NLRL

National Leading Research Lab.

미래창조과학부

Hardware-Assisted System Security

Detection Schemes for Code Reuse Attacks

백윤흥

서울대학교 전기.정보공학부

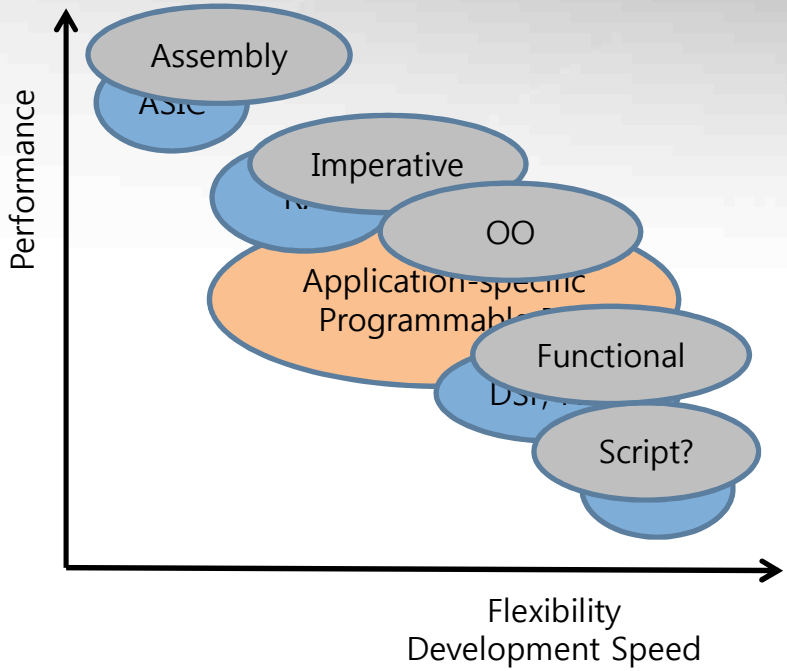
SoC Optimizations & Restructuring Lab

목차

CONTENTS

응용에 특화된 하드웨어

- 범용프로세서 (GPP)
  - 다수의 응용에 대해 빠르게 적용 가능
  - 제한된 성능 및 전력소모 문제
- 응용에 특화된 ASIC IP
  - 고속의 성능
  - 다수의 응용에 대해 대응이 어려움
    - 응용의 lifetime이 짧아짐에 따라 대응하기 위한 NRE 비용 증가
- 응용에 특화된 Programmable IP
  - Application-Specific Instruction-set Processors (ASIPs)
  - 포괄적으로는 ….
    - RAs (Reconfigurable Architectures) : CGRA, FPGA
    - Off-the-shelf ASIPs : DSP, NPU, GPU …
  - 소프트웨어 개발을 통한 다양한 응용에 최적화된 효율적인 솔루션 개발 용이 → 비정형적 ASIP구조에 특화된 고난도의 컴파일러 기술 필요



응용에 특화된 하드웨어와 보안?

- 컴퓨터 시스템에 대한 보안 공격의 기하급수적 증가
  - 누적 악성코드 2000만 이상 [McAfee]
- 축적된 보안 기술들을 모두 적용 시 이들에 99% 이상 대응 가능한 것으로 분석
  - 그런데 왜 보안은 항상 문제일까?
- 대부분 보안 솔루션들은 백신과 같은 S/W 형태로 구현
  - 보안 S/W 솔루션들은 H/W 자원을 다른 일반 응용들과 나눠쓰는 구조
  - 성능 부하가 특히 높은 고급 보안 기술들은 많은 시스템에서 배제
- 왜 이렇게 될 수 밖에 없는가?
  - 전통적으로 모든 시스템 (특히 H/W) 설계 시 "Performance First" 원칙 중시
  - 보안 기능은 기 설계된 시스템에 부가적, 대증적으로(ad-hoc) 추가되는 방식으로 구현
  - 보안 기능이 초기 시스템 설계 시 동시 반영되지 않고 나중에 S/W 형태로 추가되어 감시 효율성 저하

응용에 특화된 하드웨어와 보안의 융합

- 앞선 분석에서 얻은 교훈
  - 결국 보안도 일련의 연산(computation)들로 구성된 응용!
  - 성능 부하 정도에 의해 그 채택이 제한되고, 그 설계 구현 방식에 따라 효율성이 제한됨
  - 다른 응용들처럼 보안 연산도 효율성과 정확성을 위해 성능 최적화가 필요
- 현 보안 문제에 대한 우리의 개선책
  - 성능 우선주의의 기존 시스템(특히 H/W) 설계 방법론 변화 필요

Performance와 Security 동시 고려 설계

  - H/W-assisted Security
    - 보안 연산에 특화된 가속기 하드웨어 개발
    - 소프트웨어 위주의 보안 솔루션의 핵심 기능들을 하드웨어에 직접 지원하도록 설계
  - Security-integrated System Design
    - 컴퓨터 시스템 설계 초기부터 보안 하드웨어를 내부에 집적화함으로써 보안 감시 효율성 극대화
    - 기존 시스템 내부에서 직접적으로 정보 추출을 통한 정확한 감시 활동 지원

기존 보편적인 보안 감시 기법들

- **시그니처 (Signature) 기반 감시**
  - 악성코드의 '시그니처'를 통해 파일의 악성 여부 결정
  - '시그니처'가 없는 악성코드에 대응 불가
  - 새로운 악성코드에 대한 대응 불가
  - '시그니처'의 양이 지속적으로 증가
    - 이를 해결하기 위해 클라우드 기반의 시그니처 감시가 제안
    - 스마트폰을 포함하는 모바일 시스템이나 IoT 보안에는 적합하지 않음
- **평판 기반 감시**
  - 사용자들의 평가를 통해 파일의 악성 여부 결정
  - 새로운 악성코드에 대응하기 위해선 피해자의 발생이 필수불가결

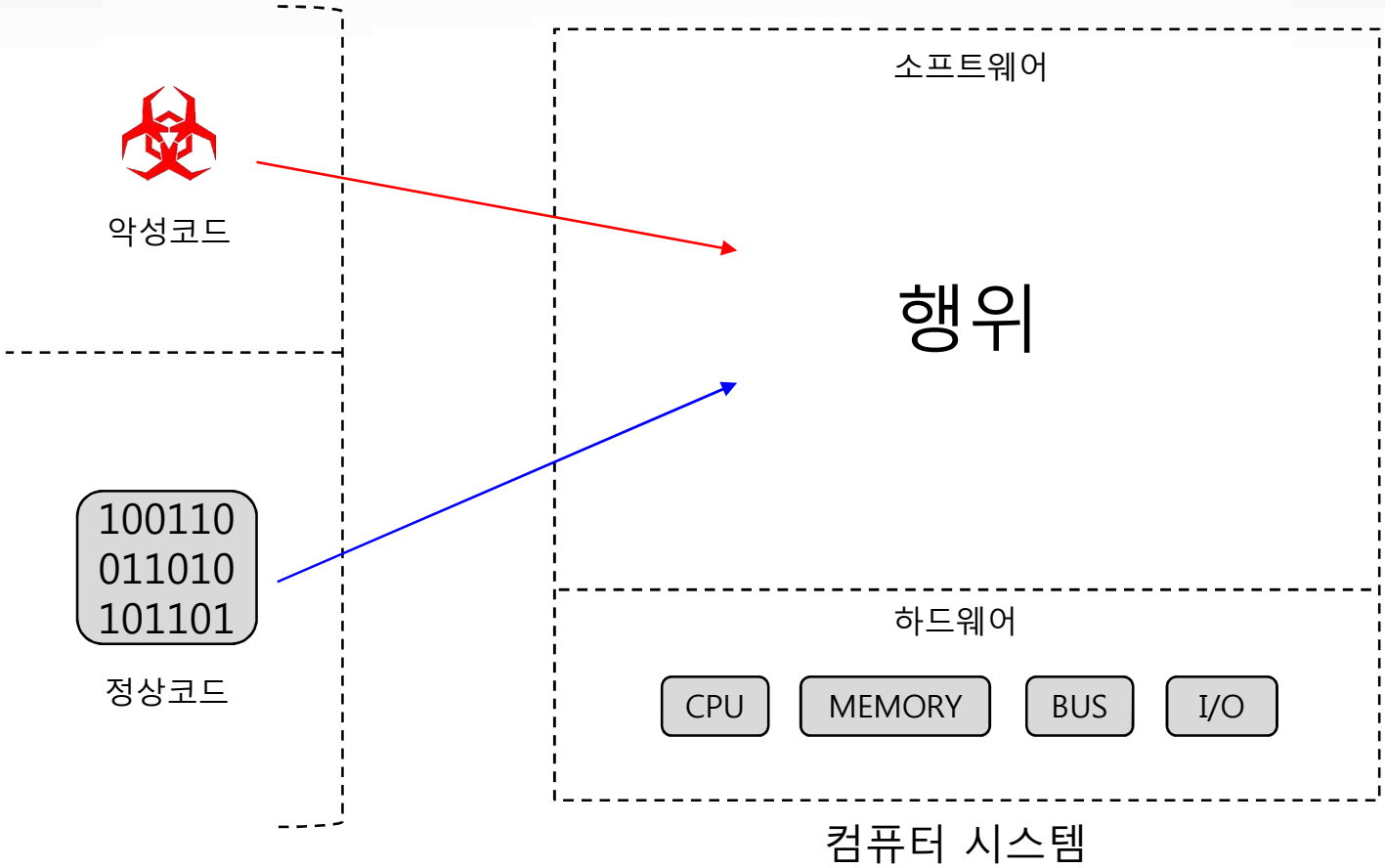
정적 구조에 기반해 악성 코드 판별

동적 행위에 기반한 시스템 보안 감시 모델 필요...

