Lecture 9: Low-Level Programming

Problem Classes if P ≠ NP:

- NP: How many problems are in the Θ(n log n) class?
  - infinite
- NP-Complete: How many problems are in P but not in the Θ(n log n) class?
  - infinite
- Subset Sum: How many problems are in NP but not in P?
  - infinite

Is it ever *useful* to be confident that a problem is hard?

Knapsack Cipher [Merkle & Hellman, 1978]

- Public Key: \( A = \{a_1, a_2, \ldots, a_n\} \)
  - Set of integers
- Plain Text: \( x_1, \ldots, x_n \)
  - \( x_i = 0 \) or \( 1 \)
- Cipher Text:
  \[
  s = \sum_{i=1}^{n} x_ia_i
  \]

Subset Sum is Hard

- Given \( s \) and \( A \) it is NP-Complete to find a subset of \( A \) that sums to \( s \)

- Need to make decrypting each (for recipient with the "private key")
Superincreasing Set

- Pick \( \{a_1, a_2, \ldots, a_n\} \) is a superincreasing sequence

\[
a_i > \sum_{j=1}^{i-1} a_j
\]

How hard is subset sum if \( A \) is superincreasing?

Knapsack Ciphers

- Private Key = \( \{p_1, p_2, \ldots, p_n\} \)
  - A superincreasing sequence
  - Values \( M \) and \( W \):
    \[ M > \sum b_i \]
    \[ GCD(M, W) = 1 \]

- Public Key = \( \{a_1, a_2, \ldots, a_n\} \)

\[
a_i \equiv (b_i W) \mod M
\]

Flawed Security Argument

- Subset Sum is NP-Complete
- Breaking knapsack cipher involves solving a subset sum problem
- Therefore, knapsack cipher is secure

Flaw: NP-Complete means there is no fast general solution. Some instances may be solved quickly.
(Note: Adi Shamir found a way of breaking knapsack cipher [1982])

Levels of Abstraction: Program

- Real World Problem
- High-Level Program

- Physical World
- Virtual World

Physical Processor

From Lecture 3

Crossing-Levels

- Python Program
- C Program
- C compiler
- x86 Instructions

Programming Languages
Why so many Programming Languages?

"Jamais Jamais Jamais" from Harmonice Musices Odhecaton A. Printed by Ottaviano Dei Petrucci in 1501 (first music with movable type)

"Jamais Jamais Jamais" from Harmonice Musices Odhecaton A. Printed by Ottaviano Dei Petrucci in 1501

Modern Music Notation

Roman Haubenstock-Ramati, Concerto a Tre

John Cage, Fontana Mix

http://www.medienkunstnetz.de/works/fontana-mix/audio/1/
Thought and Action

- Languages change the way we think
  - Scheme: think about procedures
  - BASIC: think about GOTO
  - Algol, Pascal: think about assignments, control blocks
  - Java: think about types, squiggles, exceptions
  - Python?
- Languages provide abstractions of machine resources
  - Hide dangerous/confusing details: memory locations, instruction opcodes, number representations, calling conventions, etc.

Abstractions

- Higher level abstractions
  - Python, Java, BASIC, ...
  - Easier to describe abstract algorithms
  - But, cannot manipulate low-level machine state
    - How are things stored in memory?
    - Opportunities for optimization lost
- Lower level abstractions
  - C, C++, JVML, MSIL, Assembly, ...
  - Harder to describe abstraction algorithms
  - Provides programmer with control over low-level machine state

Biggest Single Difference: Memory Management

- High-level languages (Python, Java) provide automatic memory management
  - Programmer has no control over how memory is allocated and reclaimed
  - Garbage collector reclaims storage
- Low-level languages (C, Assembly) leave it up to the programmer to manage memory

C Programming Language

- Developed to build Unix OS
- Main design considerations:
  - Compiler size: needed to run on PDP-11 with 24KB of memory (Algol60 was too big to fit)
  - Code size: needed to implement the whole OS and applications with little memory
    - Performance, Portability
- Little (if any consideration):
  - Security, robustness, maintainability

C Language

- No support for:
  - Array bounds checking
  - Null dereferences checking
  - Data abstraction, subtyping, inheritance
  - Exceptions
  - Automatic memory management
- Program crashes (or worse) when something bad happens
- Lots of syntactically legal programs have undefined behavior
Example C Program

```c
void test (int x) {
    while (x = 1) {
        printf ("I'm an imbecile!\n");
        x = x + 1;
    }
}
```

In Java:
```java
void test (int x) {
    while (x = 1) {
        printf ("I'm an imbecile!\n");
        x = x + 1;
    }
}
```

```
> javac Test.java
Test.java:21: incompatible types
found : int
required: boolean
while (x = 1) {
```

Weak type checking:
In C, there is no boolean type. Any value can be the test expression.
```c
x = 1 assigns 1 to x, and has the value 1.
```

```c
I'm an imbecile!
I'm an imbecile!
I'm an imbecile!
```

#### = vs. :=

Why does Python use = for assignment?

- Algol (designed for elegance for presenting algorithms) used :=
- CPL and BCPL based on Algol, used :=
- Thompson and Ritchie had a small computer to implement B, saved space by using = instead
- C was successor to B (also on small computer)
- C++’s main design goal was backwards compatibility with C
- Python was designed to be easy for C and C++ programmers to learn

C Bounds Non-Checking

```c
int main (void) {
    int x = 9;
    char s[4];
    gets(s);
    printf ("s is: \%s\n", s);
    printf ("x is: \%d\n", x);
}
```

```
> gcc -o bounds bounds.c
> bounds
```

```
abcdefghijkl
s is: abcdefghijkl
x is: 9
> bounds
```

```
abcdefghijklm
s is: abcdefghijklmn
x is: 1828716553
> bounds
```

```
abcdefghijkln
s is: abcdefghijkln
x is: 1845493769
> bounds
```

Note: your results may vary (depending on machine, compiler, what else is running, time of day, etc.). This is what makes C fun!

CPL (1963), U Cambridge :=
BCPL (1967), MIT :=
Basic Combined Programming Language
B (1969), Bell Labs =
C (1970), Bell Labs =
C++ (1983), Bell Labs =

Java (1995), Sun =

Charge

- Wednesday: Exam 1 is out, due Monday
- No regularly scheduled Small Hall and office hours while Exam 1 is out

```