Researchers at the University of Illinois, in collaboration with Tufts University and Northwestern University, have demonstrated a new type of biodegradable electronics technology that could introduce new design paradigms for medical implants, environmental monitors and consumer devices.

Three application areas appear particularly promising. First are medical implants that perform important diagnostic or therapeutic functions for a useful amount of time and then simply dissolve and resorb in the body. Second are environmental monitors, such as wireless sensors that are dispersed after a chemical spill, that degrade over time to eliminate any ecological impact. Third are consumer electronic systems or sub-components that are compostable, to reduce electronic waste streams generated by devices that are frequently upgraded, such as cellphones or other portable devices.
• Questions about lab?
```c
#include <stdio.h>

int globalvar;

int func1(int *x);
int func2(int x, int *y);
void func3_hint_seepage215(int x);

main() {
  int mainvar = 0;
  int ret = 0;

  globalvar = 364;

  printf("%d\t%d\t%d\n", mainvar, ret, globalvar);

  ret = func1(&mainvar);
  printf("%d\t%d\t%d\n", mainvar, ret, globalvar);

  ret = func2(mainvar, &globalvar);
  printf("%d\t%d\t%d\n", mainvar, ret, globalvar);

  func3_hint_seepage215(ret);
}
```
int func1(int *x) {
    *x = 1 + 364;
    return 0;
}

int func2(int x, int *y) {
    return x * *y;
}

void func3_hint_seepage215(int x) {
    int val = 2;
    switch(x) {
        case 1: {
            val = x + 1;
            break;
        }
        case 2: {
            val = x / 10;
            break;
        }
        case 3: {
            val = x * 10;
            break;
        }
        case 4: {
            val = x % 10;
            break;
        }
        case 128: {
            val = 3;
            break;
        }
        default: {val = 0; }
    }
    printf("Something %d\n", val);
}
“For” Loop $\rightarrow$ ... $\rightarrow$ Goto

For Version

for \((\text{Init}; \text{Test}; \text{Update})\)

Body

While Version

\text{Init};

while \((\text{Test})\) {

Body

\text{Update};

}

\text{Init};

\text{if} \ (!\text{Test})

goto \text{done};

\text{loop:}

\text{Body}

\text{Update}

\text{if} \ (\text{Test})

goto \text{loop};

done:
IA 32 Procedure Summary

• The Stack Makes Recursion Work
  – Private storage for each *instance* of procedure call
    • Instantiations don’t clobber each other
    • Addressing of locals + arguments can be relative to stack positions
  – Managed by stack discipline
    • Procedures return in inverse order of calls

• IA32 Procedures Combination of Instructions + Conventions
  – Call / Ret instructions
  – Register usage conventions
    • Caller / Callee save
    • %ebp and %esp
  – Stack frame organization conventions
IA32/Linux Register Usage

- **%eax, %edx, %ecx**
  - Caller saves prior to call if values are used later

- **%eax**
  - also used to return integer value

- **%ebx, %esi, %edi**
  - Callee saves if wants to use them

- **%esp, %ebp**
  - special
A C function \texttt{fun} has the following code body:

\begin{verbatim}
    *p = d;
    return x-c;
\end{verbatim}

The IA32 code implementing this body is as follows:

\begin{verbatim}
    movsbl 12(%ebp),%edx
    movl 16(%ebp), %eax
    movl %edx, (%eax)
    movswl 8(%ebp),%eax
    movl 20(%ebp), %edx
    subl %eax, %edx
    movl %edx, %eax
\end{verbatim}

Write a prototype for function \texttt{fun}, showing the types and ordering of the arguments \texttt{p}, \texttt{d}, \texttt{x}, and \texttt{c}. 
One step in learning to read IA32 code is to become very familiar with the way arguments are passed on the stack. The key to solving this problem is to note that the storage of d at p is implemented by the instruction at line 3 of the assembly code, from which you work backward to determine the types and positions of arguments d and p. Similarly, the subtraction is performed at line 6, and from this you can work backward to determine the types and positions of arguments x and c.

