Improving Software Security via Runtime Instruction-Level Taint Checking

Jingfei Kong, Cliff C. Zou and Huiyang Zhou

School of Electrical Engineering and Computer Science
University of Central Florida

Outline

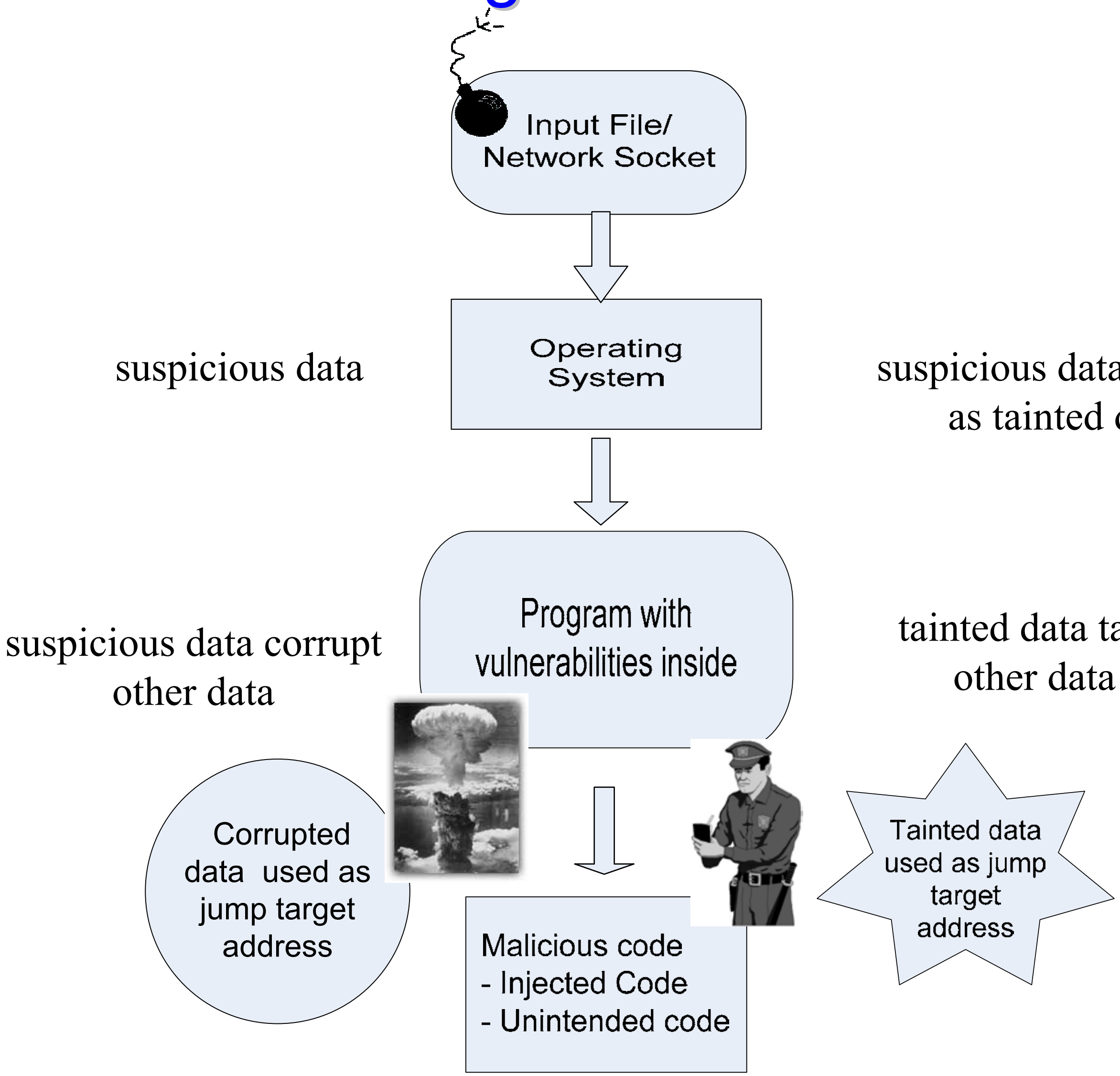
- Motivation
- Design
- Experiments
- Applications
- Conclusion

Low-level Memory Vulnerabilities

- Program defects
  - Unsafe languages (C /C++) do not check/restrict memory accesses
  - Buffer Overflow, Heap Corruption, Format String …
  - Result in many kinds of exploits – major threats to the internet