➔ 행위 기반 보안 감시 기법

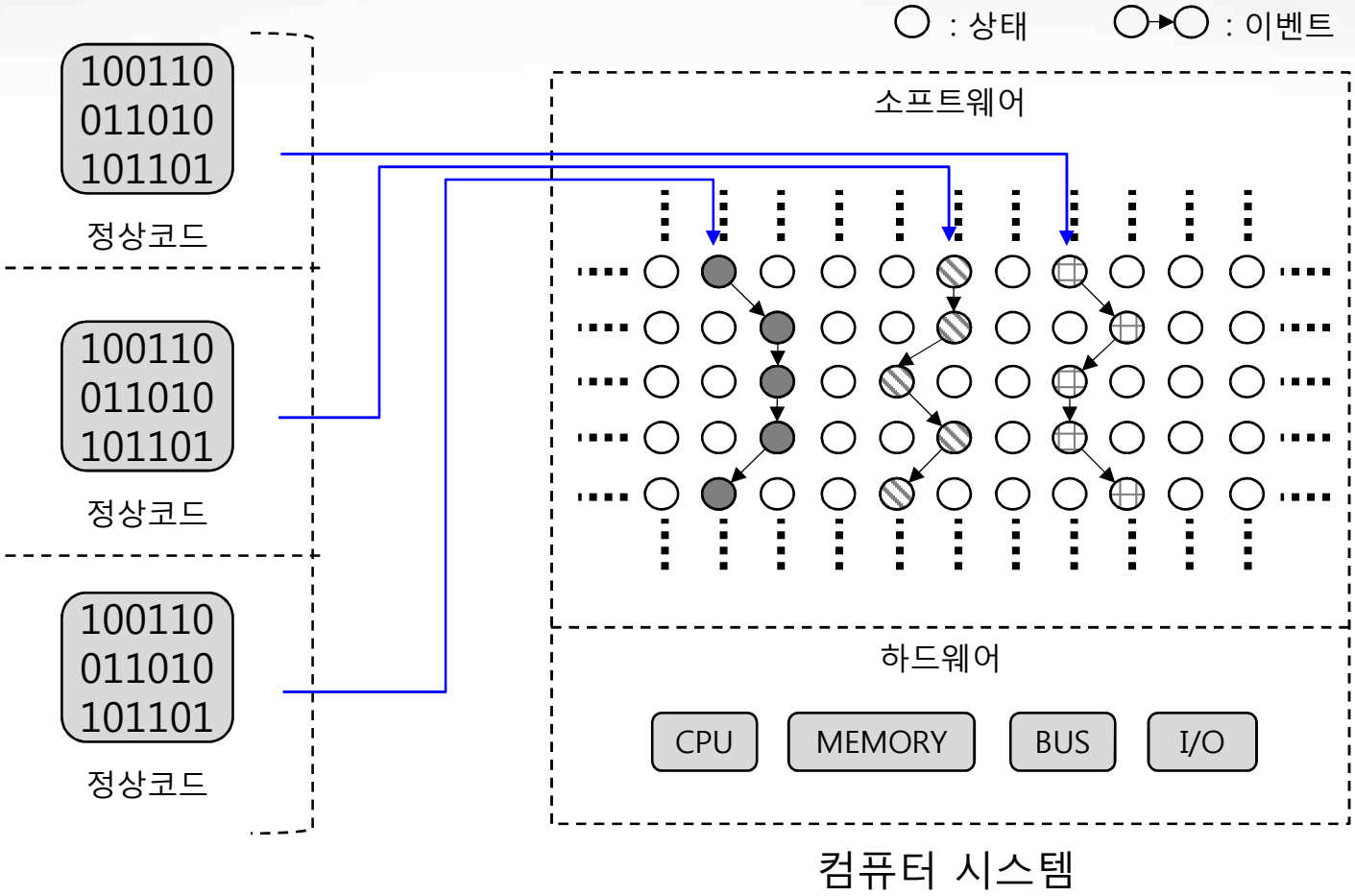
행위 기반 감시 기법

- 코드 실행 과정의 행위를 감시
- 행위란 시스템 내부의 상태(state) 변화를 일으키는 이벤트



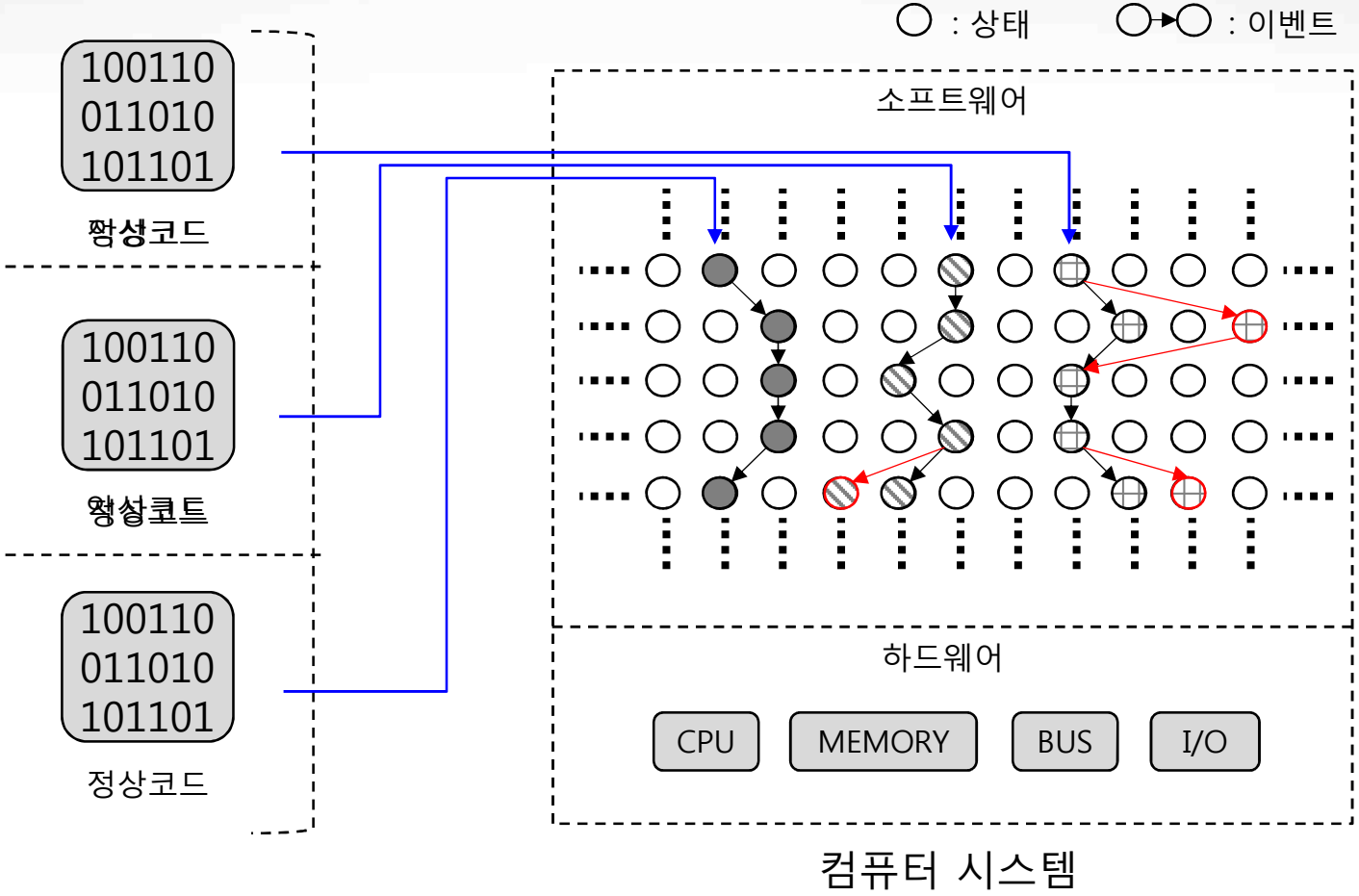
정상코드의 행위 모델링

- 각각의 코드 실행은 일련의 이벤트들 즉 컴퓨터 시스템 내부 상태 순차적 변화 과정을 일으킴



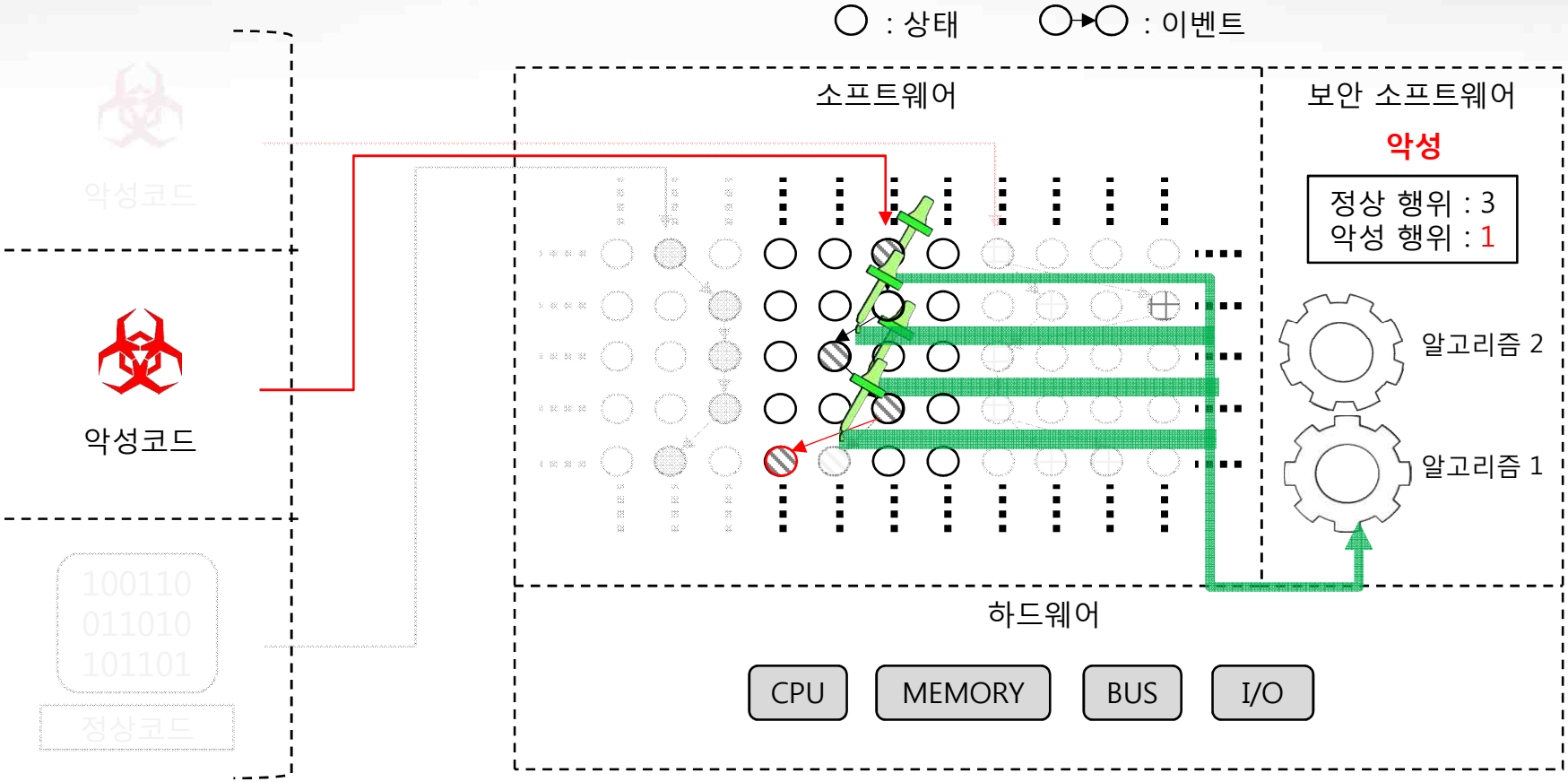
악성코드 행위 감시 기법의 목적

- 악성코드들은 일반 정상행위를 벗어나는 이벤트를 종종 발생
- 이러한 비정상 행위의 순간을 포착하여 악성코드 유무 탐지



이상적인 행위 감시 모델