The following is the function prototype:

```
int fun(short c, char d, int *p, int x);
```

As this example shows, reverse engineering is like solving a puzzle. It’s important to identify the points where there is a unique choice, and then work around these points to fill in the rest of the details.
```
int proc(void)
{
  int x,y;
  scanf("%x %x", &y, &x);
  return x-y;
}
```

Suppose proc calls scanf (line 10), and that scanf reads values 0x46 and 0x53 from the standard input. Assume that the string “%x %x” is stored at memory location 0x300070.

A. What value does %ebp get set to on line 3?
B. What value does %esp get set to on line 4?
C. At what addresses are local variables x and y stored?
D. Draw a diagram of the stack frame for proc right after scanf returns. Include as much information as you can about the addresses and the contents of the stack frame elements.
E. Indicate the regions of the stack frame that are not used by proc.
A. We started with %esp having value 0x800040. The pushl instruction on line 2 decrements the stack pointer by 4, giving 0x80003C, and this becomes the new value of %ebp.

B. Line 4 decrements the stack pointer by 40 (hex 0x28), yielding 0x800014.

C. We can see how the two leal instructions (lines 5 and 7) compute the arguments to pass to scanf, while the two movl instructions (lines 6 and 8) store them on the stack. Since the function arguments appear on the stack at increasingly positive offsets from %esp, we can conclude that line 5 computes &x, while line 7 computes line &y. These have values 0x800038 and 0x800034, respectively.

D. The stack frame has the following structure and contents:

```
<table>
<thead>
<tr>
<th>Offset</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x80003C</td>
<td>0x800060</td>
</tr>
<tr>
<td>0x800038</td>
<td>0x53</td>
</tr>
<tr>
<td>0x800034</td>
<td>0x46</td>
</tr>
<tr>
<td>0x800030</td>
<td></td>
</tr>
<tr>
<td>0x80002C</td>
<td></td>
</tr>
<tr>
<td>0x800028</td>
<td></td>
</tr>
<tr>
<td>0x800024</td>
<td></td>
</tr>
<tr>
<td>0x800020</td>
<td></td>
</tr>
<tr>
<td>0x80001C</td>
<td>0x800038</td>
</tr>
<tr>
<td>0x800018</td>
<td>0x800034</td>
</tr>
<tr>
<td>0x800014</td>
<td>0x300070</td>
</tr>
</tbody>
</table>
```

E. Byte addresses 0x800020 through 0x800033 are unused.
Basic Data Types

• Integral
  – Stored & operated on in general (integer) registers
  – Signed vs. unsigned depends on instructions used

<table>
<thead>
<tr>
<th>Intel</th>
<th>suffix</th>
<th>Bytes</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>byte</td>
<td>b</td>
<td>1</td>
<td>[unsigned] char</td>
</tr>
<tr>
<td>word</td>
<td>w</td>
<td>2</td>
<td>[unsigned] short</td>
</tr>
<tr>
<td>double word</td>
<td>l</td>
<td>4</td>
<td>[unsigned] int</td>
</tr>
<tr>
<td>quad word</td>
<td>q</td>
<td>8</td>
<td>[unsigned] long int (x86-64)</td>
</tr>
</tbody>
</table>

• Floating Point
  – Stored & operated on in floating point registers

<table>
<thead>
<tr>
<th>Intel</th>
<th>suffix</th>
<th>Bytes</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single</td>
<td>s</td>
<td>4</td>
<td>float</td>
</tr>
<tr>
<td>Double</td>
<td>l</td>
<td>8</td>
<td>double</td>
</tr>
<tr>
<td>Extended</td>
<td>t</td>
<td>10/12/16</td>
<td>long double</td>
</tr>
</tbody>
</table>
Array Allocation

• Basic Principle
  
  ```
  T A[L];
  ```
  
  – Array of data type \( T \) and length \( L \)
  
  – Contiguously allocated region of \( L \times \text{sizeof}(T) \) bytes

```
char string[12];
```

```
int val[5];
```

```
double a[3];
```

```
char *p[3];
```

