



XIAMEN UNIVERSITY MALAYSIA
廈門大學 馬來西亞分校

SWE301


Software Validation & Verification

Chapter 3 : Software Error Mitigation Approach Part 3

Ms. Subashini Raghavan

A1-409

subashini.raghavan@xmu.edu.my

 012-3214859



Control Flow Hijacking

- Software flaws leads to vulnerabilities in software. One common and yet still prevalent flaw is stack buffer overflow.
- Stack buffer overflow leads to overflow of memory areas, hence disrupting the normal execution sequence (i.e. control flow) of a program.
- Consider a function `main()` calling a sub function `func()`.



Control hijacking attacks

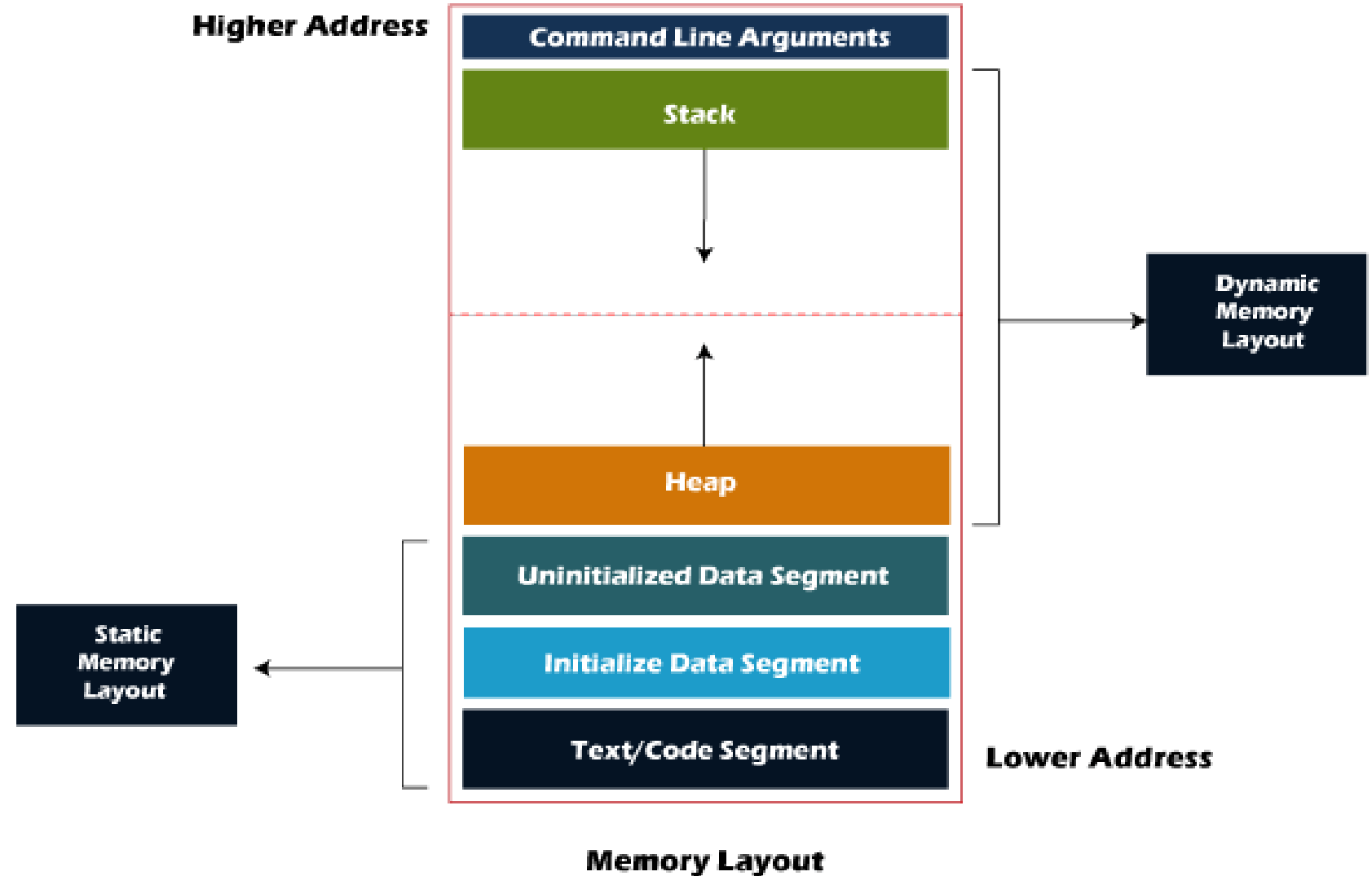
- Attacker's goal:

Take over target machine (e.g. web server)

Execute arbitrary code on target by hijacking application control flow

Control hijacking attacks

- MEMORY LAYOUT

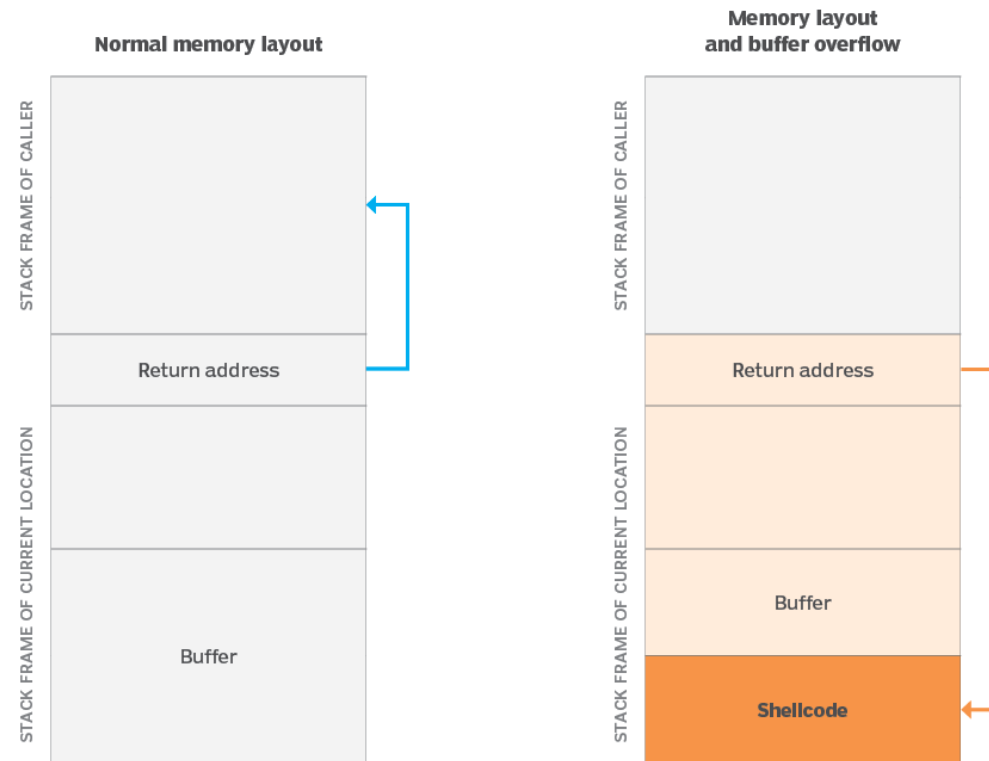


Control hijacking attacks

- BUFFER OVERFLOW

Stack buffer overflow attack

Memory layout before and after a stack buffer overflow attack



Control hijacking attacks

- MEMORY ERRORS

1. Stack-Based Buffer Overflow

```
int main(int argc, char *argv[]) {  
    char buff[256];  
    strcpy(buff, argv[1]);  
}
```

Control hijacking attacks

- MEMORY ERRORS

2. Heap-Based Buffer Overflow

```
int main(int argc, char *argv[]) {  
    char *buf = malloc(256);  
    strcpy(buf, argv[1]);  
}
```


Control hijacking attacks

```
#include <stdio.h>
```

```
int main(int argc, char *argv[]) {  
    if (argc > 1) {  
        char *user_input = argv[1];  
        printf("%s", user_input); // Use a hardcoded format string.  
    } else {  
        printf("No input provided.\n");  
    }  
    return 0;  
}
```

Control hijacking attacks

```
#include <stdio.h>

int main(int argc, char *argv[]) {
    if (argc > 1) {
        char *user_input = argv[1];
        printf("%s", user_input); // Use a hardcoded format string.
    } else {
        printf("No input provided.\n");
    }
    return 0;
}
```