- No Silver bullet so far
  - Hard to find and fix defects in advance
  - Exploit-focused solutions can always be bypassed.
  - Defect-focused solutions are costly

Tainting Architectures



Non-control Data Attacks and Pointer Taintedness Checking

- Security-Critical Non-Control Data (Chen' Usenix Security 2005)
  - Configuration data
  - User input
  - User identity data
  - Decision-making data
  - Conclusion: as effective as the well-known control data attacks

- Pointer Taintedness Checking (Chen' DSN 2005)
  - Tainted data are not allowed to be used as pointers
  - Untaint tainted data only when *compared* with untainted value

Problems with Pointer Taintedness Checking

There are some attacks which do not use tainted pointers

```
void information_leakage()
{
    unsigned int limit = 50;
    char buf[20];
    int  p[50];
    unsigned int i;

    //buffer overflow!
A:  scanf("%s", buf);
B:  for (i=0; i<limit; i++)
    printf(" %x ", *(p+i));
}
```

Another Example

```
*void leak() {
        int secret_key;
        char buf[12];
 A:     recv(s,buf,12,0);
 B:     printf(buf); //format  string!
}
```

*From Chen' DSN 2005

A New Format String Attack

This new format string attack can write arbitrary ***untainted*** values (Dalton' WDDD 2006), even with arbitrary untainted target addresses

- A typical format string attack
  - The format string supplies the target address directly
  - The format string also contains constant widths to specify the value to be written
- The new attack
  - Use "*" specifier to get field widths from *untainted values* in the *stack* to construct arbitrary untainted values

Our Idea: Instruction-Level Taint Checking

- Instruction-level -- a generic form of taint checking
  - Cover more data taint checking than previous ones
  - Minor changes to the existing taintedness tracking architectures
  - Provide a higher degree of security protection

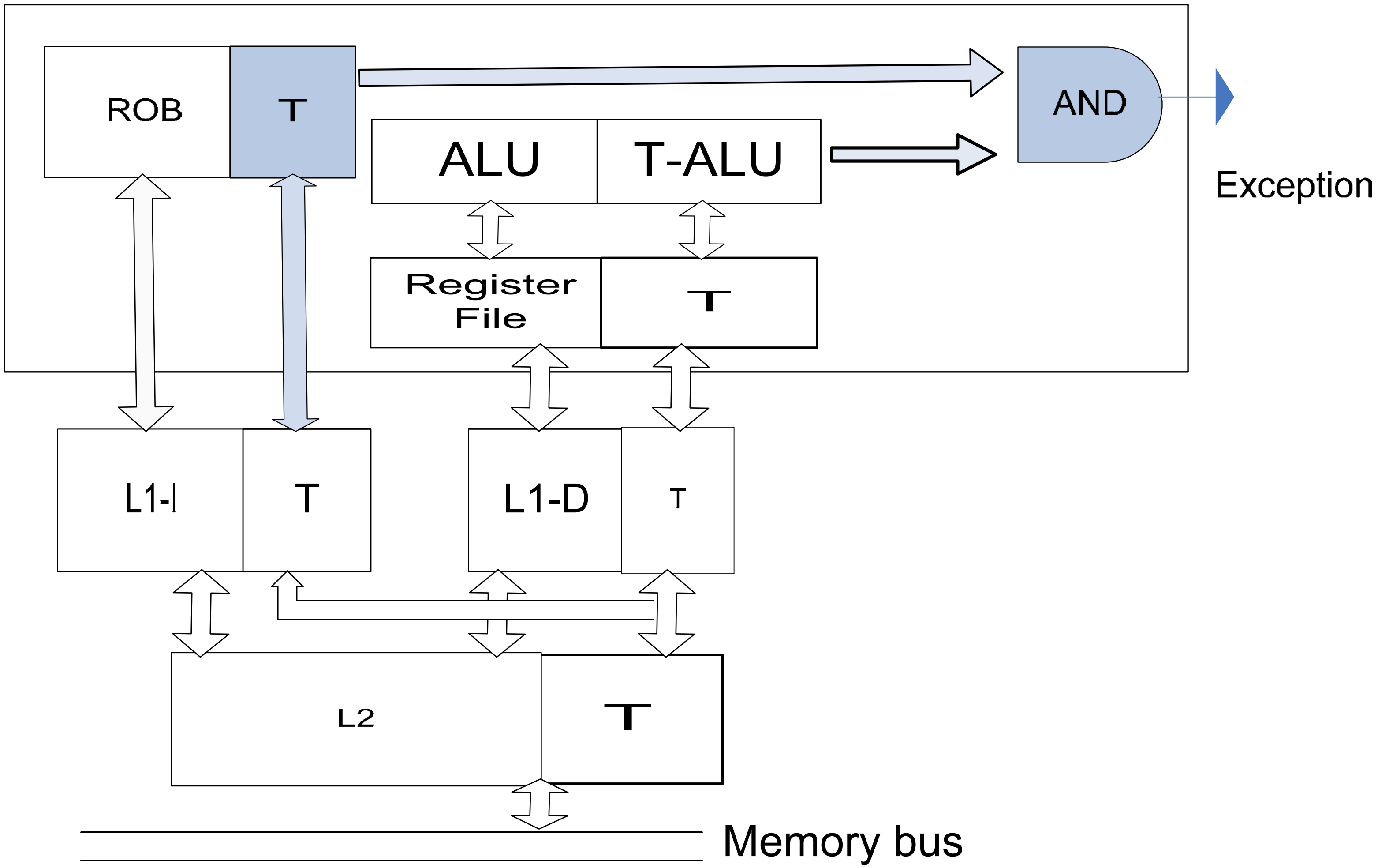
- Taintless-Instruction
  - An instruction does not deal with tainted data