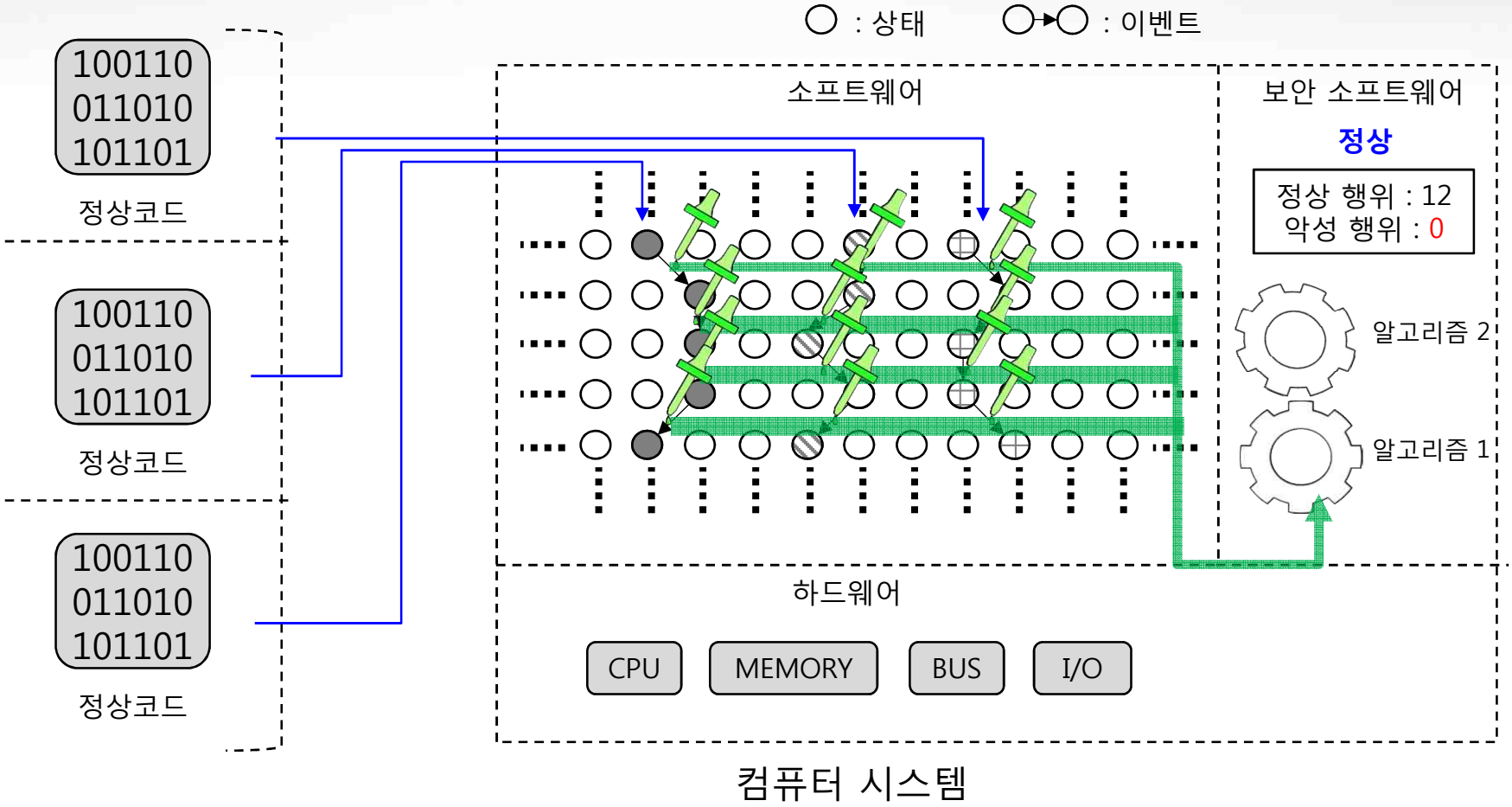
- 각 코드의 모든 행위를 감시한다고 가정
- 시스템 내 모든 코드가 발생시키는 모든 이벤트들을 상시 감시



컴퓨터 시스템

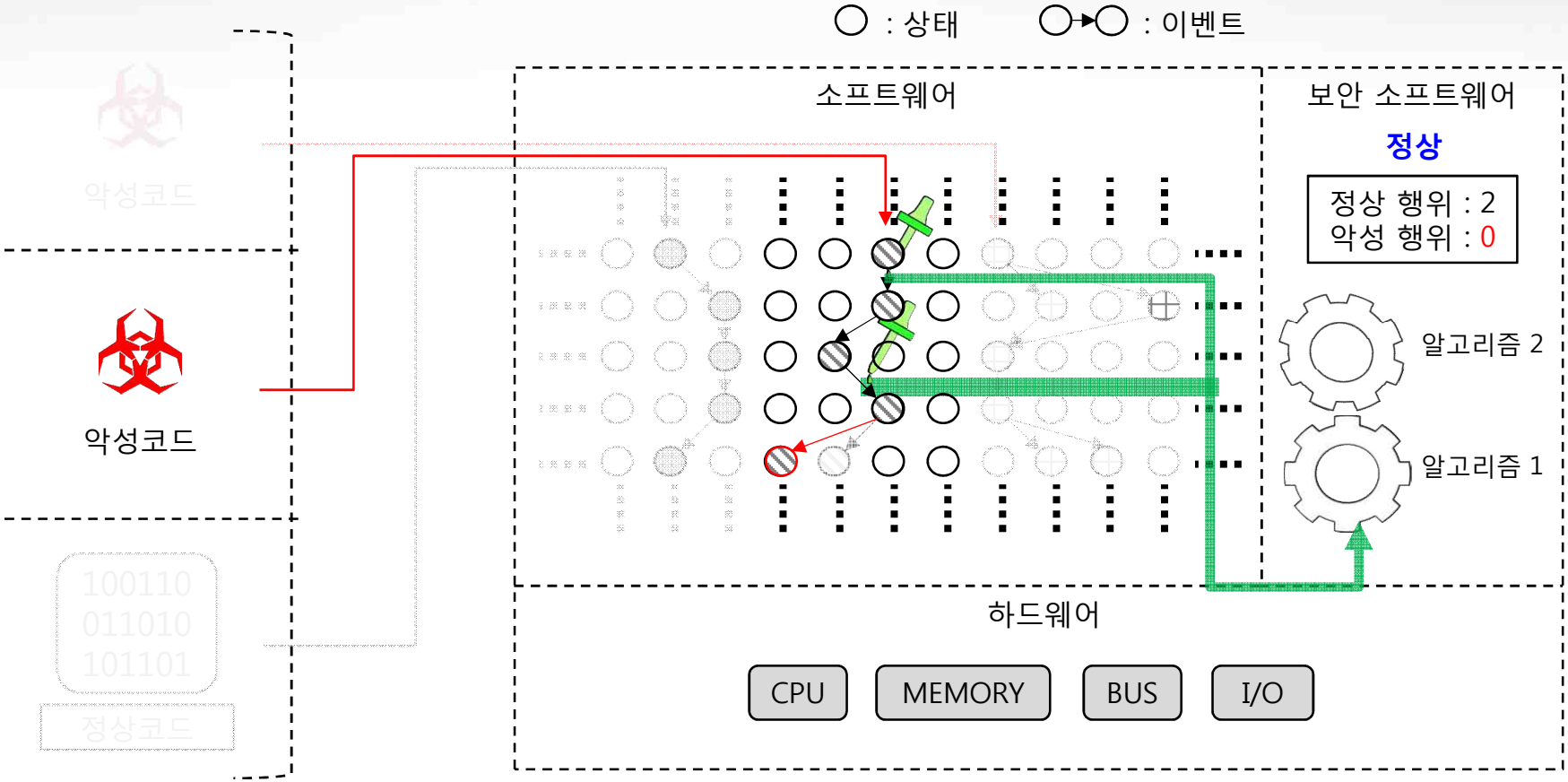
이상적인 행위 감시 모델의 한계

- 행위 감시는 '난제' ➔ 감시 대상/범위 증가시, 보안 연산량 기하급수적 증가
- 악성행위는 극히 일부 ➔ 대부분의 경우엔 필요이상 과다한 연산량 사용



샘플링기반 행위 감시 모델

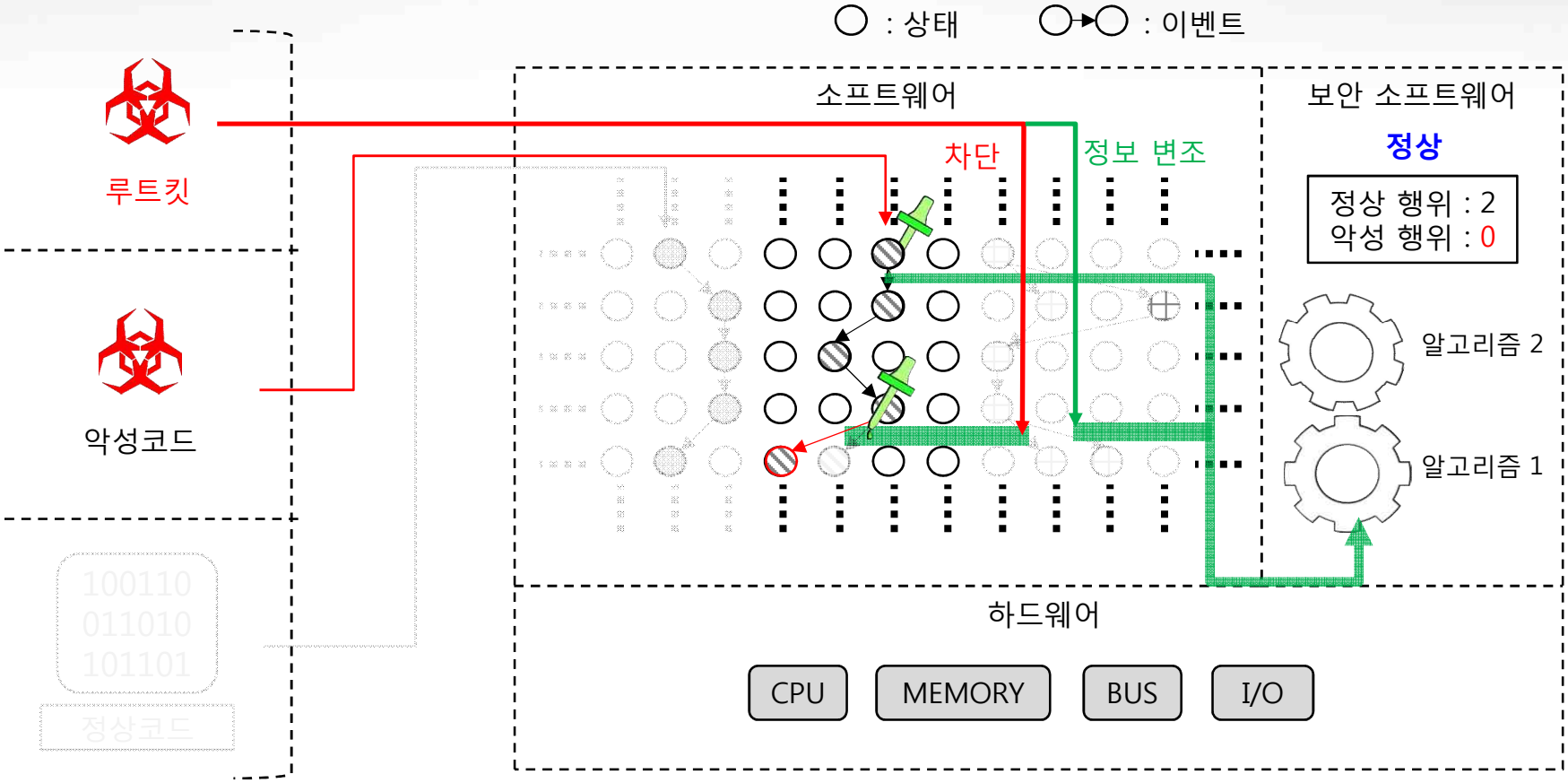
- 모니터링에 필요한 연산량 감소
- 탐지 오류의 가능성? ➔ 성능 부하 적은 상시 감시 기법 필요