Control hijacking attacks

- MEMORY ERRORS

2. Integer Overflow or Integer Wraparound

```
int main(int argc, char *argv[]) {  
    char buf[50];  
    int i = atoi(argv[1]);  
    unsigned short s = i;  
    if (s > 10) {  
        return;  
    }  
    memcpy(buf, argv[2], i);  
}
```

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
```

```
int main(int argc, char *argv[]) {
    if (argc < 3) {
        printf("Usage: %s <integer> <data>\n", argv[0]);
        return 1;
    }
    int i = atoi(argv[1]);
    if (i <= 0 || i > 10) {
        printf("Invalid integer value. Must be between 1 and 10.\n");
        return 1;
    }

    char buf[50];
    strncpy(buf, argv[2], i);
    buf[i] = '\0'; // Ensure null-termination.

    // Now, 'buf' contains at most 'i' characters from the second argument.
    return 0;
}
```

Control hijacking attacks

- MEMORY ERRORS

4. Use After Free

```
int main(void){  
    char *pt_1,*pt2;  
    pt_1 = malloc(100);  
    free(pt_1);  
    pt_2 = malloc(100);  
    strncpy(pt_1,"HELLO",100); ← Use pt_1 after being freed.  
    printf("%snn", pt_2);  
}
```

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(void) {
    char *pt_1, *pt_2;
    pt_1 = malloc(100);
    free(pt_1); // Free memory pointed to by pt_1

    pt_2 = malloc(100); // Allocate new memory with pt_2

    if (pt_2 != NULL) {
        strncpy(pt_2, "HELLO", 100); // Use pt_2
        printf("%s\n", pt_2);
        free(pt_2); // Free memory allocated with pt_2
    } else {
        printf("Memory allocation failed for pt_2.\n");
    }

    return 0;
}
```

Control hijacking attacks

- MEMORY ERRORS

5. NULL Pointer Dereference

```
#include <stdio.h>

int main() {
    int *ptr = NULL; // Initialize a pointer to NULL
    *ptr = 42; // NULL pointer dereference

    return 0;
}
```

```
#include <stdio.h>
```

```
int main() {
```

```
    int *ptr = NULL; // Initialize a pointer to NULL
```

```
    if (ptr != NULL) {
```

```
        *ptr = 42; // Dereference only if the pointer is not NULL
```

```
    } else {
```

```
        printf("Pointer is NULL, so it should not be dereferenced.\n");
```

```
    }
```

```
    return 0;
```

```
}
```


More on Buffer Overflow

What causes buffer overflows?

Example: `gets`

```
char buf[20];  
gets(buf) ; // read user input until  
             // first EoL or EoF character
```

- *Never* use `gets`
- Use `fgets(buf, size, stdin)` instead

Example: **strcpy**

```
char dest[20];
```

```
strcpy(dest, src); // copies string src to dest
```

- **strcpy** assumes **dest** is long enough , and assumes **src** is null-terminated
- Use **strncpy**(dest, src, size) instead

Spot the defect! (1)

```
char buf[20];  
char prefix[] = "http://";  
...  
strcpy(buf, prefix);  
    // copies the string prefix to buf  
strncat(buf, path, sizeof(buf));  
    // concatenates path to the string  
    buf
```

Spot the defect! (1)

```
char buf[20];  
char prefix[] = "http://";  
...  
strcpy(buf, prefix);  
// copies the string prefix to buf  
strncat(buf, path, sizeof(buf));  
// concatenates path to the string buf
```

strncat's 3rd parameter is number of chars to copy, not the buffer size

Another common mistake is giving `sizeof(path)` as 3rd argument...

Spot the defect! (2)

```
char src[9];  char dest[9];
```

```
char* base_url = "www.ru.nl";
```

```
strncpy(src, base_url, 9);
```

```
// copies base_url to src
```

```
strcpy(dest, src);
```

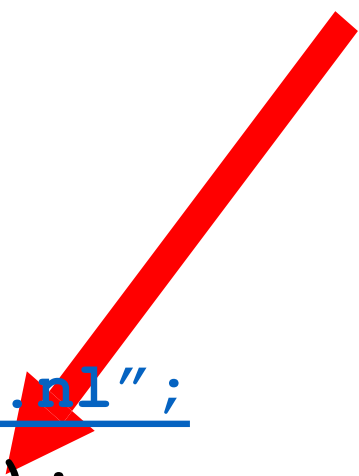
```
// copies src to dest
```

Spot the defect! (2)

```
char src[9];  
char dest[9];
```

```
char* base_url = "www.ru.nl";  
strncpy(src, base_url, 9);  
// copies base_url to src  
strcpy(dest, src);  
// copies src to dest
```