- Tainted-Instruction
  - An instruction deals with tainted data

Outline

- Motivation
- Design
- Experiments
- Applications
- Conclusion

Our Design



ROB
T
AND
ALU
T-ALU
Exception
Register File
T
L1-I
T
L1-D
T
L2
T
Memory bus

Instruction-Level Taint Checking

- Taintless-Instruction Profile collection
  - Manual annotation
  - Static analysis
  - Dynamic training

- Taint checking is carried out in four steps at runtime
  - Load the collected Taintless-Instruction profile
  - Tag data from suspicious input channels as tainted
  - Track taintedness propagation through execution
  - Raise an alarm when a Taintless-Instruction encounters some tainted operand

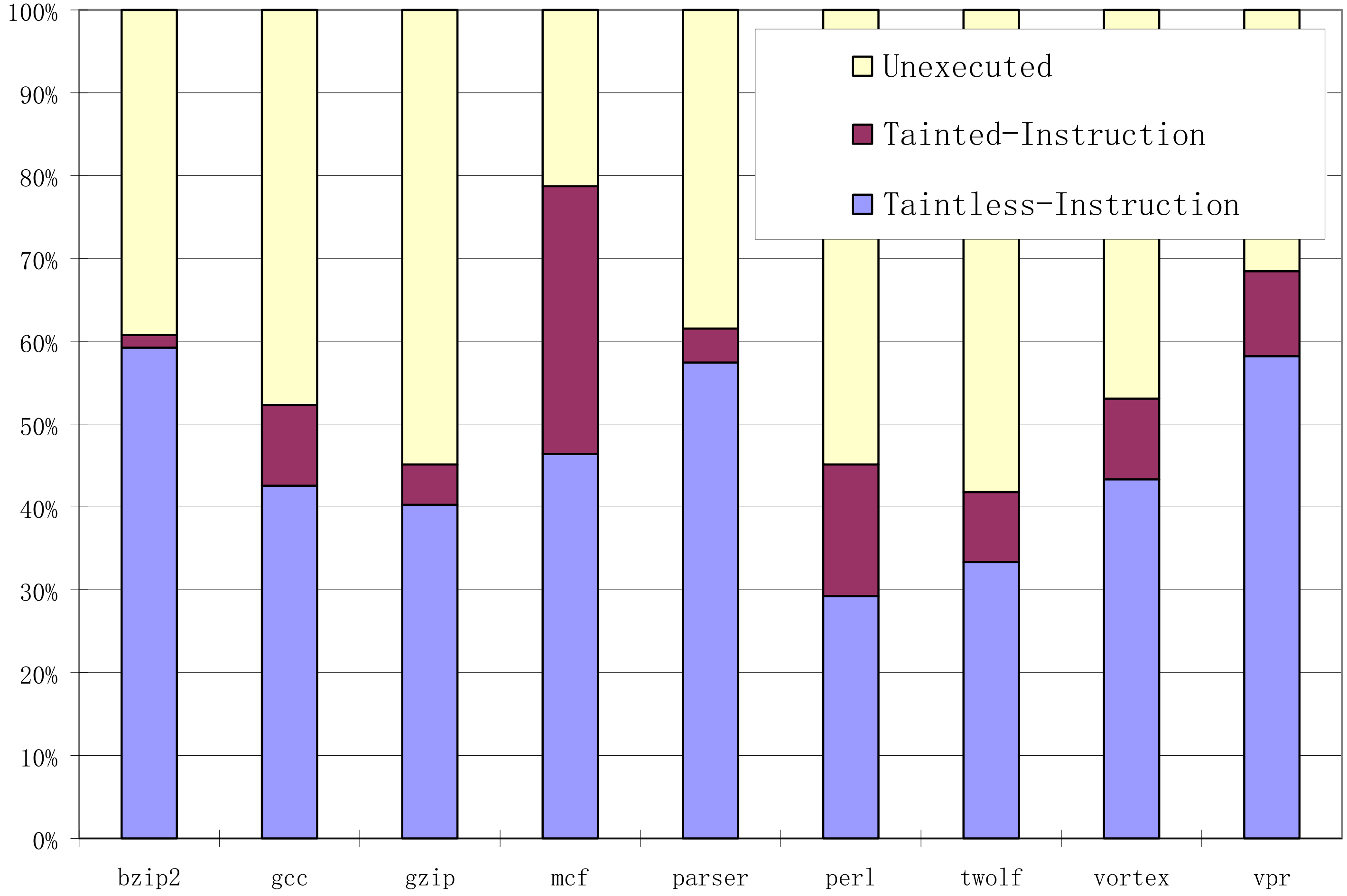
Outline

- Motivation
- Design
- Experiments
- Applications
- Conclusion

Experimental Methodology

- SimpleScalar processor simulator with the PISA instruction set
  - Byte-granularity taintedness tracking
  - Taint data from I/O system calls such as READ, RECV, etc.
  - Bitwise OR of the taintedness bits from source operands
    - Similar to the rules used in Chen' DSN 2005

Preliminary Results from SPEC CPU2000



Outline

- Motivation
- Design
- Experiments
- Applications
- Conclusion

Canary Protection Schemes Against Buffer Overflow

```
/* declaration part of local variables */
    volatile int guard;

/* the entry point */
    guard = guard_value;

/* the exit point */
    if (guard != guard_value) {
    /* output error log */
    /* halt execution */
    }
```

ProPolice
http://www.trl.ibm.com/projects/security/ssp/

Buffer Overflow

Taintless Instructions

```
void  vuln_stack_function_ptr(int choice)
   volatile unsigned int pad_end = 0;
   long stack_buffer[BUFSIZE];
   volatile unsigned  int pad_begin = 0;

   void (*stack_function_pointer)(void);