컴퓨터 시스템

기존 행위 감시 모델의 또 다른 한계

- 루트킷(Rootkit)같은 OS 커널 보다 높은 권한을 가지는 악성 코드의 방해 ➔ **루트킷등의 방해에 자유로운 탐지 기법 필요**

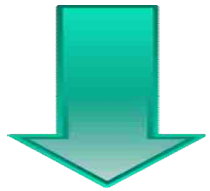


컴퓨터 시스템

행위 보안 감시 모델에 요구되는 특성

- **효율성** 높은 보안 감시를 위한 고속의 저부담 행위 감시 기술
- 루트킷 등으로부터 **안전한** 행위 추출 및 감시 기술

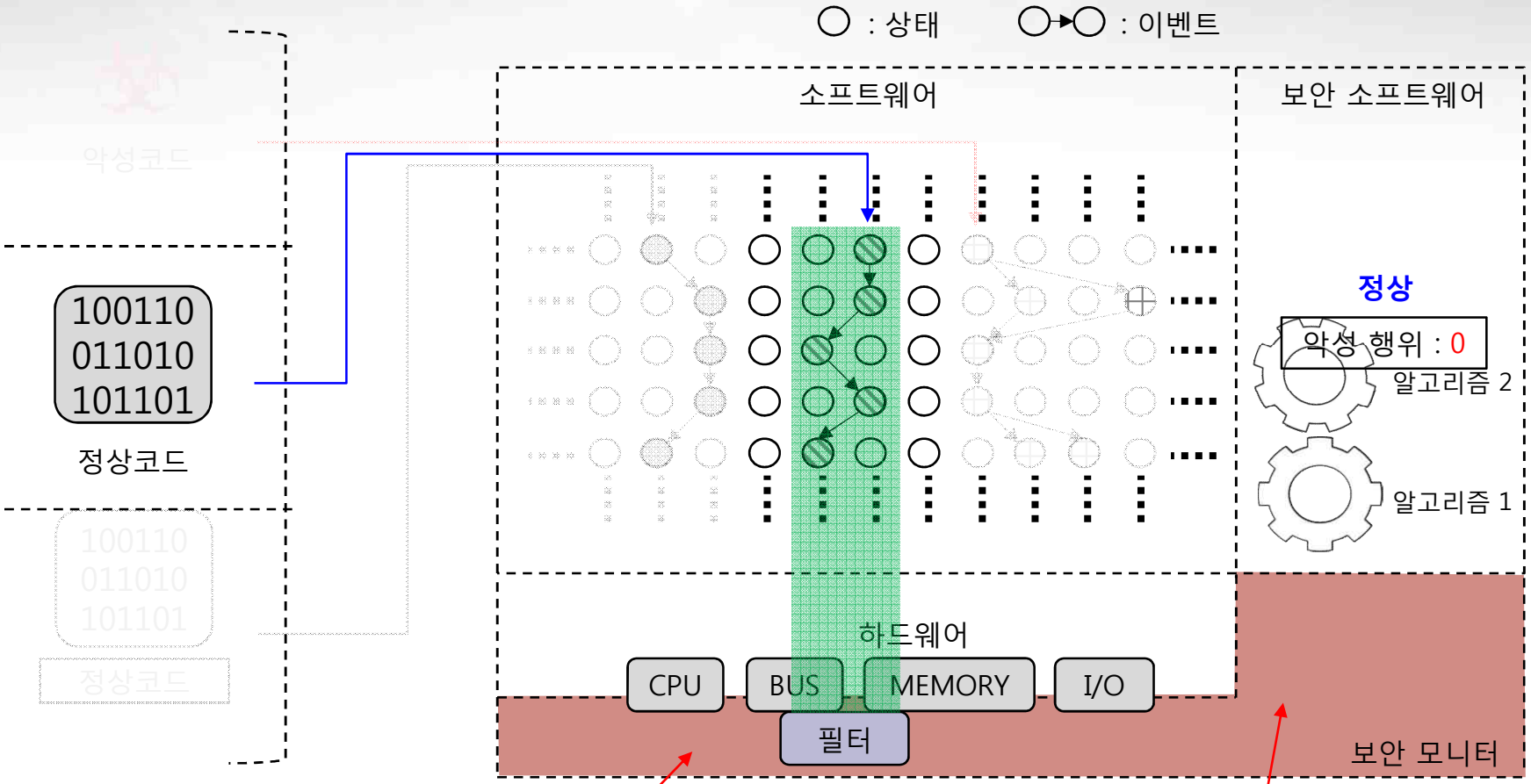
효율적이고 안전한 보안 감시



상위 어떤 감시 대상 S/W 코드보다 높은 권한을 가지며,
고속의 보안 감시 기능으로 무장된
전용 하드웨어 모듈

보안 기능이 집적된 하드웨어를 통한 감시 예시

정상코드의 행위 감시

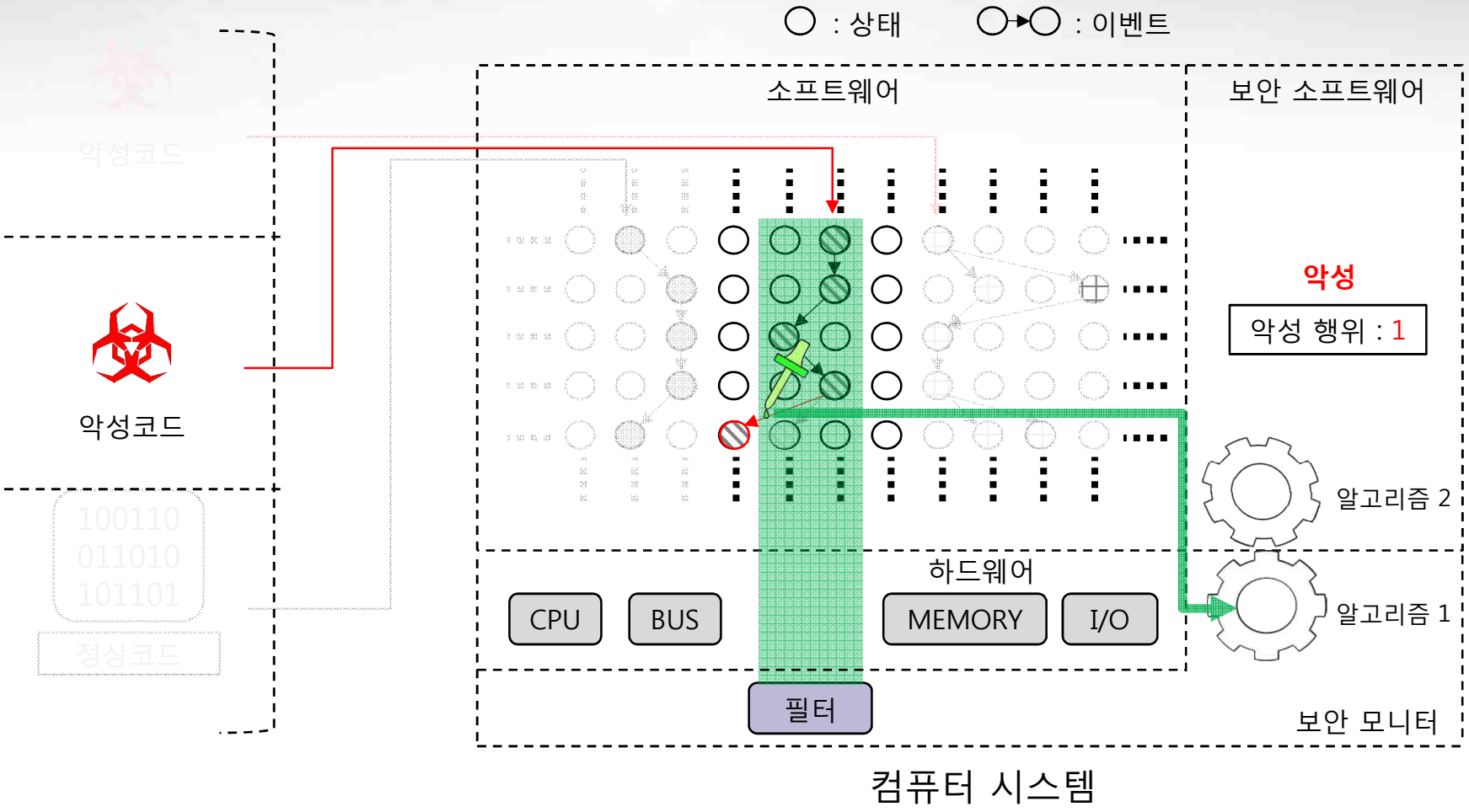


기존 호스트 H/W보다 낮은 Layer에 위치해
높은 특권을 가진 보안 모니터 H/W

컴퓨터 시스템
기존 순수 S/W 형태 보안 알고리즘
가속화를 위해 일부 H/W 내부로 흡수

보안 기능이 집적된 하드웨어를 통한 감시 예시

H/W 지원하의 효율적이고 안전한 악성코드의 행위 감시



목차

Security의 정의 ➔ CIA

- Confidentiality
  - Preventing unauthorized **reading** of information
  - Examples:
    - Software license keys
    - Banking info
- Integrity
  - Preventing, or at least detecting, unauthorized **writing**
  - Examples:
    - Pharming
    - Banking transaction
- Availability
  - Information and services are **accessible in a timely fashion to authorized people or systems**
  - Examples:
    - Denial of Service (DoS), Distributed Denial of Service (DDoS)

악성코드(Malware)는 어떻게 수행되게 되나?