IA32

x86-64
Array Access

- Basic Principle
  \[ T \, A[L] ; \]
  - Array of data type \( T \) and length \( L \)
  - Identifier \( A \) can be used as a pointer to array element 0: Type \( T^* \)

```
int val[5];
```

- Reference Type Value

<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>val</td>
<td></td>
</tr>
<tr>
<td>val+1</td>
<td></td>
</tr>
<tr>
<td>&amp;val[2]</td>
<td></td>
</tr>
<tr>
<td>val[5]</td>
<td></td>
</tr>
<tr>
<td>*(val+1)</td>
<td></td>
</tr>
<tr>
<td>val + i</td>
<td></td>
</tr>
</tbody>
</table>
Array Access

• Basic Principle
  \( T \ A[L] \);
  – Array of data type \( T \) and length \( L \)
  – Identifier \( A \) can be used as a pointer to array element 0: Type \( T^* \)

\[
\begin{array}{c}
\text{int val[5];} \\
\hline
\text{x} & \text{x + 4} & \text{x + 8} & \text{x + 12} & \text{x + 16} & \text{x + 20} \\
\end{array}
\]

• Reference Type
<table>
<thead>
<tr>
<th>Reference</th>
<th>Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>val[4]</td>
<td>int</td>
<td>3</td>
</tr>
<tr>
<td>val</td>
<td>int *</td>
<td>( x )</td>
</tr>
<tr>
<td>val+1</td>
<td>int *</td>
<td>( x + 4 )</td>
</tr>
<tr>
<td>&amp;val[2]</td>
<td>int *</td>
<td>( x + 8 )</td>
</tr>
<tr>
<td>val[5]</td>
<td>int</td>
<td>??</td>
</tr>
<tr>
<td>*(val+1)</td>
<td>int</td>
<td>5</td>
</tr>
<tr>
<td>val + i</td>
<td>int *</td>
<td>( x + 4i )</td>
</tr>
</tbody>
</table>
Array Example

```c
int a[5][5];

typedef int zip_dig[5];
zip_dig b[5];  //equivalent to a
```
Array Example

typedef int zip_dig[5];

zip_dig cmu = { 1, 5, 2, 1, 3 };
zip_dig mit = { 0, 2, 1, 3, 9 };
zip_dig ucb = { 9, 4, 7, 2, 0 };

• Declaration “zip_dig cmu” equivalent to “int cmu[5]”
• Example arrays were allocated in successive 20 byte blocks
  – Not guaranteed to happen in general
Array Accessing Example

```c
int get_digit(zip_dig z, int dig) {
    return z[dig];
}
```

IA32

```assembly
# %edx = z
# %eax = dig
movl (%edx,%eax,4),%eax  # z[dig]
```

- Register %edx contains starting address of array
- Register %eax contains array index
- Desired digit at $4 \times %eax + %edx$
- Use memory reference $(%edx, %eax, 4)$
int E[10]; //an array of integers
int i; //index into array

E is in %edx, i is in %ecx
Move something into %eax

E      int *      movl %edx, %eax
E[0]   ?          movl ???, %eax
E[i]   ?          movl ???, %eax
&E[2]  ?          leal ???, %eax
&E[i]-E ?          movl ???, %eax
int E[10]; //an array of integers
int i; //index into array

E is in %edx, i is in %ecx
Move something into %eax

E    int *    movl %edx, %eax
E[0] int      movl (%edx), %eax
E[i] int      movl (%edx, %ecx, 4), %eax
&E[2] int*    leal 8(%edx), %eax
&E[i]-E int    movl %ecx, %eax
# Referencing Examples

<table>
<thead>
<tr>
<th>Reference</th>
<th>Address</th>
<th>Value</th>
<th>Guaranteed?</th>
</tr>
</thead>
<tbody>
<tr>
<td>mit[3]</td>
<td>36 + 4* 3 = 48</td>
<td>3</td>
<td>Yes</td>
</tr>
<tr>
<td>mit[5]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mit[-1]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cmu[15]</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Referencing Examples