      . . .
   memcpy(stack_buffer,
          overflow_buffer, overflow+4);
   pad_begin++;
   pad_end++;
   (void)(*stack_function_pointer)();
```

Untainted data

```
jal     00404e80 <memcpy>
lw      $v0[2],32($sp[29])
addiu   $v0[2],$v0[2],1
sw      $v0[2],32($sp[29])
lw      $v0[2],32($sp[29])
lw      $v0[2],104($sp[29])
addiu   $v0[2],$v0[2],1
sw      $v0[2],104($sp[29])
lw      $v0[2],104($sp[29])
lw      $v0[2],108($sp[29])
jalr    $ra[31],$v0[2]
```

Code from the testbed of twenty buffer overflow attacks by John Wilander' NDSS 2003

Comparison

- Our improvements
  - Fewer number of instructions to be executed
  - More secure since the guard values do not have to remain secret

Format String

```
int
DEFUN(vfprintf, (s, format, args),
  register FILE *s AND CONST char
      *format AND va_list args)
{
/* Pointer into the format string.  */
      register CONST char *f;
          . . .
      f = format;

A:  while (*f != '\0')

      {
B:        . . .
```

The Devil enters here

Mark them as Taintless-Instructions to catch format string attacks

```
        addu  $t0[8],$zero[0],$s0[16]
        j     A
B:      . . .
        . . .
A:      lb    $v0[2],0($t0[8])
        lbu   $v1[3],0($t0[8])
        bne   $v0[2],$zero[0], B
```

Conclusion

- A new generic instruction-level taint checking architecture
  - Minor changes to the existing taintedness tracking architectures
  - Minor performance overhead
  - Compatible with existing ISAs
  - Provide a higher degree of security protection

Ongoing Work

- Experiments on real applications
  - Instruction-Level program taintedness behavior
  - Collection of Taintless-Instructions profile
    - Static analysis
    - Dynamic training
  - Evaluation of taint checking schemes

Thank you!

Questions?