`base_url` is 10 chars long,
incl. its null terminator, so
`src` will not be
null-terminated



Spot the defect! (2)

```
char src[9];  char  
dest[9];
```

```
char* base_url = "www.ru.nl";  
strncpy(src, base_url, 9);  
// copies base_url to src  
strcpy(dest, src);  
// copies src to dest
```

`base_url` is 10 chars long, incl. its
null terminator, so `src` will not be
null-terminated

so `strcpy` will overrun the buffer `dest`

Example: **strcpy** and **strncpy**

- Don't replace

`strcpy(dest, src)`

by

`strncpy(dest, src, sizeof(dest))`

but by

`strncpy(dest, src, sizeof(dest)-1)`

`dst[sizeof(dest)-1] = `\\0`;`

if `dest` should be null-terminated!

- Btw: a **strongly typed programming language** could of course enforce that strings are always null-terminated...

Spot the defect! (3)

```
char *buf;  int i, len;
```

```
read(fd, &len, sizeof(int));
```

```
// read sizeof(int) bytes, ie. an int,
```

```
// and store these in len
```

```
buf = malloc(len);
```

```
read(fd,buf,len); // read len bytes into buf
```

Spot the defect! (3)

```
char *buf;  int i, len;
```

len might become negative




```
read(fd, &len, sizeof(int));
```

```
    // read sizeof(int) bytes, ie. an int,
```

```
    // and store these in len
```

```
buf = malloc(len);
```

```
read(fd, buf, len); // read len bytes into buf
```



len cast to unsigned, so negative length overflows
read then goes beyond the end of **buf**

Spot the defect!(3)

```
char *buf;  int i, len;

read(fd, &len, sizeof(len));  if (len < 0)
{error ("negative length"); return; }
buf = malloc(len);  read(fd,buf,len);
```

Remaining problem may be that `buf` is not null-terminated

Spot the defect!(3)

```
char *buf;  
int i, len;
```

May result in **integer overflow**; we should check that `len+1` is positive



```
read(fd, &len, sizeof(len)); if (len < 0)  
{error ("negative length"); return; }  
buf = malloc(len+1); read(fd,buf,len);  
buf[len] = '\0'; // null terminate buf
```

Spot the defect! (4)

```
#define MAX_BUF 256

void BadCode (char* input)
{
    short len;
    char buf[MAX_BUF];

    len = strlen(input);

    if (len < MAX_BUF) strcpy(buf,input);
}
```

Spot the defect! (4)

```
#define MAX_BUF 256
```

```
void BadCode (char* input)
```

```
{  short len;
```

```
  char buf[MAX_BUF];
```

```
  len = strlen(input);
```

```
  if (len < MAX_BUF) strcpy(buf, input);
```

```
}
```

What if `input` is longer than 32K ?

len will be a negative number,
due to integer overflow

hence: potential buffer
overflow

The integer overflow is the root problem, but the (heap) buffer overflow that this enables make it exploitable

Spot the defect!(5)

```
#ifdef UNICODE
#define _sntprintf _snwprintf
#define TCHAR wchar_t
#else
#define _sntprintf _snprintf #define TCHAR
char
#endif


TCHAR buff[MAX_SIZE];
_sntprintf(buff, sizeof(buff), "%s\n", input);
```


Spot the defect!(5)

```
#ifdef UNICODE
#define _sntprintf _snwprintf
#define TCHAR wchar_t
#else
#define _sntprintf _snprintf
#define TCHAR char
#endif
```

```
TCHAR buff[MAX_SIZE];
_sntprintf(buff, sizeof(buff), "%s\n", input);
```

**_sntprintf's 2nd param is # of chars in
buffer, not # of bytes**



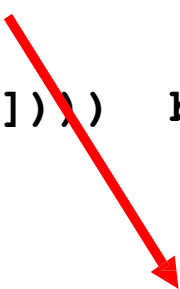
The CodeRed worm exploited such a mismatch: code written under the assumption that 1 char was 1 byte allowed buffer overflows after the move from ASCII to Unicode