### Example 1
```
zip_dig cmu;
```
```
+---+---+---+---+---+---+---+
| 1 | 5 | 2 | 1 | 3 | 9 |
+---+---+---+---+---+---+
| 16| 20| 24| 28| 32| 36|
```

### Example 2
```
zip_dig mit;
```
```
+---+---+---+---+---+---+---+
| 0 | 2 | 1 | 3 | 9 |
+---+---+---+---+---+---+
| 36| 40| 44| 48| 52| 56|
```

### Example 3
```
zip_dig ucb;
```
```
+---+---+---+---+---+---+---+
| 9 | 4 | 7 | 2 | 0 |
+---+---+---+---+---+---+
| 56| 60| 64| 68| 72| 76|
```

### Reference Examples

<table>
<thead>
<tr>
<th>Reference</th>
<th>Address</th>
<th>Value</th>
<th>Guaranteed?</th>
</tr>
</thead>
<tbody>
<tr>
<td>mit[3]</td>
<td>36 + 4*3 = 48</td>
<td>3</td>
<td>Yes</td>
</tr>
<tr>
<td>mit[5]</td>
<td>36 + 4*5 = 56</td>
<td>9</td>
<td>No</td>
</tr>
<tr>
<td>mit[-1]</td>
<td>36 + 4*-1 = 32</td>
<td>3</td>
<td>No</td>
</tr>
<tr>
<td>cmu[15]</td>
<td>16 + 4*15 = 76</td>
<td>??</td>
<td>No</td>
</tr>
</tbody>
</table>

- No bound checking
- Out of range behavior implementation-dependent
- No guaranteed relative allocation of different arrays
Intermission

http://cheezburger.com/42481409
Array Loop Example

Original

```c
int zd2int(zip_dig z)
{
    int i;
    int zi = 0;
    for (i = 0; i < 5; i++) {
        zi = 10 * zi + z[i];
    }
    return zi;
}
```

Transformed

- As generated by GCC
- Eliminate loop variable i
- Convert array code to pointer code
- Express in do-while form (no test at entrance)

```c
int zd2int(zip_dig z)
{
    int zi = 0;
    int *zend = z + 4;
    do {
        zi = 10 * zi + *z;
        z++;
    } while (z <= zend);
    return zi;
}
```
Array Loop Implementation (IA32)

- Registers
  
  \%ecx  \texttt{z} \\
  \%eax  \texttt{zi} \\
  \%ebx  \texttt{zend}

```c
int zd2int(zip_dig z)
{
    int zi = 0;
    int *zend = z + 4;
    do {
        zi = 10 * zi + *z;
        z++;
    } while(z <= zend);
    return zi;
}
```

```assembly
# %ecx = z
xorl %eax,%eax
leal 16(%ecx),%ebx
.L59:
    leal (%eax,%eax,4),%edx
    movl (%ecx),%eax
    addl $4,%ecx
    leal (%eax,%edx,2),%eax
    cmpl %ebx,%ecx
    jle .L59
```
Array Loop Implementation (IA32)

- Registers
  %ecx  z
  %eax  zi
  %ebx  zend

- Computations
  - \(10 \times zi + *z\) implemented as
    \(*z + 2 \times (zi + 4 \times zi)\)
  - \(z++\) increments by 4

```c
int zd2int(zip_dig z)
{
    int zi = 0;
    int *zend = z + 4;
    do {
        zi = 10 * zi + *z;
        z++;
    } while(z <= zend);
    return zi;
}
```

# %ecx = z
xorl %eax,%eax  # zi = 0
leal 16(%ecx),%ebx  # zend = z+4

.L59:
leal (%eax,%eax,4),%edx  # 5*zi
movl (%ecx),%eax  # *z
addl $4,%ecx  # z++
leal (%eax,%edx,2),%eax  # zi = *z + 2*(5*zi)
cmpl %ebx,%ecx  # z : zend
jle .L59  # if <= goto loop
Using Nested Arrays

• Strengths
  – C compiler handles doubly subscripted arrays
  – Generates very efficient code