- Software vulnerability: Exploit
  - Software may have bugs that allows adversaries to run their own code sequence by exploiting some of them.
  - any weakness in an overall system that makes it open to attack due to system administration/configuration flaws or dangerous user behavior
- Social engineering
  - Tricks user into running/installing
  - Ex) claiming itself to be a useful and benign software
- Attacker with local access
  - a malicious visitor ➔ access to your PC ➔ download and install any malware.

Exploiting vulnerabilities

- Malware is designed to exploit **vulnerabilities** in an OS or applications.
  - Unguarded buffer overflow in OS command allows attacker to run arbitrary command, gain root access, etc.
  - Failure to validate user input
  - Allowing ActiveX controls to be run from scripts
- Malware is usually created in machine/assembly code.
  - It is quite difficult and time-consuming for security professionals to analyze malicious attacks in the code.
  - It demands expertise in low-level machine and OS as well as the original code under attack.

Classification of vulnerability

- One that allows malicious **read**
  - Open SSL Vulnerability (CVE-2014-0160): announced Apr 07, 2014
  - Allow adversaries to read the secret key of the server

Adversaries can pretend to be the victim

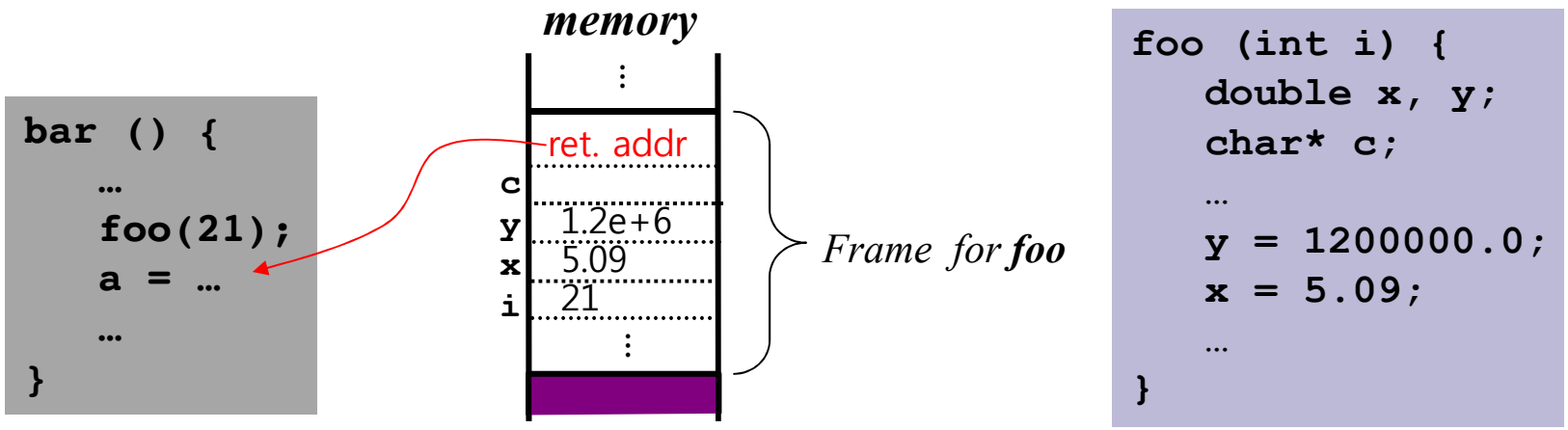
- One that allows malicious **write**
  - The pharming example: malicious write to hosts

- One that allows **execution of malicious code**
  - The most powerful: also allows write/read
  - The yahoo example
    - Malicious ad run its exploit to make **the payloads execute on the victim's machine**

Vulnerability example: buffer overflow

- In C/C++, array bounds are not checked at run time.
- Memory layout at run time is usually fixed and known.
  - When a function is invoked, all local variables and other information necessitated for the invocation are maintained within a segment of memory called a **frame**.
  - **Return address** is the pointer to the location inside the caller.

    → It is usually at a defined distance after local variables.



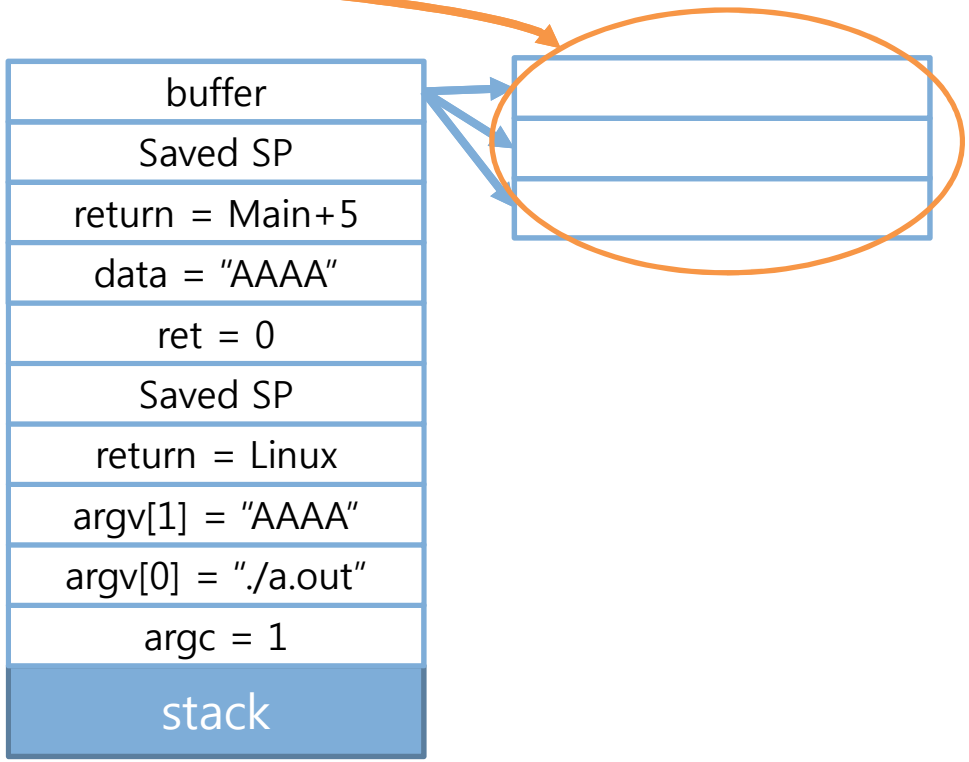
Buffer overflow vulnerability: execution

More realistic

text

```
int copybuffer(char data[]){
        char buffer[12];
        strcpy(buffer, data);
        printf("%s₩n",buffer);
        return 1;
}

int main(int argc, char* argv[]) {
        int ret = 0;
        ret = copybuffer(argv[1])
        return ret;
}
```

data




```
$ ls
example.c
$ gcc example.c
$ ls
a.out example.c
$ ./a.out AAAA
```

Buffer overflow vulnerability: execution

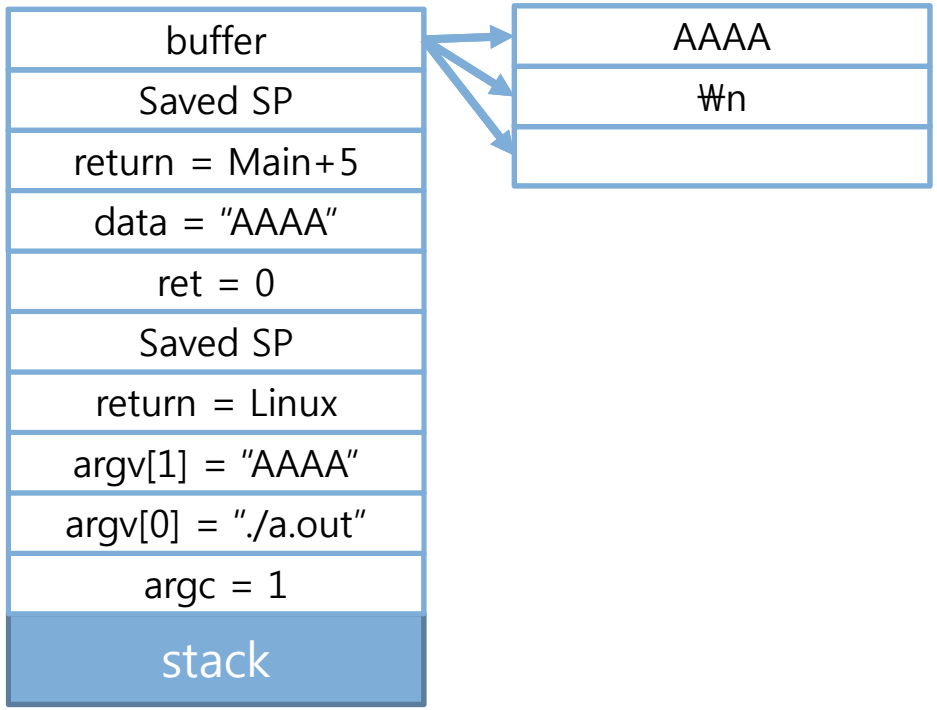
More realistic

text

```
int copybuffer(char data[]){
        char buffer[12];
        strcpy(buffer, data);
        printf("%s₩n",buffer);
        return 1;
}

int main(int argc, char* argv[]) {
        int ret = 0;
        ret = copybuffer(argv[1])
        return ret;
}
```

```
$ ls
example.c
$ gcc example.c
$ ls
a.out example.c
$ ./a.out AAAA
```

data



Buffer overflow vulnerability: overflows!!

More realistic

text

```
int copybuffer(char data[]){
        char buffer[12];
        strcpy(buffer, data);
        printf("%s₩n",buffer);
        return 1;
}

int main(int argc, char* argv[]) {
        int ret = 0;
        ret = copybuffer(argv[1])
        return ret;
}
```

```
$ ls
example.c
$ gcc example.c
$ ls
a.out example.c
$ ./a.out AAAAAAAAAAAAAA
```

14 characters

data

buffer
Saved SP
return = Main+5
data = "AAAA"
ret = 0
Saved SP
return = Linux
argv[1] = "AAAA"
argv[0] = "./a.out"
argc = 1

stack

AAAA
AAAA
AAAA
AA₩n

overflows!