Spot the defect! (6)

```
bool CopyStructs(InputFile* f, long count)
{
    structs = new Structs[count];
    for (long i = 0; i < count; i++)
    { if !(ReadFromFile(f,&structs[i])) break;
    }
}
```

Spot the defect! (6)

```
bool CopyStructs(InputFile* f, long count)
{
    structs = new Structs[count];
    for (long i = 0; i < count; i++)
    {
        if !(ReadFromFile(f, &structs[i])) break;
    }
}
```



effectively does a `malloc(count*sizeof(type))` which
may cause integer overflow

And this integer overflow can lead to a (heap) buffer overflow.
Since 2005 the Visual Studio C++ compiler adds check to prevent this

Spot the defect!(7)

```
char buff1[MAX_SIZE], buff2[MAX_SIZE];

// make sure url is valid and fits in buff1 and buff2:
if (! isValid(url)) return;
if (strlen(url) > MAX_SIZE - 1) return;

// copy url up to first separator, ie. first '/', to buff1
out = buff1; do {
// skip spaces
if (*url != ' ') *out++ = *url;
} while (*url++ != '/'); strcpy(buff2, buff1);
...
```

[slide from presentation by Jon Pincus]

Spot the defect! (7) - Loop termination

```
char buff1[MAX_SIZE], buff2[MAX_SIZE];

// make sure url is valid and fits in buff1 and buff2:
if (! isValid(url)) return;
if (strlen(url) > MAX_SIZE - 1) return;

// copy url up to first separator, ie. first '/', to buff1
out = buff1; do {
    // skip spaces
    if (*url != ' ') *out++ = *url;
} while (*url++ != '/'); strcpy(buff2, buff1);

...
```

length up to the first null

what if there is no '/' in the URL?

Spot the defect!(7)

```
char buff1[MAX_SIZE], buff2[MAX_SIZE];

// make sure url is valid and fits in buff1 and buff2:

if (! isValid(url)) return;
if (strlen(url) > MAX_SIZE - 1) return;

// copy url up to first separator, ie. first '/', to buff1
out = buff1;
do {
    // skip spaces
    if (*url != ' ') *out++ = *url; } while (*url != '/') &&
    (*url++ != 0); strcpy(buff2, buff1);
```

• • [• slide from presentation by Jon Pincus]

What about 0-length URLs?
Is buff1 always null-terminated?

Spot the defect!(8)

```
#include <stdio.h>

int main(int argc, char* argv[])
{
    if (argc > 1)
printf(argv[1]);  return 0;
}
```

This program is vulnerable to **format string attacks**, where calling the program with strings containing special characters can result in a buffer overflow attack.



<https://essentials.edmarket.org/2021/01/rethinking-the-computer-labs-of-the-future/>



PRACTICAL SESSION

How stack overflow can occur?

Recursive functions

```
int factorial(int n) {  
    if (n == 0) {  
        return 1;  
    } else {  
        return n * factorial(n - 1);  
    }  
}
```

Large local variables

```
void foo() {  
    char buffer[10000]; // This is a  
                        // very large buffer  
    // ...  
}
```

Deep call chains

```
void foo() {  
    bar();  
}
```

```
void bar() {  
    baz();  
}
```

```
void baz() {  
    qux();  
}
```

```
void qux() {  
    // ...  
}
```

```
int main() {  
    foo();  
    return 0;  
}
```

How the attacker takes control?

STEP 1: Control flow instructions that could be manipulated by the attacker.

1. Indirect Jump

The destination address is not embedded in the opcode but read from a memory address or is the value of a processor register.

//Direct Jump

JMP 0x12345678

//Indirect Jump

JMP [EAX] ; Jump to the address stored in the EAX register

JMP [0x789ABCDEF] ; Jump to the address stored in memory location 0x789ABCDEF

How the attacker takes control?

STEP 1: Control flow instructions that could be manipulated by the attacker.

2. Indirect Call / Call Indirect

Indirect calls are the equivalent of the previous ones but rather than a jump, the opcode corresponds to a call instruction..

