffer, argv[1]);
    printf("Safe program?\n");
    return 0;
}
```
Shellcode

\x31\xc0\x50\x68\x2\x2\x73\x68\x68\x2\x62\x69\x6\xe3\x50\x53\xb0\x3\x50\xcd\x80

Wednesday, June 3, 2009
Buffer Overflows
Impact of Buffer Overflows

• Buffer overflows (including those on the stack, the heap, and also integer overflows) have dominated exploitable programming flaws for years. (http://www.sans.org/top20/)

• E.g., worms: Blaster, Morris, Slammer, Witty, etc....

• It would be great if programmers just wrote better code and did their own bounds checking. But in practice, even good coders make mistakes.
• C/C++ allows overwriting memory arbitrarily via pointer accesses.

• C/C++ are the root of a lot of evil (but not all) in computer security
Countermeasures

- Compilers and compiler extensions can perform some countermeasures via static analysis and code modification at compile time.

- Binary rewriters can also do this statically.

- Much of what compilers do is look for certain invariants: conditions that should hold and sometimes do not.
Static Analysis

- Some things are easy at compile time:
  - `strcpy` → `strncpy`
  - `strcat` → `strncat`

- Some things are easy at runtime, e.g., bounds checking:
  - for `strcpy(dest, src)`, how large is the space allocated for `dest` vs. the length of `src`?
  - (strncpy does this anyhow)
  - Java also does this
Safe Functions/Libraries

• Some C Functions Have “Safe” Equivalents
  • strcat/strncat
  • strcpy/strncpy
  • sprintf/snprintf

• Some functions cannot be used safely:
  • fgets
  • (the Morris worm exploited this in 1988)
void func() {
    char buf[100], buf2[100], buf3[200];
    /* do stuff */
}

void func_mod() {
    char buf[100], buf2[100], buf3[200];
    __stack_buf(buf1, sizeof(buf1));
    __stack_buf(buf2, sizeof(buf2));
    __stack_buf(buf3, sizeof(buf3));
    /* do stuff */
}
StackGuard and Others

- Checks the to see if the stack has been altered when a function returns.
- A “canary” value is stored where the return address usually is stored (and maybe other places too).
  - Sometimes the return address it put elsewhere
  - Sometimes the return address is just stored XORed
- When the function returns it checks to see if the canary has changed.
- Can’t protect against buffer overflows in the heap.
• XOR-ing pointers is used in other places too, and doesn’t have to refer only to return addresses.

• Can XOR all pointers.

• Microsoft worked on this for Windows at one point using a compiler extension.
Address Space Layout and Direction

• The stack can run up, down, or both.

• Or, address areas (stack, heap, etc...) can be placed arranged randomly in a process’s address space.

• Larger memory space = more entropy = harder to attack
Non-Executable Stack

• Some operating systems prevent execution (e.g., shellcode that says, “exec /bin/sh”) on the stack.
Bounds Checking

// C allows you do do this
int main(int argc, char *argv[]) {
    char myArray[500];
    printf("ThisValue: %d",
           (int)myArray[argv[1]]);
    return 0;
}
// Maybe compilers should do this?
int main(int argc, char *argv[]) {
    char myArray[500];
    if ((int)argv[1] < 500) {
        printf("ThisValue: %d",
                (int)myArray[argv[1]]);
    }
    else { ... }
    return 0;
}
// C allows you do do this
int main(int argc, char *argv[]) {
    short a = 32767;
    int i;
    for (i = 0; i < atoi(argv[1]); i++)
        a++;
    if (a > 0)
        ...
}
// 32677, -32768, -32767, -32766...
if (access(filename, W_OK) == 0) {
    if ((fd = open(filename, O_WRONLY)) == NULL){
        perror(filename);
        return(0);
    }
    /* now write to the file */
}
The scenario in Figure 1 is an example of the TOCTTOU binding flaw. Figure 1a shows the state of the system at the time of the access system call; the solid arrow indicates the access refers to "/tmp/X". Both "/tmp/X" and "/etc/passwd" name distinct objects. However, before the process makes its open system call, "/tmp/X" is deleted and a direct alias (hard link) for "/etc/passwd" is created, and is named "/tmp/X". Then the open accesses the data associated with "/etc/passwd" when it opens "/tmp/X", since "/tmp/X" and "/etc/passwd" now refer to the same file. Figure 1b shows this, with the dashed arrow indicating which data is actually read and the solid arrow indicating the name given to open. The unprivileged process can then write to the protected password file. Several versions of the terminal emulation program xterm (1) [16] suffer from this flaw, which arises when logging sessions to a file. Another instance of this flaw occurs on SunOS and HP/UX systems. The program passwd (1) allows the user to name the password file as a parameter. An attacker can gain access to any other user's accounts using a variant of the attack presented above [6]. Under normal conditions, the passwd program takes the following steps:

1. opens and reads the password file to get the entry for the user; then closes the password file;
2. creates and opens a temporary file called "ptmp" in the directory of the password file;
3. opens the password file again, and copies the contents to "ptmp", updating the changed information; and
4. closes the password file and "ptmp" and renames "ptmp" to be the password file.

Figure 1. Example of the TOCTTOU binding flaw.
int main() {
    // start with root privileges
    do something with privilege();
    exec ("/bin/sh/");
}

vs.

int main() {
    // start with root privileges
    do something with privilege();
    // drop privileges
    exec ("/bin/sh/");
}
Privilege Escalation

```
// The program has root privilege
if ((passwd = getpwuid(getuid())) != NULL)
{
    fprintf(log, “drop priv for %s”, passwd->pw_name);
    setuid(getuid()); // drop privilege
}
execl(“/bin/sh”, “/bin/sh”, NULL); // risky syscall
```
TOCTTOU FSA Property

Figure 3: An FSA illustrating Property 3 (but the path from the model checking it can take advantage of).

Figure 4: A program where the security property is satisfied in the MOPS model checking framework.

The TOCTTOU FSA Property is illustrated in the diagram, showing the transitions and states that represent the security property.

The figure includes a state machine with states labeled 'unpriv noexec', 'priv noexec', 'unpriv exec', and 'priv exec'. The transitions between these states are labeled with system calls such as 'execl()', 'seteuid(0)', and 'seteuid(!0)'.
Other Race Conditions

int a; // global

main() {
    spawn_threads();
}

thread_run() {
    if(thread_num % 2 == 0)
        a++;
    ...
}
Other Race Conditions

int a; // global

main() {
    spawn_threads();
}

thread_run() {
    if(thread_num % 2 == 0)
        lock();
    a++;
    unlock();
    ...
}
There are a lot of elements to security

- Good system design (technical and procedural)
- Good coding practice
- Good tools, including *compilers* (static analysis and instrumentation) and OS/virtual environments (runtime)
- Good *languages* that have fewer tradeoffs between security vs. usability vs. performance