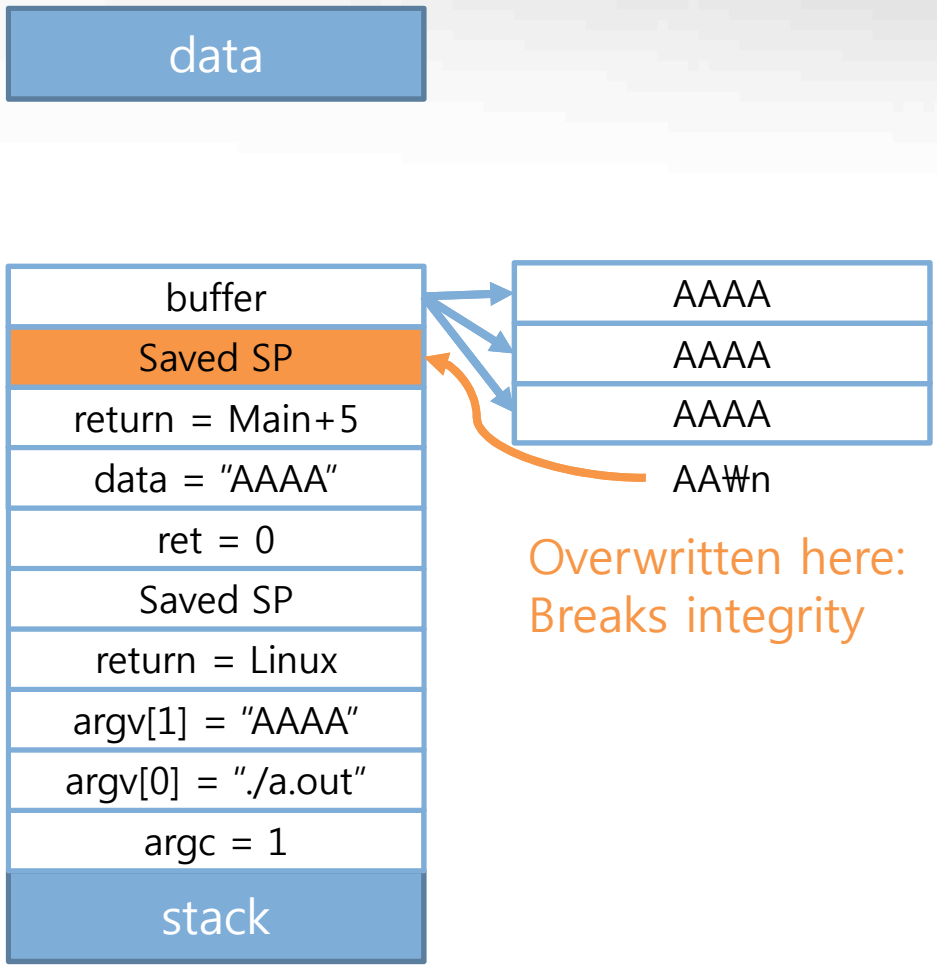
Exploiting buffer overflow: why dangerous?

More realistic

text

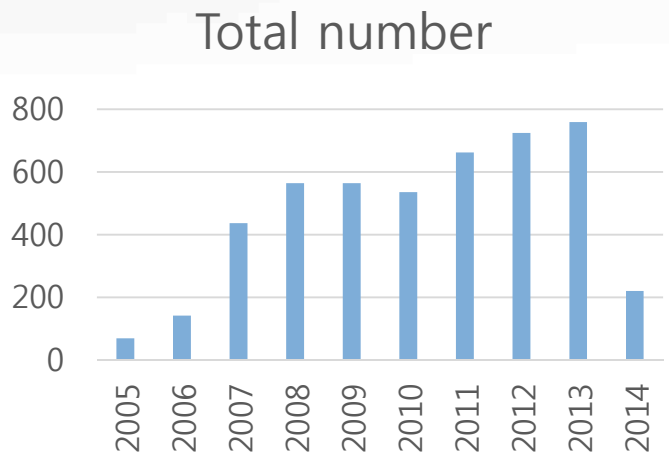
```
int copybuffer(char data[]){
        char buffer[12];
        strcpy(buffer, data);
        printf("%s₩n",buffer);
        return 1;
}

int main(int argc, char* argv[]) {
        int ret = 0;
        ret = copybuffer(argv[1])
        return ret;
}
```

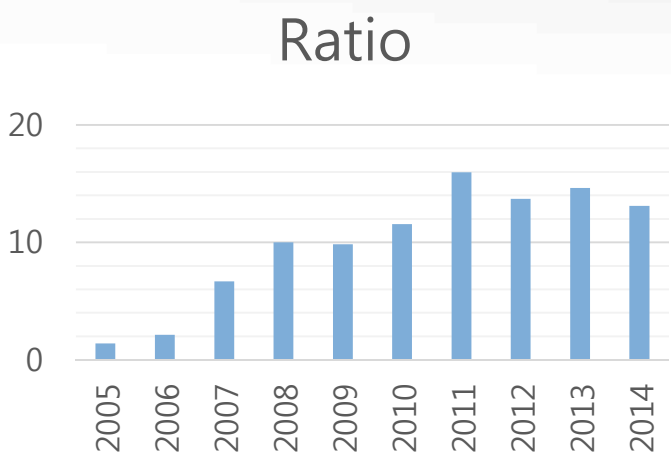
```
$ ls
example.c
$ gcc example.c
$ ls
a.out example.c
$ ./a.out AAAAAAAAAAAAAA
```



Buffer overflow vulnerability: trend

- Graphs are the number and ratio of buffer overflow vulnerabilities

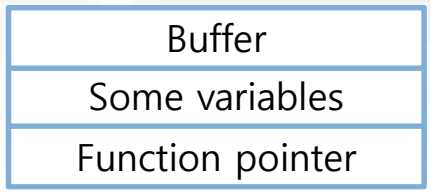




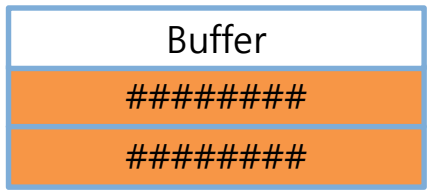
- Recent example: CVE-2014-1758
  - published Apr 08, 2014
  - Stack-based buffer overflow in **Microsoft Word 2003 SP3** allows remote attackers **to execute arbitrary code** via a **crafted document**

Heap attacks

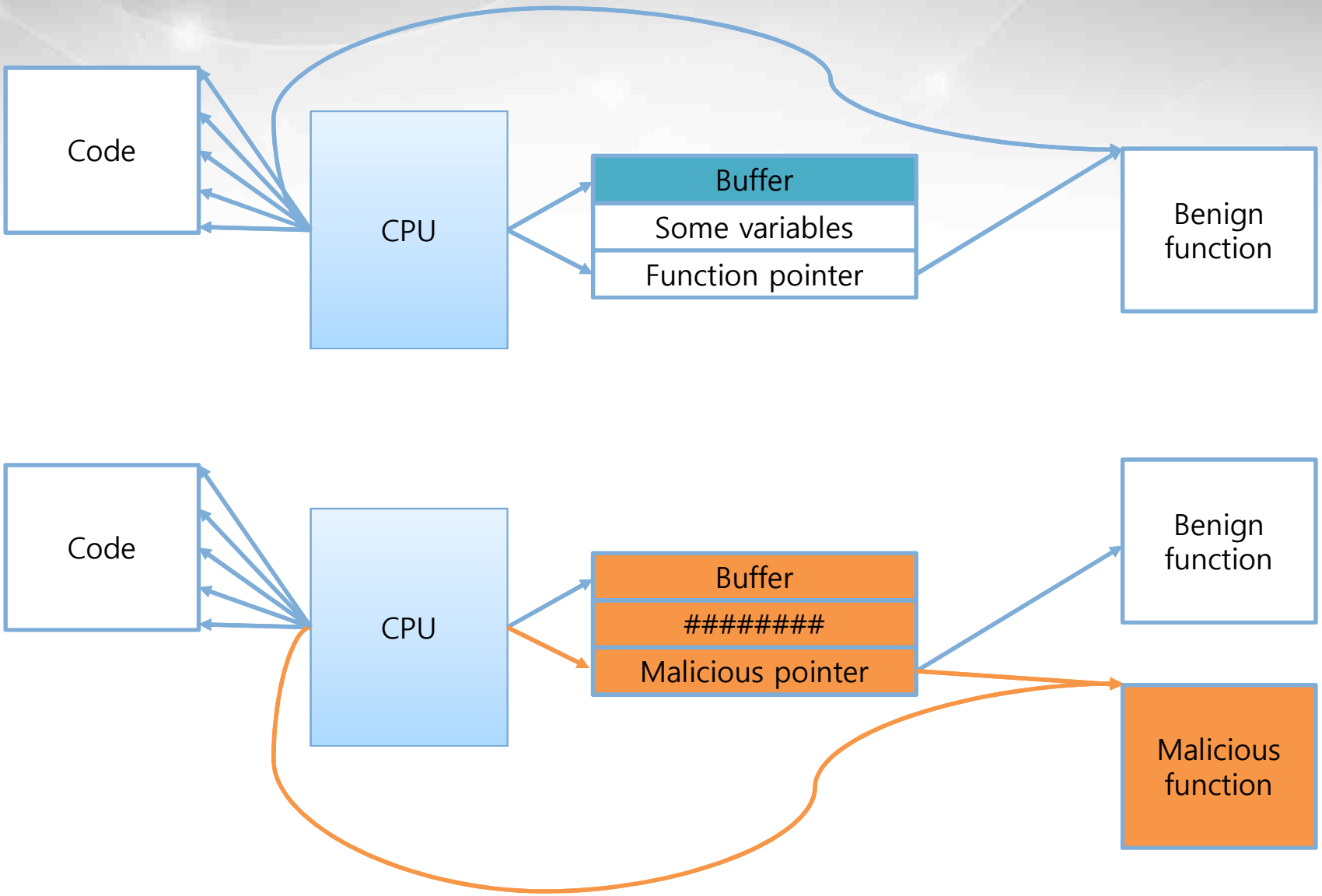
- Memory blocks in heap sometimes look like:



- The function pointer can also be manipulated by making the buffer overflow



Heap attacks: flow



Unsafe functions

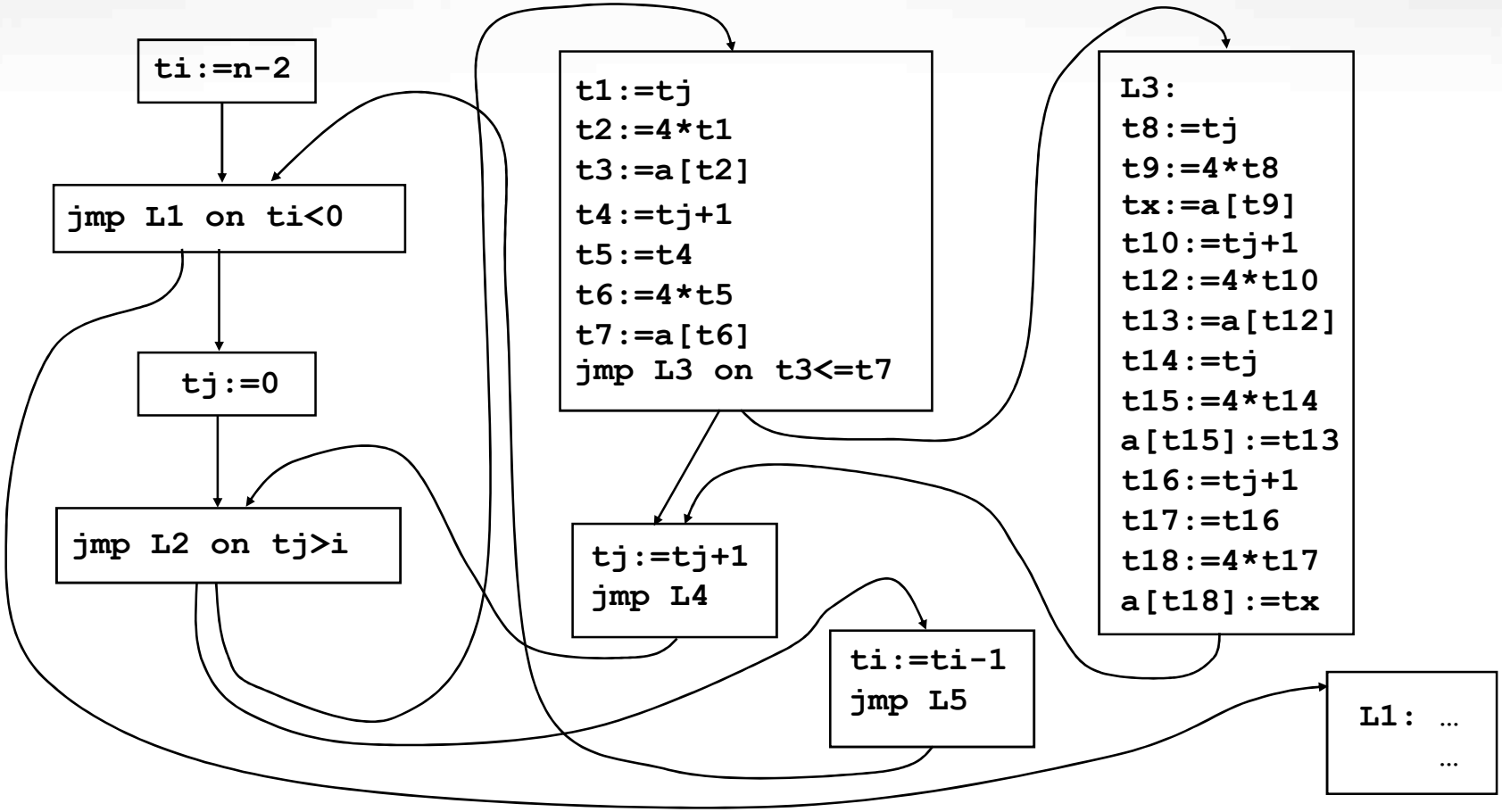
- Many libc function are unsafe: causes buffer overflow
  - strcpy(char *dest, const char *src)
  - strcat(char *dest, const char *src)
  - gets(char *s)
  - scanf(const char *format, ...)
  - sprintf(char *str, const char *format, ...)
  - ...
- Why we don't eliminate them?
  - Legacy codes use them: we cannot re-implement all of them
  - They are not the sole causes of the vulnerabilities

Control Hijacking Attacks

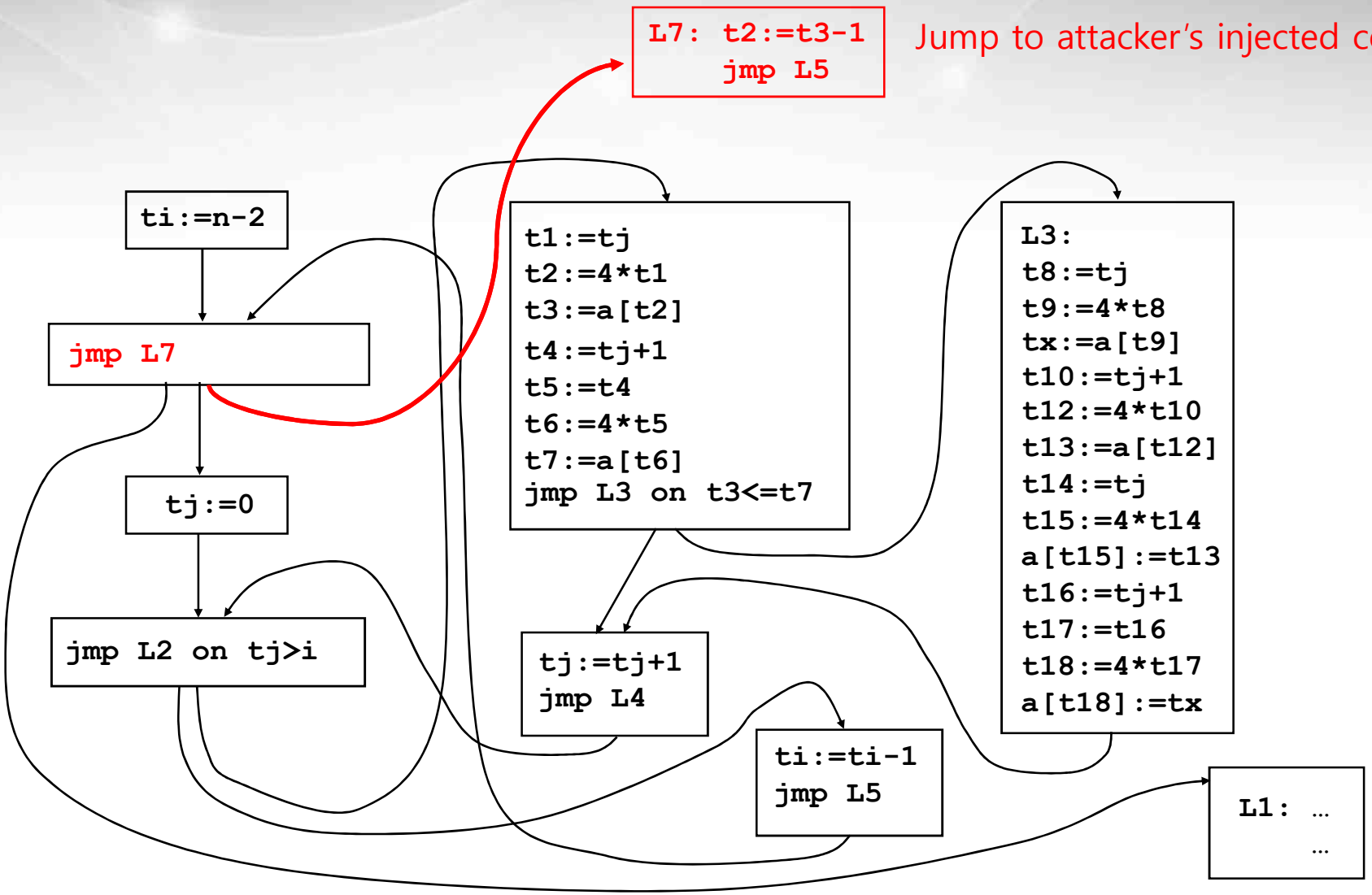
- Buffer overflows are usually exploited to launch *control hijacking attacks*
  - Control hijacking is widely used by adversaries ➔ 96 % of kernel-level rootkits employs control hijacking
  - Overwrites some data structures in a victim program that affects its control flow ➔ Ex) function pointer, return address
  - Changes the control flow to the addresses to which attackers want to jump
- At those points, the attacker
  - Injected code (a.k.a. code injection attack)
    - Prepare a code before exploiting buffer overflow vulnerability and jump to it
  - Existing code (a.k.a. return to libc attack)
    - Exploit buffer overflow to an existing code such as libc function, which is not supposed to be called

Example : Control Flow Graph (CFG)