• Limitation
  – Only works for fixed array size

```c
#define N 16
typedef int fix_matrix[N][N];

/* Compute element i,k of fixed matrix product */
int fix_prod_ele
(fix_matrix a, fix_matrix b, int i, int k)
{
    int j;
    int result = 0;
    for (j = 0; j < N; j++)
        result += a[i][j]*b[j][k];
    return result;
}
```

i-th row
j-th column
(a) Original C code

```c
1 /* Compute i,k of fixed matrix product */
2 int fix_prod_ele (fix_matrix A, fix_matrix B, int i, int k) {
3     int j;
4     int result = 0;
5
6     for (j = 0; j < N; j++)
7         result += A[i][j] * B[j][k];
8
9     return result;
10 }
```

(b) Optimized C code

```c
1 /* Compute i,k of fixed matrix product */
2 int fix_prod_ele_opt(fix_matrix A, fix_matrix B, int i, int k) {
3     int *Arow = &A[i][0];
4     int *Bptr = &B[0][k];
5     int result = 0;
6     int j;
7     for (j = 0; j != N; j++) {
8         result += Arow[j] * *Bptr;
9         Bptr += N;
10     }
11     return result;
12 }
```
Dynamic Nested Arrays

- **Strength**
  - Can create matrix of any size

- **Programming**
  - Must do index computation explicitly

- **Performance**
  - Accessing single element costly
  - Must do multiplication

```c
int * new_var_matrix(int n)
{
    return (int *)
        calloc(sizeof(int), n*n);
}
```

```c
int var_ele
    (int *a, int i, int j, int n)
{
    return a[i*n+j];
}
```

```assembly
movl 12(%ebp),%eax       # i
movl 8(%ebp),%edx        # a
imull 20(%ebp),%eax      # n*i
addl 16(%ebp),%eax       # n*i+j
movl (%edx,%eax,4),%eax  # Mem[a+4*(i*n+j)]
```
Dynamic Array Multiplication

- Without Optimizations
  - Multiplies: 3
    - 2 for subscripts
    - 1 for data
  - Adds: 4
    - 2 for array indexing
    - 1 for loop index
    - 1 for data

```c
/* Compute element i,k of variable matrix product */
int var_prod_ele
(int *a, int *b,
 int i, int k, int n)
{
    int j;
    int result = 0;
    for (j = 0; j < n; j++)
        result +=
            a[i*n+j] * b[j*n+k];
    return result;
}
```
Optimizing Dynamic Array Multiplication

- **Optimizations**
  - Performed when set optimization level to `-O2`

- **Code Motion**
  - Expression `i*n` can be computed outside loop

- **Strength Reduction**
  - Incrementing `j` has effect of incrementing `j*n+k` by `n`

- **Operations count**
  - 4 adds, 1 mult

- **Compiler can optimize regular access patterns**

```c
{  
    int j;
    int result = 0;
    for (j = 0; j < n; j++)
        result +=
            a[i*n+j] * b[j*n+k];
    return result;
}
```

```c
{  
    int j;
    int result = 0;
    int iTn = i*n;
    int jTnPk = k;
    for (j = 0; j < n; j++)  
    {  
        result +=
            a[iTn+j] * b[jTnPk];
        jTnPk += n;
    }  
    return result;
}
```
Structures

```c
struct rec {
    int i;
    int a[3];
    int *p;
};
```

### Memory Layout

![Memory Layout](image)

### Concept
- Contiguously-allocated region of memory
- Refer to members within structure by names
- Members may be of different types

### Accessing Structure Member

```c
void set_i(struct rec *r, int val)
{
    r->i = val;
}
```

### IA32 Assembly

```
# %eax = val
# %edx = r
movl %eax, (%edx)    # Mem[r] = val
```
Generating Pointer to Structure Member

```c
struct rec {
    int i;
    int a[3];
    int *p;
};
```

- Generating Pointer to Array Element
  - Offset of each structure member determined at compile time

```c
int *find_a
    (struct rec *r, int idx)
{
    return &r->a[idx];
}
```

```assembly
# %ecx = idx
# %edx = r
leal 0(,%ecx,4),%eax     # 4*idx
leal 4(%eax,%edx),%eax   # r+4*idx+4
```