`CALL [EBX]` ; Call the subroutine whose address is in the EBX register

`CALL [0xDEADBEEF]` ; Call the subroutine at the address stored in memory location 0xDEADBEEF

3. Return Indirect

`RET` ; Return to the address on the top of the stack (indirect)

How the attacker takes control?

- STEP 2: Once the attackers know how to manipulate the control flow, the second step is to use it to execute their malicious code.
- **Code Injection**

The attacker uploads or injects malicious code into the target process and transfers the control flow to the entry point to its code.

 - Just in Time Compilers (JIT).
 - Backward compatible applications
- **Code Reuse Attack**

the attacker has the opportunity to inject or modify data on the target and redirect the control flow to code that is already in the target

 - Return-to-libc
 - Return Oriented Programming
 - GOT Overwriting
 - Jump Oriented Programming
 - Sigreturn
 - Speculate execution

Control Flow Integrity (CFI)

- Security mechanism used in computer systems and software to protect against various types of attacks, particularly those that exploit vulnerabilities related to control flow hijacking.
- Helps ensure that a program's control flow, which includes the order of execution of instructions and the targets of function calls and jumps, follows a predefined and legitimate path.
- Key aspects of Control Flow Integrity
 - Control Flow Graph (CFG)
 - Policy Enforcement
 - Labels and Checks
 - Violation Handling
- CFI is effective at mitigating various attacks
 - Return-oriented programming (ROP)
 - Code injection attacks
 - Memory corruption vulnerabilities

Control-Flow Integrity

- A defense policy which restricts unintended control flow transfers to unauthorized locations.