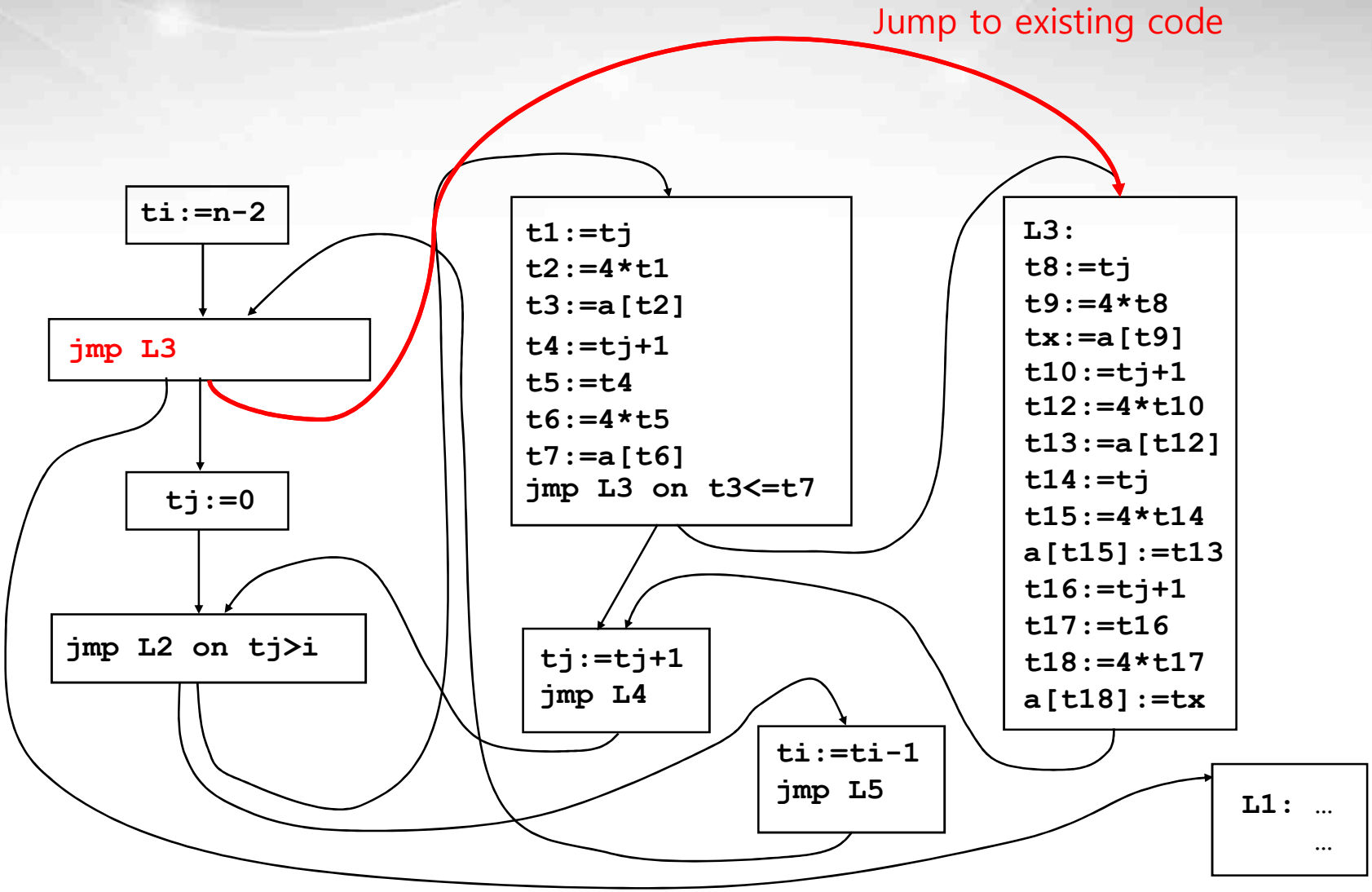
- CFG for the bubble sort code



Example : CFI violation

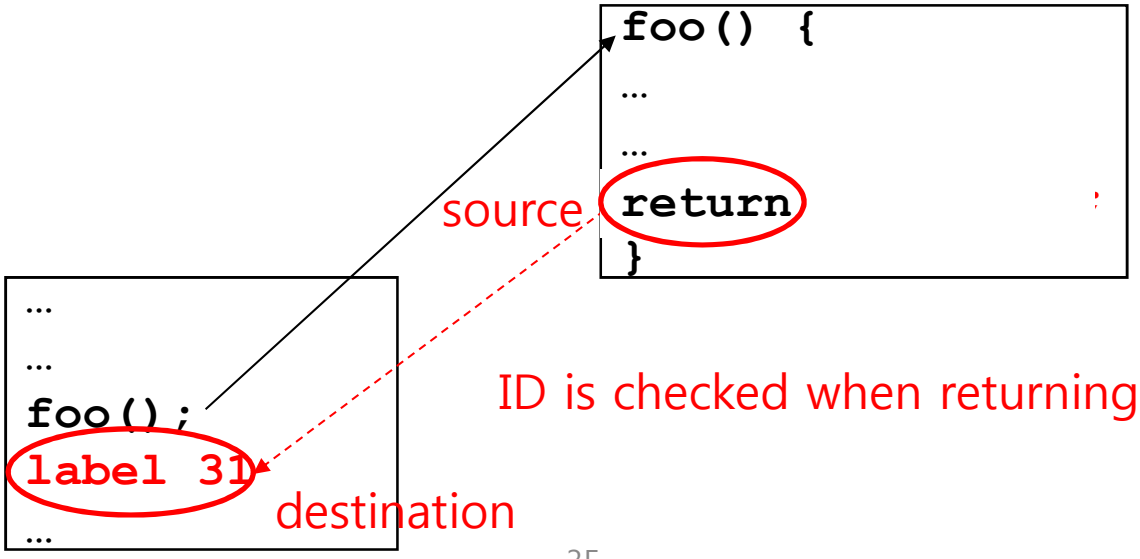


Example : CFI violation



Code restructuring for CFI enforcement

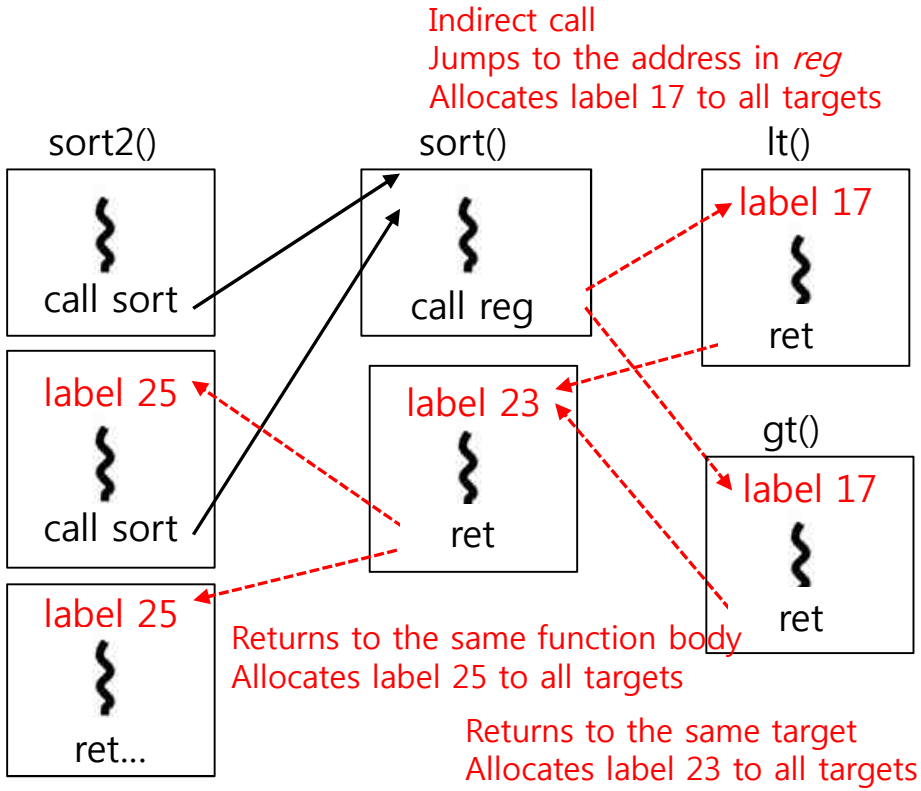
- M.Abadi et al., "Control-Flow Integrity", CCS '05
  - The first work of CFI enforcement
  - Introduces CFI enforcement to prevent illegal control hijacking
- Principle
  - Allocates IDs to every control flow targets
    - Making source/destination pairs
  - Checks the ID whether the control flow is transferred to the valid target
    - Instrument (or rewrite) program binary to add the ID checking codes



Example : Indirect Call

- Indirect calls can have multiple targets
- Allocates the same ID to the targets of the call
- Multiple returns can have the same ID for a target

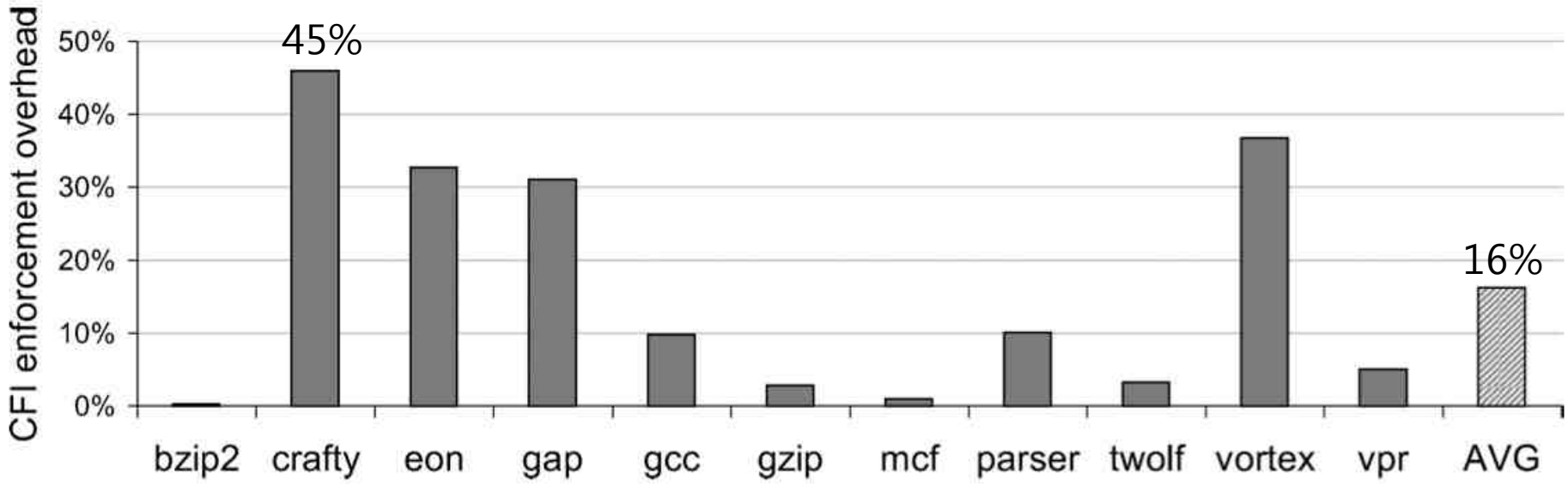
```
bool lt (int x, int y) {
    return x < y;
}
bool gt (int x, int y) {
    return x > y;
}

sort2 (int a[], int b[], int
len)
{
    sort (a, len, lt);
    sort (b, len, gt);
}
```



Limitation : Performance Overhead

Size: increase 8% avg

Time: increase 0-45%; 16% avg



Execution overhead of inlined CFI enforcement on SPEC2000 benchmarks.

Buffer overflow prevention: Compiler

- Compiler's program analysis techniques may also be useful.
- Modern day compilers can easily see that the potential access range of buf[] is [0:∞].
- A compiler may remove such vulnerability by changing the source or assembly code when it generates code.

```
void foo(int n) {
  char buf[64];
  printf("Enter your data as a string.₩n");
  for (int i = 0; i < 64; i++)
    if (!(buf[i] = getchar(c)) break; // Now the bound checked!
  …
}
```

Now the compiler guarantees that the possible access range is [*0:63*].

- But this is against the C/C++ language rule

결론

- 보안도 일정의 연산
  - 성능 제약에 취약
  - 보다 효율적인 보안 연산 기술의 개발을 보다 안전한 시스템을 보장
- 효율적인 보안 연산을 위한 기술들
  - OS 수준의 최적화 및 보안 알고리즘 자체의 개선을 통한 성능 향상
  - 컴파일러 또는 코드 분석 기술의 적용을 통한 성능 개선
  - 하드웨어 기술을 통한 성능 향상 -- 단 제한적인 영역에 적용이 바람직
- 보안 기술의 진화에 따른 연산 기법의 진화
  - 악성 코드의 대응 기술의 발달로 악성 코드 자체도 계속 진화
  - 현재로는 대부분의 알려진 공격 기법들은 발견 및 방지 기법들 개발됨
    - 코드 분석 기술 및 하드웨어 기술의 적용 방법은?