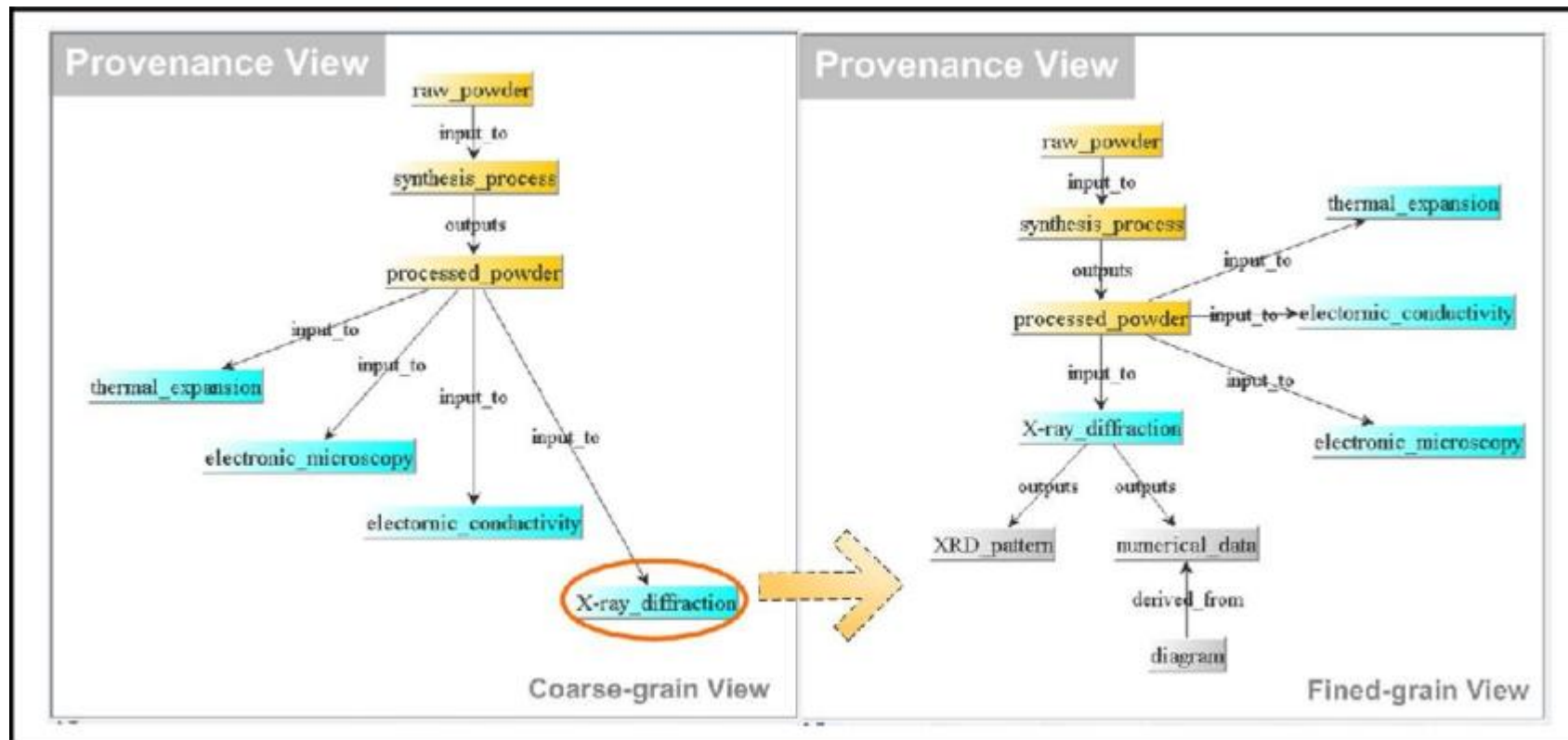


Image from [10.2218/ijdc.v3i2.55](https://doi.org/10.2218/ijdc.v3i2.55)

Fine-grained (CFI) and Coarse-grained CFI

Fine-Grained CFI:

Precision:

Enforces control flow restrictions at a very granular level, often at the level of individual instructions or basic blocks within functions.

Complexity:

More complex to implement because it requires detailed analysis of the program's control flow graph and the insertion of numerous runtime checks and labels to verify each transition.

Security:

Higher level of security because it makes it extremely difficult for attackers to deviate from the legitimate control flow path. It is effective against various advanced control flow hijacking attacks, including Return-Oriented Programming (ROP).

Overhead:

Higher runtime overhead due to the fine granularity of checks, which can impact the program's performance.

Coarse-Grained CFI:

Precision:

Enforces control flow restrictions at a higher level of abstraction, often at the level of entire functions or procedures

Complexity:

Simpler to implement compared to fine-grained CFI because it doesn't require the same level of detail in control flow tracking and checks.

Security:

Provides a reasonable level of security by preventing attacks like code injection and high-level control flow hijacking,

Overhead:

Introduces lower runtime overhead compared to fine-grained CFI because it performs fewer checks and has less detailed tracking.

Control-Flow Integrity

Table 1. Key features of CFI techniques.

CFI Techniques	Based on		Compiler Modified	Shadow Stack	CFG	Label	Coarse Grained	Fine Grained	Backward Edge	CFI Enforcement
CFI [51,52]		✓		✓	✓	✓	✓		✓	Inlined CFI
CCFI [46]	✓	✓	✓					✓	✓	Dynamic Analysis
binCFI [42]		✓				✓	✓			Static Binary Rewriting
CCFIR [59]		✓				✓	✓		✓	Binary Rewriting
HW-CFI [41]	✓	✓	✓	✓	✓	✓		✓	✓	Landing Point
PICFI [44]		✓	✓		✓			✓	✓	Static Analysis
KCoFI [61]		✓	✓			✓	✓		✓	SVA Instrumentation
Kernel CFI [62]		✓			✓			✓	✓	Retrofitting Approach
IFCC [43]		✓	✓		✓			✓		Dynamic Analysis
CFB [45]		✓		✓	✓			✓	✓	Precise Static CFI
SAFEDISPATCH [65]		✓	✓				✓			Static Analysis
C-Guard [66]		✓	✓		✓		✓			Dynamic Instrumentation
RAP [49]		✓	✓		✓			✓	✓	Type Based
O-CFI [47]	✓	✓					✓	✓	✓	Static Rewriting

Control Flow Integrity (CFI) in the Linux kernel

Kees ("Case") Cook
@kees_cook
keescook@chromium.org

Linux Conf AU 2020

<https://outflux.net/slides/2020/lca/cfi.pdf>

0:16 / 38:51 • Control Flow Integrity (CFI) in the Linux kernel >



<https://youtu.be/0Bj6W7qrOOI?si=Y6qwVggSNE0x14rX>



38:51 mins

Exercises



Buffer overflow is the most commonly found vulnerability in network-aware code that can be exploited to compromise a system. Explain the reason.

WRITE THE ANSWER and HANDOVER to MS.SUBA before LEAVING the CLASS.